Notes
CSC-211
Data Structures

Dr. Qixin Deng

January 2025

Contents

1 Introduction

Before we begin exploring various data structures and their wide-ranging applications, it is crucial to establish our method of study. In broad terms, we will adopt the following standard procedure:

1. **Motivation:** Introduce a new abstract data type or data structure by considering illustrative examples, together with reflecting on our existing knowledge.
2. **Structure and Operations:** Provide a detailed description of how the data structure is organized and how its operations are implemented.
3. **Correctness:** Justify why these operations function as intended and achieve the desired outcomes.
4. **Performance Analysis:** Evaluate the running-time performance of the data structure's operations to assess their computational efficiency.

We already have experience with basic data structures like Python lists. A natural question arises: why should we devote so much effort to studying a variety of data structures when arrays or simple pointers—readily available in many programming languages—can be used to store and manipulate data?

While it is true that any data storage requirement or operation can, in principle, be realized with these primitives, the key motivation for designing, refining, and specializing data structures lies in the potential performance gains. By carefully selecting or inventing a suitable data structure for a particular problem, we can:

- **Reduce Time Complexity:** Achieve faster algorithms for searching, insertion, deletion, or other operations. For instance, using trees or hash tables often outperforms naive array-based approaches in specific scenarios.
- **Improve Space Utilization:** Some specialized data structures can be more memory-efficient than primitive structures, especially under large-scale data or constraints on memory usage.
- **Enhance Code Organization and Reusability:** Abstract data types provide clear interfaces and can be reused in different parts of a program or in future projects.
- **Clarify Problem-Solving Strategies:** Exploring different data structures naturally leads to recognizing problem patterns (e.g., tree-based, graph-based, or queue-based solutions), streamlining the development of robust algorithms.

With these points in mind, our journey through data structures will emphasize not only the *mechanics* of each data structure but also the *reasoning* behind why and when a particular structure provides significant advantages over basic constructs. Such an understanding empowers us to build programs and systems that are both efficient and maintainable.

2 What Is Running Time?

When we write an algorithm and want to determine how fast it runs, one idea is to simply measure how many seconds it takes on a computer. While this *empirical* approach can be helpful in certain contexts, it also depends heavily on the hardware being used and the level of background computation on that machine. For instance, the same program might behave differently on a powerful server compared to a small laptop, or even on the same machine if other processes are active.

In many situations, especially in high-level software development, we do not control the exact hardware on which the code will run. Moreover, measuring real time can be influenced by networking, background tasks, and operating system schedules. Because of this, *empirical measurements of runtime*, though useful for performance tuning, are often less useful as a foundation for theoretical or language-agnostic analyses.

2.1 Counting "Basic Operations"

Instead of recording the wall-clock time, we introduce an *abstract measure* of runtime: *the number of basic operations* an algorithm performs. This approach frees us from the details of a particular machine. Of course, the term "basic operation" itself can be somewhat ambiguous:

- Should a single variable assignment count as one basic operation?
- Should a call to a printing function count as one or more operations, given that it takes additional time to actually display text to a user?
- What about function call overhead or memory allocation?

Despite these nuances, the core idea is that the *qualitative* growth of the number of operations matters more than the exact count. Different methods of counting might yield slight differences (e.g., n, $2n$, $3n + 5$, $\dots$), but if all forms grow proportionally to n, we say the algorithm has a *linear* running time.

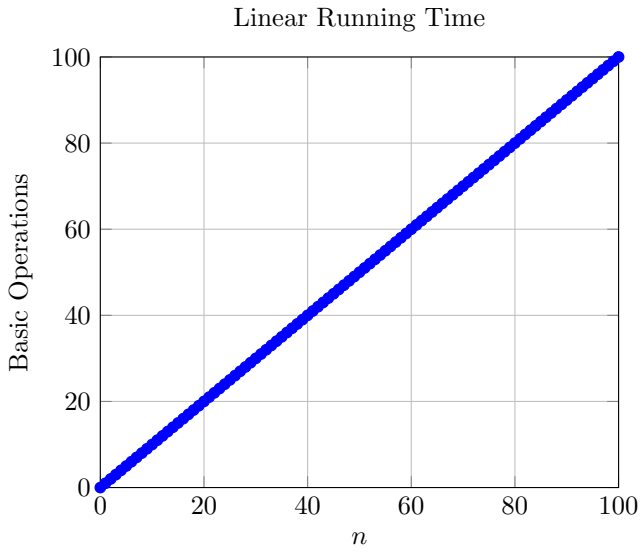
Below, we explore several Python functions whose running times illustrate these key ideas. We will count the number of calls to `print` as our proxy for the number of "basic operations," but one could choose a different primitive operation and arrive at the same essential insights.

2.2 Examples of Different Growth Rates

2.2.1 Linear Running Time

Consider a function that prints the first n natural numbers (from `0` to `n-1`):

```
def print_integers(n: int) -> None:
    for i in range(0, n):
        print(i)
```

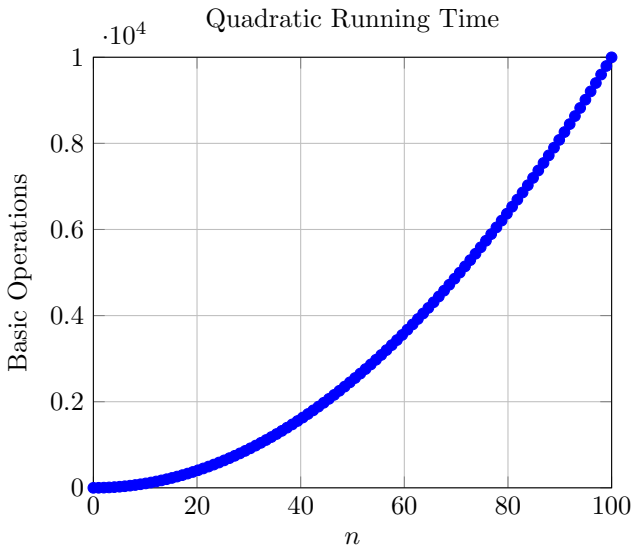


This function performs one `print` call per loop iteration, and the loop runs n times. Hence, there are n basic operations (by our chosen measure). This linear growth means that when n doubles, the number of print statements (and thus the total work, in our simplified model) also roughly doubles.

2.2.2 Quadratic Running Time

Next, consider printing all pairs of integers between `0` and `n-1`:

```
def print_pairs(n: int) -> None:
    """Print all combinations of pairs of the first n natural numbers."""
    for i in range(0, n):
        for j in range(0, n):
            print(i, j)
```

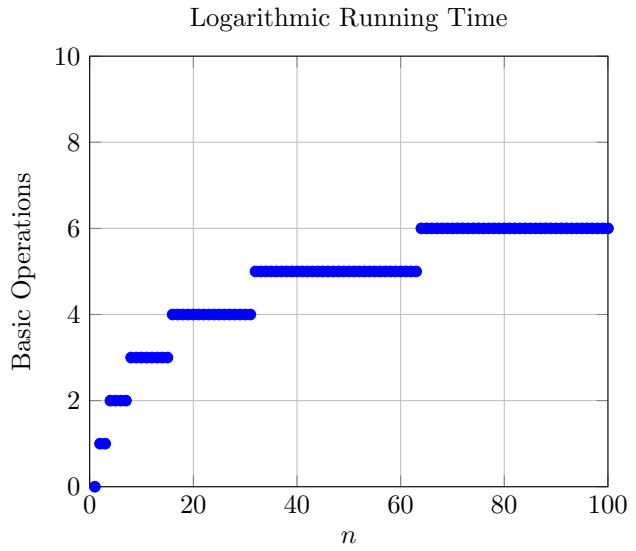


Here, the *outer* loop runs n times. For each iteration of the outer loop, the *inner* loop also runs n times. Thus, the total number of `print` calls is $n \times n = n^2$. We say this function has *quadratic* running time.

2.2.3 Logarithmic Running Time

Next, consider a function that prints all powers of two strictly less than a given positive integer `n`:

```
import math
def print_powers_of_two(n: int) -> None:
    """Print the powers of two that are less than n.
    """
    for i in range(0, math.ceil(math.log2(n))):
        print(2 ** i)
```



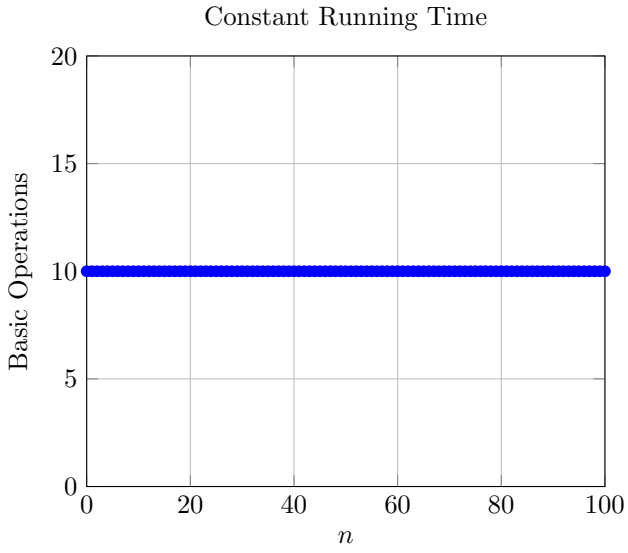
In this example, the loop runs approximately $\log_2(n)$ times. Hence, there are about $\log_2(n)$ `print` calls. When n increases significantly, $\log_2(n)$ grows much more slowly than n itself, so we categorize this as a *logarithmic* running time.

2.2.4 Constant Running Time

Finally, observe that we can have code whose number of operations does *not* depend on n at all:

```
def print_ten(n: int) -> None:
    """Print n ten times."""
```

```
    for i in range(0, 10):
        print(n)
```



No matter what integer n we provide, the loop always runs exactly 10 times. Hence, this function exhibits a *constant* running time. As far as our measure is concerned, it simply does 10 `print` operations.

2.3 Refining the Notion of "Basic Operations"

In these examples, we chose `print` calls as our basic operation. However, one could argue that variable assignments, loop increments, or function call overhead should also be included. Even more, we could factor in that `print` calls often take longer than simple arithmetic operations. Consequently, different analysts might come up with expressions like:

$$n, \quad n+3, \quad 5n+2, \quad \dots$$

all of which represent *linear* growth rates, even if their constant coefficients differ. Similarly, if the function is fundamentally doing n^2 work, small changes in counting will not alter the fact that the growth rate is quadratic.

This observation underscores a key goal of runtime analysis: **capture how the cost of an algorithm scales with the size of the input, without getting lost in the weeds of machine-specific details or constant factors.** In other word, we focus on just one central measure of performance: the relationship between an algorithm's input size and the number of basic operations the algorithm performs. **We do not try to precisely quantify the exact number of such operations, but instead categorize how the number grows relative to the size of the algorithm's input.**

2.4 Mathematical Shorthand and Notation

Symbol	English Expression(s)
$\forall$	"for all," "for each"
$\exists$	"there exists," "there is"
$\in$	"in," "is an element of"
$\implies$	"implies"
$\iff$	"if and only if"

Symbol	Name	Typical Elements
$\mathbb{N}$	Natural Numbers	$\{1, 2, 3, \dots\}$
$\mathbb{Z}$	Integers	$\{\dots, -2, -1, 0, 1, 2, \dots\}$
$\mathbb{Q}$	Rational Numbers	$\left\{ \frac{p}{q} \mid p \in \mathbb{Z},\, q \in \mathbb{Z} \setminus \{0\} \right\}$
$\mathbb{R}$	Real Numbers	All decimal expansions, including irrationals (e.g., $\sqrt{2}, \pi$)

2.5 Four Kinds of Dominance

Asymptotic analysis is a fundamental tool for studying the growth rates of functions. In the context of algorithms, it enables us to analyze their efficiency by focusing on the relationship between the size of the input and the number of basic operations performed. This approach is critical for understanding how different algorithms scale with increasingly large inputs. In algorithm analysis, we often deal with functions that map the natural numbers to non-negative real numbers:

$$f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}.$$

These functions represent the number of operations performed by an algorithm as a function of input size. The goal of asymptotic analysis is to compare the `long-term growth rates` of such functions, while abstracting away constants and lower-order terms that do not significantly affect their behavior for large inputs.

2.5.1 Absolute Dominance.

Definition 1 (Absolute Dominance)**.** *Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say that g* is absolutely dominated by *f if and only if*

$$\forall n \in \mathbb{N}, \quad g(n) \leq f(n).$$

Example 1. *Let $f(n) = n^2$ and $g(n) = n$. Prove that g is absolutely dominated by f.*

Translation. This is a direct application of the definition: we need to show that for every $n \in \mathbb{N}$, $g(n) \leq f(n)$.

Proof. Let $n \in \mathbb{N}$. We want to show $n \leq n^2$.

Case 1: Suppose $n = 0$. Then $n^2 = 0$, so $n = 0 \leq 0 = n^2$, and the claim holds.

Case 2: Suppose $n \geq 1$. Since $n \geq 1$, multiplying both sides by n yields

$$n^2 \geq n,$$

which is the same as $n \leq n^2$. Hence in both cases $n \leq n^2$, completing the proof.

Remark 1. *Absolute dominance can be too strict. For instance, if we take $g(n) = 2n$ and $f(n) = n^2$, we see that $g(n) \leq f(n)$ holds everywhere except for $n = 1$, yet g is still not absolutely dominated by f under this definition.*

2.5.2 Dominance up to a Constant Factor.

To address the limitations of absolute dominance, we say g is **dominated by f up to a constant factor** if there exists a positive constant $c > 0$ such that for all $n \in \mathbb{N}, g(n) \leq c \cdot f(n)$. This relaxation allows us to ignore multiplicative constants, making it much more useful in algorithm analysis.

Definition 2 (Dominated up to a Constant Factor)**.** *Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say that g* is dominated by *f* up to a constant factor *if there exists $c > 0$ such that*

$$\forall n \in \mathbb{N}, \quad g(n) \leq c \cdot f(n).$$

Example 2. *Consider $f(n) = n^2$ and $g(n) = 2n$. Prove that g is dominated by f up to a constant factor.*

Translation. We need to show there is some $c \in \mathbb{R}^{>0}$ such that for all $n \in \mathbb{N}$,

$$g(n) \leq c \cdot f(n).$$

Equivalently, we must prove that $2n \leq c \cdot n^2$ for all $n \in \mathbb{N}$.

The phrase "dominated up to a constant factor" highlights that we only care about whether one function can be bounded by a constant multiple of another. We already know that $n \leq n^2$ for $n \geq 1$, so multiplying n by 2 naturally suggests multiplying n^2 by the same constant. In other words, if n is dominated by n^2, then $2n$ should be dominated by $2n^2$.

Proof. **Claim:** $g(n) \leq c \cdot f(n)$ holds with $c = 2$.

Case 1: $n = 0$. Then $g(0) = 2 \cdot 0 = 0$ and $f(0) = 0^2 = 0$. Hence $g(0) = 0 \leq 2 \cdot 0 = 0$.

Case 2: $n \geq 1$. We have

$$n \geq 1 \quad \Longrightarrow \quad n^2 \geq n \quad \Longrightarrow \quad 2n^2 \geq 2n.$$

Thus $2n \leq 2n^2$, which shows

$$g(n) \;=\; 2n \;\leq\; 2 \cdot n^2 \;=\; 2 \cdot f(n).$$

In both cases, $g(n) \leq 2 \cdot f(n)$. Therefore g is dominated by f up to the constant factor $c = 2$.

Intuitively, "dominated by up to a constant factor" allows us to ignore multiplicative constants in our functions. This will be very useful in our running time analysis because **it frees us from worrying about the exact constants used to represent numbers of basic operations**: n, 3n, and 12n are all equivalent in the sense that each one dominates the other two up to a constant factor.

2.5.3 Eventual Dominance

Sometimes, functions only exhibit the desired relationship for sufficiently large input sizes. We say g is **eventually dominated by** f if there exists a positive threshold n_0 such that for all $n \geq n_0, g(n) \leq f(n)$. This definition enables us to ignore the behavior of functions for small inputs.

Definition 3 (Eventually Dominated)**.** *Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say that g* is eventually dominated by *f if there exists some $n_0 \in \mathbb{R}^{>0}$ such that for all $n \in \mathbb{N}$ with $n \geq n_0$, we have*

$$g(n) \leq f(n).$$

Example 3. *Let $f(n) = n^2$ and $g(n) = n + 90$. Prove that g is eventually dominated by f.*

Translation. We need to exhibit an $n_0 \in \mathbb{R}^{>0}$ such that, whenever $n \geq n_0$,

$$g(n) \leq f(n).$$

In other words, we want to show there is a threshold n_0 beyond which $(n + 90) \leq n^2$.

Rather than trying to find a constant factor to multiply f by, we look only at sufficiently large n so that $n + 90 \leq n^2$. Once n is big enough, the quadratic term n^2 grows faster than the linear term n, even with an additional constant 90. The key point is that we get to pick any n_0 that makes the inequality hold for all larger n.

Proof. Choose $n_0 = 90$. Let $n \in \mathbb{N}$, and assume $n \geq n_0$. We need to prove

$$n + 90 \ \leq \ n^2.$$

Start with the left-hand side and chain inequalities to reach n^2:

$$n + 90 \ \leq \ n + n \ (\textit{since } 90 \leq n) = \ 2n \ \leq \ n \cdot n \ (\textit{since } 2 \leq n) = \ n^2,$$

where the first step uses $n \geq 90$ (so $90 \leq n$), and the second step uses $n \geq 2$ (making $2n \leq n^2$). Therefore, for all $n \geq 90$, we have $(n + 90) \leq n^2$.

Hence g is eventually dominated by f with $n_0 = 90$.

Intuitively, this definition allows us to ignore "small" values of n and focus on the long term, or asymptotic, behaviour of the function. This is particularly important for ignoring the influence of slow-growing terms in a function, which may affect the function values for "small" , but eventually are overshadowed by the faster-growing terms. In the above example, we knew that n^2 grows faster than n, but because an extra $+90$ was added to the later function, it took a while for the faster growth rate of n^2 to "catch up" to $n + 90$.

2.5.4 Big-O Notation

Definition 4 (Big-O Notation)**.** *Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say that g is* eventually dominated by *f* up to a constant factor *if there exist $c, n_0 \in \mathbb{R}^{>0}$ such that for all $n \in \mathbb{N}$,*

$$n \geq n_0 \implies g(n) \ \leq \ c \cdot f(n).$$

In this situation, we also write $g \in O(f)$ and say g is Big-O of *f.*

Formally, we define

$$O(f) \ = \ \big\{ g : \mathbb{N} \to \mathbb{R}^{\geq 0} \ \big| \ \exists\, c, n_0 \in \mathbb{R}^{>0} \ \textit{s.t.} \ \forall\, n \in \mathbb{N}, n \geq n_0 \implies g(n) \leq c\, f(n) \big\}.$$

Example 4. *Let $f(n) = n^3$ and $g(n) = n^3 + 100n + 5000$. Prove that $g \in O(f)$.*

Translation. We must show that there is some $c > 0$ and some threshold $n_0 > 0$ such that, for every integer $n \geq n_0$,

$$n^3 + 100n + 5000 \ \leq \ c\, n^3.$$

Equivalently, beyond some sufficiently large n, we can bound $n^3 + 100n + 5000$ by a constant multiple of n^3.

One way to see that $g(n)$ can be "eventually bounded" by $f(n)$ (up to a constant) is to notice that n^3 grows faster than n or a fixed constant, so eventually n^3 will be larger than any linear or constant term. We only need to choose

how large n must be in order to make n^3 dominate $100n$ and 5000, and we also need to choose *how large* we make our constant c to absorb the extra pieces $100n$ and 5000.

Approach 1: Focus on choosing n_0.

We try to keep c small (say $c = 3$) but choose n_0 large enough that

$$n^3 \;\geq\; 100n \quad \text{and} \quad n^3 \;\geq\; 5000$$

for all $n \geq n_0$.

- Clearly $n^3 \geq n^3$ always holds (with an implicit coefficient of 1). - The condition $n^3 \geq 100n$ is satisfied for $n \geq 10$ (since $10^3 = 1000 \geq 100 \cdot 10$). - The condition $n^3 \geq 5000$ is satisfied for $n \geq \sqrt[3]{5000} \approx 17.1$.

Hence if we let $n_0 = \max\{10, \sqrt[3]{5000}\}$ (which is effectively about 17.1), then for all $n \geq n_0$ we have:

$$n^3 + 100n + 5000 \;\leq\; n^3 + n^3 + n^3 \;=\; 3\,n^3.$$

Thus it suffices to pick $c = 3$ and $n_0 \approx 17.1$, proving $g(n) \leq c\,f(n)$ whenever $n \geq n_0$.

Proof (Approach 1). Take $c = 3$ and $n_0 = \max\{10, \sqrt[3]{5000}\}$. If $n \geq n_0$, then

$$n^3 \;\geq\; 100n, \quad n^3 \;\geq\; 5000, \quad n^3 \;=\; n^3.$$

Adding these yields

$$n^3 + 100n + 5000 \;\leq\; n^3 + n^3 + n^3 \;=\; 3\,n^3 \;=\; c\,n^3.$$

Approach 2: Focus on choosing c.

Alternatively, we can choose n_0 to be quite small (for instance, $n_0 = 1$) but then let c be large enough to absorb the constant and linear terms:

$$n^3 \leq 1 \cdot n^3, \qquad 100n \leq 100n^3, \qquad 5000 \leq 5000n^3 \quad (\text{for } n \geq 1).$$

Summing up these three inequalities gives us

$$n^3 + 100n + 5000 \;\;\leq\;\; n^3 + 100\,n^3 + 5000\,n^3 \;=\; 5101\,n^3.$$

Hence with $c = 5101$ and $n_0 = 1$, we again have $g(n) \leq c\,f(n)$ for all $n \geq n_0$.

Proof (Approach 2). Take $c = 5101$ and $n_0 = 1$. For $n \geq n_0$, observe:

$$n^3 \leq n^3, \quad 100n \leq 100n^3, \quad 5000 \leq 5000n^3.$$

Hence

$$n^3 + 100n + 5000 \;\leq\; n^3 + 100n^3 + 5000n^3 \;=\; 5101\,n^3 \;=\; c\,n^3.$$

Both approaches demonstrate that $g(n) \leq c\,f(n)$ once n is large enough. Thus $g \in O(f)$.

Big-O notation lets us ignore small values of n (as in eventual dominance) and also ignore the exact constant multiplier (as in dominance up to a constant factor). Symbolically:

$$g \in O(f) \quad \Longleftrightarrow \quad \exists\, c, n_0 > 0 : \forall n \geq n_0,\; g(n) \leq c\,f(n).$$

2.6 Omega and Theta

Big-O notation provides a convenient way to describe the long-term growth of functions. However, it does not require an exact growth description. For instance, $n + 10$ lies in $O(n^{100})$, yet this hardly gives a precise sense of how $n + 10$ grows. Hence, Big-O expresses `upper bounds` on a function's growth but cannot distinguish between a tight bound and a large overestimate.

2.6.1 Omega (Ω)

To capture a `lower` bound, we introduce the Omega (Ω) notation:

Definition 5. *Let $f, g\colon \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say g is `Omega of` f, written $g \in \Omega(f)$, if there exist constants $c > 0$ and $n_0 > 0$ such that for all $n \in \mathbb{N}$ with $n \geq n_0$,*

$$g(n) \;\geq\; c\,f(n).$$

In other words, if g is in $\Omega(f)$, then f serves as a lower bound on the growth of g for sufficiently large n. For example, $n^2 - n$ can be shown to be in $\Omega(n^2)$ because $n^2 - n \geq \frac{1}{2}n^2$ for large n.

2.6.2 Theta (Θ)

We combine Big-O (an upper bound) and Omega (a lower bound) into a single concept when g grows `at the same rate` as f asymptotically:

Definition 6. *Let $f, g\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$. We say g is* Theta of f*, written $g \in \Theta(f)$, if g is in both $O(f)$ and $\Omega(f)$. Equivalently, there exist positive constants c_1, c_2, and n_0 such that for all $n \geq n_0$,*

$$c_1\, f(n) \;\leq\; g(n) \;\leq\; c_2\, f(n).$$

A *Theta bound* therefore means f and g have the same *asymptotic* growth, not just one being larger or smaller. For instance, $10n + 5 \in \Theta(n)$, but $10n + 5 \notin \Theta(n^2)$ since linear growth does not keep pace with quadratic growth.

Most of the time, when people say "Big-O" they actually mean Theta, i.e., a Big-O upper bound is meant to be the tight one, because we rarely say upper bounds that overestimate the rate of growth.

2.7 $\mathbf{O(1)}, \Omega(1),$ and $\Theta(1)$

We have discussed Big-O notation for functions like $O(n)$ or $O(n^2)$, where the functions in parentheses grow without bound. However, not all functions increase with larger inputs; some are *bounded* and never exceed a fixed constant.

Consider $f(n) = 1$, a constant function that always outputs 1. To say g is in $O(f)$ means there exist constants $c > 0$ and n_0 such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$. Since $f(n) = 1$, this is the same as $g(n) \leq c$. In other words, beyond some point n_0, the values of $g(n)$ stay at or below c. We call such functions *asymptotically bounded* and write $g \in O(1)$.

$\Omega(1)$. By the same reasoning, we say $g \in \Omega(1)$ if there exist $c > 0$ and n_0 so that $g(n) \geq c$ for all $n \geq n_0$. That is, g remains above some positive constant in the long run. While it may seem trivial, not every function meets this condition. For instance, $g(n) = \frac{1}{n+1}$ is certainly in $O(1)$ (since it is bounded above), but it is *not* in $\Omega(1)$ (because it approaches zero as n increases).

$\Theta(1)$. Finally, if g is both $O(1)$ and $\Omega(1)$, we say $g \in \Theta(1)$. These functions stay within constant bounds *above* and *below*. For example, $g(n) = n^2$ is in $\Omega(1)$ (clearly it grows above a constant) but not in $O(1)$ (it is unbounded), so $n^2 \notin \Theta(1)$. Meanwhile, a function that stays near a nonzero constant—say, $g(n) = 5$—will be both $O(1)$ and $\Omega(1)$, so $g(n) = 5 \in \Theta(1)$.

2.8 Properties of Big-O, Omega, and Theta

Proving relationships like $g(n) \in O(f(n))$ directly via chains of inequalities can become tedious. Instead, we can use a few powerful properties of Big-O, Ω, and Θ. These properties allow us to combine or compare functions more easily and save time in the long run. We will highlight one sample proof; the rest often follow by similar reasoning, or else they can be shown using calculus methods.

2.8.1 Elementary functions

In the next theorem, we compare four broad classes of what we will call "elementary" functions: *constant functions*, *logarithms*, *powers of n*, and *exponential functions*.

Let $a, b \in \mathbb{R}^{>0}$. Then the following statements hold:

1. If $a > 1$ and $b > 1$, then $\log_a(n) \in \Theta\big(\log_b(n)\big)$.
2. If $a < b$, then $n^a \in O(n^b)$ and $n^a \notin \Omega(n^b)$.
3. If $a < b$, then $a^n \in O(b^n)$ and $a^n \notin \Omega(b^n)$.
4. If $a > 1$, then $1 \in O\big(\log_a(n)\big)$ and $1 \notin \Omega\big(\log_a(n)\big)$.
5. $\log_a(n) \in O\big(n^b\big)$ but $\log_a(n) \notin \Omega\big(n^b\big)$.
6. If $b > 1$, then $n^a \in O\big(b^n\big)$ and $n^a \notin \Omega\big(b^n\big)$.

Growth illustration. A rough sketch of these classes in order of increasing growth might look like this:

$$\text{(constant: 1)}, \quad \log_2(n),\, \log_3(n), \ldots, \log_{100}(n), \quad n^{0.5},\; n,\; n^2,\; n^{10^7}, \quad 2^n,\; 100^n, \ldots$$

The diagram below (in a typical text or on the board) shows how each function eventually outgrows the preceding one:

Hierarchy: constants < logarithms < polynomials < exponentials

These comparisons illustrate why, for example, any polynomial n^a is eventually outpaced by any exponential b^n when $b > 1$, or why different logarithms differ only by a constant factor, etc. Having these facts in hand makes Big-O, Ω, and Θ proofs substantially more straightforward.

2.9 Worst-Case Analysis

In earlier sections on *analyzing algorithm running time*, we explored asymptotic notation to describe how the number of "basic operations" grows with input size. This lets us set aside low-level details like the computing environment, or the exact way primitive operations are implemented, and instead focus on the *relationship* between input size and the total operations performed.

However, basing our analysis *solely* on input size can be too restrictive. While we can define "input size" for each algorithm (e.g., the length of a list), in reality an algorithm's running time can also depend on `the values` in the input. For instance, consider this function, `has_even`, which checks whether a list of integers contains an even number:

```
def has_even(numbers: list[int]) -> bool:
    """Return whether 'numbers' contains an even integer."""
    for number in numbers:
        if number % 2 == 0:
            return True
    return False
```

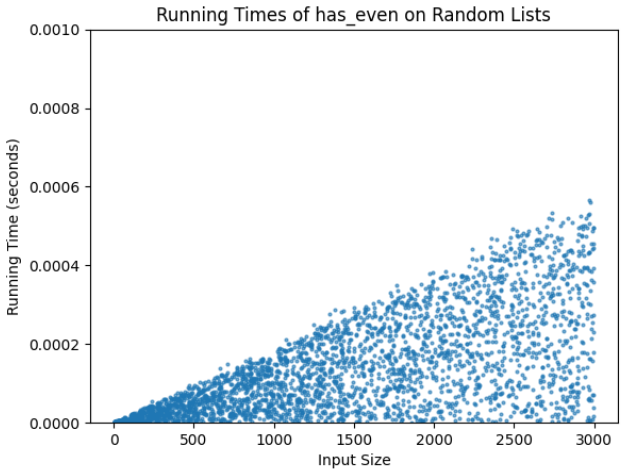
Since `has_even` stops immediately upon finding an even number, its running time might not be proportional to the entire list length. Even if two lists share the same length, if one has an even integer near the front, the function will return very quickly. Hence, for inputs of size n, the runtime can vary significantly, depending on the distribution of even and odd numbers.

To illustrate, we ran a timing experiment using `timeit` on randomly generated lists of different sizes, and we plotted the measured running times. Although all timing tests involve some randomness and measurement noise, the spread of values is more than what could be explained by input size alone. Asymptotic notation describes the growth of an entire function as the input size grows, but it does not capture the full range of possible running times for a fixed n. To handle this discrepancy, we often focus on the `maximum` running time at each size (i.e. the worst case), which yields a consistent function describing the slowest possible performance for any given n.

Below is an example of Python code that uses `timeit` to measure the runtime of `has_even` across a range of list sizes, then plots the results using `matplotlib`:

```
import timeit
import random
import matplotlib.pyplot as plt

def has_even(numbers):
    for number in numbers:
        if number % 2 == 0:
            return True
    return False

# Generate random input sizes
sizes = [random.randint(1,3000) for _ in range(3000)]


times = []
for s in sizes:
    # Create a list of length s with random integers
    arr = [1] * (s - 1)
    # Append one even number, for instance 2
    arr.append(2)
    # Optionally shuffle the list so the even number isn't always at the end
    random.shuffle(arr)
    # Measure the time (1 execution) for has_even on arr
    t = timeit.timeit(lambda: has_even(arr), number=10)
    times.append(t)
```

```
26
27  # Plot the results
28  plt.scatter(sizes, times, s=4, alpha=0.6)
29  plt.xlabel("Input Size")
30  plt.ylabel("Running Time (seconds)")
31  plt.ylim(0, 0.001)
32  plt.title("Running Times of has_even on Random Lists")
33  plt.show()
```



In the resulting scatter plot, we observe that runtimes tend to increase with list size, but there is considerable variation at each size due to differing contents of the lists. This demonstrates why we often shift to *worst-case* or *average-case* analyses in theoretical studies, where we either pick an input arrangement that forces maximal work (worst case) or compute an expected value over all inputs (average case).

Whereas asymptotic analysis studies the relationship between input size and running time, worst-case analysis studies only the relationship between input size `and the maximum possible` running time. In other words, rather than answering the question "what is the running time of this algorithm for an input size n?" we instead ask "what is the *maximum possible* running time of this algorithm for an input size n?"

We typically use the name $T(n)$ to represent the *maximum* possible running time as a function of n, the input size. A result of our analysis might be something like $T(n) = \Theta(n)$, meaning that the worst-case running time of our algorithm grows linearly with the input size.

It is difficult to compute the *exact maximum* number of basic operations performed by this algorithm for every input size, which requires that we identify an input for each input size, count its maximum number of basic operations, and then prove that every input of this size takes at most this number of operations. Instead, we will generally take a two-pronged approach: proving matching *upper* and *lower bounds* on the worst-case running time of our algorithm.

2.9.1 Upper bounds on the worst-case runtime

Definition 7. *$T(n)$ is the worst-case runtime function of a given algorithm, we say that a function $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ is an upper bound on the worst-case runtime when $T(n) \in O(f)$.*

To get some intuition about what an upper bound on the worst-case running means, suppose we use absolute dominance rather than Big-O. In this case, there's a very intuitive way to expand the phrase "$T_{\texttt{func}}$ is absolutely dominated by f":

$$\begin{aligned}
&\forall n \in \mathbb{N}, T(n) \leq f(n) \\
&\iff \forall n \in \mathbb{N}, \max\{\text{running time of executing } \texttt{func}(x) \mid x \in \mathcal{I}_{\texttt{func},n}\} \leq f(n) \\
&\iff \forall n \in \mathbb{N}, \forall x \in \mathcal{I}_{\texttt{func},n}, \text{ running time of executing } \texttt{func}(x) \leq f(n)
\end{aligned}$$

The last line comes from the fact that if we know the maximum of a set of numbers is less than some value K, then *all* numbers in that set must be less than K. Thus an upper bound on the worst-case runtime is equivalent to an upper bound on the runtimes of *all* inputs.

Now when we apply the definition of Big-O instead of absolute dominance, we get the following translation of $T(n) \in O(f)$:

$$\exists c, n_0 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_0 \Rightarrow (\forall x \in \mathcal{I}_{\texttt{func},n}, \text{ running time of executing } \texttt{func}(x) \leq c \cdot f(n))$$

To approach an analysis of an upper bound on the worst-case, we typically find a function g such that $T(n)$ is absolutely dominated by g, and then find a simple function f such that $g \in O(f)$. But how do we find such a g? And what does it mean to upper bound *all* runtimes of a given input size? We'll illustrate the technique in our next example.

Example 5. *Find an asymptotic upper bound on the worst-case running time of* `has_even`*.*

First, let $n \in \mathbb{N}$ and let `numbers` be an arbitrary list of length n.

Now we'll analyze the running time of `has_even`, except we *can't* assume anything about the values inside `numbers`, because it's an arbitrary list. But we can still find an upper bound on the running time:

- The loop (`for number in numbers`) iterates *at most* n times. Each loop iteration counts as a single step (because it is constant time), so the loop takes *at most* $n \cdot 1 = n$ steps in total.
- The `return False` statement (if it is executed) counts as 1 basic operation.

Therefore the running time is *at most* $n + 1$, and $n + 1 \in O(n)$. So we can conclude that the worst-case running time of `has_even` is $O(n)$.

Note that we did *not* prove that `has_even(numbers)` takes exactly $n + 1$ basic operations for an arbitrary input `numbers` (this is false); **we only proved an upper bound on the number of operations**. And in fact, we don't even care that much about the exact number: what we ultimately care about is the asymptotic growth rate, which is linear in $n + 1$. This allowed us to conclude that the worst-case running time of `has_even` is $O(n)$.

But because we calculated an upper bound rather than an exact number of steps, we can only conclude a Big-O, not Theta bound: we don't yet know that this upper bound is tight.

2.9.2 Lower bounds on the worst-case runtime

So how do we prove our upper bound is tight? Since we've just shown that $T_{\texttt{has\_even}}(n) \in O(n)$, we need to prove the corresponding lower bound $T_{\texttt{has\_even}}(n) \in \Omega(n)$. But what does it mean to prove a lower bound on the maximum of a set of numbers? Suppose we have a set of numbers S, and say that "the maximum of S is at least 50." This doesn't tell us what the maximum of S actually is, but it does give us one piece of information: there has to be a number in S which is at least 50.

The key insight is that the converse is also true—if I tell you that S contains the number 50, then you can conclude that the maximum of S is at least 50.

$$\max(S) \geq 50 \Leftrightarrow (\exists x \in S, x \geq 50)$$

Using this idea, we'll give a formal definition for a lower bound on the worst-case runtime of an algorithm.

Definition 8. *Let* `func` *be a program, and $T_{\texttt{func}}$ its worst-case runtime function. We say that a function $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ is a lower bound on the worst-case runtime when $T_{\texttt{func}} \in \Omega(f)$.*

In an analogous fashion to the upper bound, we unpack this definition first by using absolute dominance:

$$\begin{aligned}
&\forall n \in \mathbb{N}, T_{\texttt{func}}(n) \geq f(n) \\
&\iff \forall n \in \mathbb{N}, \max\{\text{running time of executing } \texttt{func}(x) \mid x \in \mathcal{I}_{\texttt{func},n}\} \geq f(n) \\
&\iff \forall n \in \mathbb{N}, \exists x \in \mathcal{I}_{\texttt{func},n}, \text{ running time of executing } \texttt{func}(x) \geq f(n)
\end{aligned}$$

And then using Omega:

$$\exists c, n_0 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_0 \Rightarrow (\exists x \in \mathcal{I}_{\texttt{func},n}, \text{ running time of executing } \texttt{func}(x) \geq c \cdot f(n))$$

Remarkably, the crucial difference between this definition and the one for upper bounds is a change of quantifier: now the input x is existentially quantified, meaning we get to pick it. Or really, our goal is to find an *input family*—a set of inputs, one per input size n—whose runtime is asymptotically larger than our target lower bound.

Example 6. *Find an asymptotic lower bound on the worst-case running time of* `has_even`*.*

Running time analysis. (Lower bound on worst-case)

Let $n \in \mathbb{N}$. Let `numbers` be the list of length n consisting of all 1s. Now we'll analyze the (exact) running time of `has_even` on this input.

In this case, the `if` condition in the loop is always False, so the loop never stops early. Therefore it iterates exactly n times (once per item in the list), with each iteration taking one step.

Finally, the `return False` statement executes, which is one step. So the total number of steps for this input is $n + 1$, which is $\Omega(n)$.

2.9.3 Early returning in Python built-ins

We have encountered several Python functions and methods whose running time depends on more than just the size of their inputs. One such example is the list search operation using the `in` keyword:

```
lst = list(range(0, 1000000))
timeit.timeit(lambda: 0 in lst, number=10, globals=globals())
>>>2.833000000007635e-06
timeit.timeit(lambda: -1 in lst, number=10, globals=globals())
>>>0.059140791
```

In the first `timeit` expression, `0` appears as the first element of `lst`, so it is found immediately. In the second case, `-1` does not appear in `lst` at all, so all one million elements of `lst` must be checked. This results in a running time proportional to the length of the list. The worst-case running time of the `in` operation for lists is $\mathcal{O}(n)$, where n is the length of the list.

We have also seen two more functions, `any` and `all`, which are implemented using an early return:

- `any` searches for a single `True` in a collection and stops as soon as it finds one.
- `all` requires all elements of a collection to be `True` and stops as soon as it finds a `False` value.

Here are some examples:

```
import timeit
all_trues = [True] * 1000000
all_falses = [False] * 1000000
print(timeit.timeit('any(all_trues)', number=10, globals=globals()))
>>>9.579999999953515e-07

print(timeit.timeit('any(all_falses)', number=10, globals=globals()))
>>>0.03838899999999998

print(timeit.timeit('all(all_trues)', number=10, globals=globals()))
>>>0.03972500000000001

print(timeit.timeit('all(all_falses)', number=10, globals=globals()))
>>>1.624999999977339e-06
```

From the examples above:

- `any(all_trues)` returns `True` immediately after checking the first list element.
- `any(all_falses)` returns `False` only after checking all one million elements.
- `all(all_trues)` returns `True` only after checking all one million elements.
- `all(all_falses)` returns `False` immediately after checking the first list element.

Thus, `any` and `all` have a worst-case running time of $\mathcal{O}(n)$, where n is the size of the input collection. However, in practice, they can be much faster if the "right" boolean value is encountered early.

Using `any` and `all` with Comprehensions There is a subtlety when using `any` or `all` with comprehensions. For example:

```
import timeit
print(timeit.timeit(lambda: any([x == 0 for x in range(0, 1000000)]), number=10))
>>>0.23009787500000004
```

This is much slower than expected, considering the first element checked is `x = 0`. The issue is that the entire comprehension is evaluated before `any` is called. As discussed in the chapter on analyzing comprehensions and while loops, the running time of evaluating a comprehension is proportional to the size of its source collection. In this example, the source collection is `range(0, 1000000)`, which contains one million numbers.

A better approach is to use **generator comprehensions**, which do not evaluate all elements at once but instead produce elements as needed:

```
print(any(x == 0 for x in range(0, 1000000)))
>>> True
```

This achieves the fast running time we expected:

```
>>> print(timeit.timeit(lambda: any(x == 0 for x in range(0, 1000000)), number=10))
4.833000000009635e-06
```

Here, only `x = 0` from the generator comprehension is evaluated. None of the other possible values (`x = 1, 2, ..., 999999`) are checked by the `any` call. Generator comprehensions thus enable efficient use of `any` and `all`.

2.9.4 Don't assume bounds are tight!

Our intuition here pulls us towards the bounds being *"obviously"* the same, but this is really a side effect of the examples we have studied so far in this section being rather straightforward. However, this won't always be the case: the study of more complex algorithms and data structures exhibits numerous situations where obtaining an upper bound on the running time involves a completely different analysis than the lower bound.

Let us consider one such example that deals with manipulating strings. A string is called a **palindrome** if it can be read the same forwards and backwards. Examples of palindromes include `"abba"`, `"racecar"`, and `"z"`. A string s_1 is called a **prefix** of another string s_2 if s_1 is a substring of s_2 starting at index 0 of s_2. For example, the string `"abc"` is a prefix of `"abcdef"`.

The following algorithm takes a non-empty string as input and returns the length of the longest prefix of that string that is a palindrome. For instance, given the string `"attack"`, the non-empty prefixes that are palindromes are `"a"` and `"atta"`, so the algorithm will return 4.

```
def palindrome_prefix(s: str) -> int:
    n = len(s)
    for prefix_length in range(n, 0, -1):  # Iterate from n down to 1
        # Check whether s[0:prefix_length] is a palindrome
        is_palindrome = all(s[i] == s[prefix_length - 1 - i]
                            for i in range(0, prefix_length))
        # If a palindrome prefix is found, return its length
        if is_palindrome:
            return prefix_length
```

Here are a few notable details about the algorithm:

- The `for` loop uses `range(n, 0, -1)`, where the third argument `-1` causes the loop variable to start at n and decrease by 1 at each iteration. In other words, the loop checks prefixes starting from the longest prefix (length n) and working down to the shortest prefix (length 1).
- The call to `all` checks pairs of characters starting at either end of the current prefix. It employs a generator comprehension so that it can terminate early as soon as a mismatch is found (i.e., when `s[i] != s[prefix_length - 1 - i]`).
- Even though the only `return` statement is inside the `for` loop, the algorithm is guaranteed to find a palindrome prefix because the first letter of the string s by itself is always a palindrome.

The code presented is structurally simple. Indeed, it is not too difficult to show that the worst-case runtime of this function is $O(n^2)$, where n is the length of the input string. However, demonstrating that the worst-case runtime is $\Omega(n^2)$ requires identifying an input family whose runtime achieves $\Theta(n^2)$. To do so, we must find input cases that avoid two points in the code that could result in fewer-than-maximum loop iterations.

The difficulty lies in the fact that these two points arise from different types of inputs. The `all` call can terminate early as soon as the algorithm detects that a prefix is not a palindrome, while the `return` statement is executed only when the algorithm determines that a prefix is indeed a palindrome. To illustrate this tension more clearly, let us consider two extreme input families that might seem plausible at first glance but do not result in a runtime of $\Theta(n^2)$.

Suppose the entire string s is a palindrome of length n. In this case:

- During the first iteration of the loop, the entire string is checked.
- The `all` call examines all pairs of characters and determines that `is_palindrome = True`.
- As a result, the loop exits during its very first iteration, and the algorithm returns immediately.

Since the `all` call takes n steps in this case, the runtime for this input family is $\mathcal{O}(n)$.

Now suppose the entire string s consists of n distinct letters. In this case:

- The only palindrome prefix is the first letter of s itself.
- Consequently, the loop runs for all n iterations, only returning in the final iteration (when `prefix_length = 1`).
- However, the `all` call always terminates after a single step, as it compares the first letter of s with another letter, which is guaranteed to differ by construction.

Thus, this input family also results in a runtime of $\mathcal{O}(n)$.

The key idea is to choose an input family that:

- Does not contain a long palindrome (so the loop runs for many iterations).
- Has prefixes that are *almost* palindromes (so the `all` call checks many pairs of letters).

Let $n \in \mathbb{Z}^+$. We define the input string s_n as follows:

- $s_n[\lfloor n/2 \rfloor] = b$
- Every other character in s_n is equal to a.

For example:

$$s_4 = \texttt{aaba}, \quad s_{11} = \texttt{aaaaaabaaaa}.$$

Note that s_n is very close to being a palindrome. If the single character b were replaced with a, the resulting string would be a palindrome (the all-a string). However:

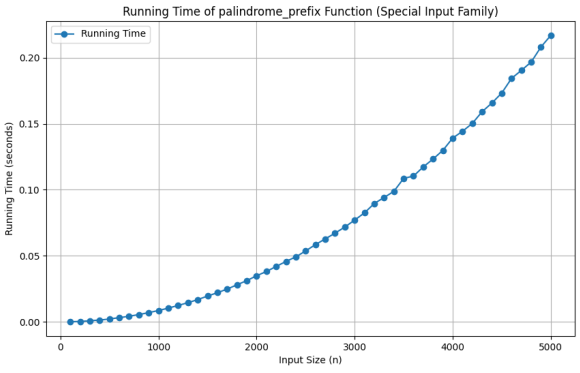
- By setting the center character to b, we ensure that the longest palindrome in s_n has length approximately $n/2$, causing the loop to iterate roughly $n/2$ times.
- For prefixes of s_n longer than $n/2$, the outer characters of each prefix are all the same, which causes the `all` call to compare many pairs before detecting the mismatch between a and b.

It turns out that this input family does indeed have an $\Omega(n^2)$ runtime!

```python
import time
import matplotlib.pyplot as plt

def palindrome_prefix(s: str) -> int:
    n = len(s)
    for prefix_length in range(n, 0, -1):  # Iterate from n down to 1
        # Check whether s[0:prefix_length] is a palindrome
        is_palindrome = all(s[i] == s[prefix_length - 1 - i]
                            for i in range(0, prefix_length))
        # If a palindrome prefix is found, return its length
        if is_palindrome:
            return prefix_length
    return 0  # No palindrome prefix found

# Function to generate the special input family s_n
def generate_special_input(n: int) -> str:
    mid = n // 2
```

```
    s = ["a"] * n
    s[mid] = "b"
    return "".join(s)

# Function to measure running time
def measure_running_time(func, s):
    start_time = time.time()
    func(s)
    end_time = time.time()
    return end_time - start_time

# Generate test cases
input_sizes = range(100, 5001, 100)  # Input sizes from 100 to 5000, step 100
special_inputs = [generate_special_input(n) for n in input_sizes]  # Special input family s_n

# Measure running time for each input size
running_times = []
for s in special_inputs:
    running_time = measure_running_time(palindrome_prefix, s)
    running_times.append(running_time)

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(input_sizes, running_times, marker='o', label="Running Time")
plt.xlabel("Input Size (n)")
plt.ylabel("Running Time (seconds)")
plt.title("Running Time of palindrome_prefix Function (Special Input Family)")
plt.grid(True)

plt.legend()
plt.show()
```

The running time figure is:



2.10 Average-Case Analysis

In your studies as computer scientists, you have primarily focused on worst-case algorithm analysis. However, in practice, worst-case analysis can often be misleading, as many algorithms and data structures with poor worst-case performance still perform well on the majority of inputs.

This discrepancy is not surprising: focusing solely on the maximum of a set of numbers (e.g., running times) tells us little about the *typical* values or the distribution of values in that set. In this section, we introduce a powerful technique for analyzing the *typical* running time of an algorithm.

Consider the following algorithm, which operates on a non-empty array of integers:

```
def evens_are_bad(lst):
    # Check if every number in the list is even
    if all(x % 2 == 0 for x in lst):
        # Repeat len(lst) times
```

```
5           for _ in range(len(lst)):
6               # Calculate and print the sum of the list
7               print(sum(lst))
8           return 1
9       else:
10          return 0
```

Let n represent the length of the input list `lst`. Suppose `lst` contains only even numbers. The initial check on line `if` takes $\mathcal{O}(n)$ time, while the computation in the `if` branch takes $\mathcal{O}(n^2)$ time. Thus, the worst-case running time of this algorithm is $\Theta(n^2)$.

However, the loop executes only when every number in `lst` is even. If just one number is odd, the running time is $\mathcal{O}(n)$, the maximum time required to check for all-even numbers. Intuitively, it seems much more likely that not every number in `lst` is even, so the more `typical case` for this algorithm has a running time bounded above by $\mathcal{O}(n)$, with the $\Theta(n^2)$ case occurring only rarely.

To rigorously define the `typical case`, we use tools from probability theory. Specifically, we define the **average-case running time** of an algorithm as the function $T_{\text{avg}}(n)$, which takes an input size n and returns the (weighted) average of the algorithm's running time for all inputs of size n. Let us compute the average-case running time of `evens_are_bad`. First, we fix the input size n and define the set of inputs as lists whose elements are between 1 and 5, inclusive. This **simplifies** our calculations.

For this analysis, we count the number of times a list element is accessed, either during the even-checking operation or while computing the sum. Each step corresponds to one list access.

To simplify calculations, assume the even-checking operation on line `if` always accesses all n elements. The loop executes n^2 steps (each number is accessed n times). There are two cases:

- Lists with all even numbers: 2^n such lists exist, each taking $n^2 + n$ steps.
- All other lists: $5^n - 2^n$ such lists exist, each taking n steps.

The average running time is:

$$T_{\text{avg}}(n) = \frac{2^n(n^2 + n) + (5^n - 2^n)n}{5^n}$$

Simplify the expression:

$$T_{\text{avg}}(n) = \frac{2^n n^2}{5^n} + n$$

As $n \to \infty$, the term $\frac{2^n n^2}{5^n} \to 0$ because exponential growth dominates polynomial growth. Thus:

$$T_{\text{avg}}(n) = \Theta(n)$$

3 List

In computer science, data types are categorized into **abstract data types (ADTs)** and **concrete data types**. These concepts are essential for understanding how data is organized and manipulated in programming.

An `abstract data type (ADT)` is a theoretical model of an entity, along with the set of operations that can be performed on that entity. The key term here is *abstract*: an ADT is a definition that can be understood and communicated without any code at all.

A `data structure`, on the other hand, is a value in a program that can be used to store and operate on data. To illustrate the distinction, consider the difference between the `List ADT` and an array data structure.

3.1 List ADT and List Data Structure

The List ADT supports the following operations:

- **Length(L)**: Return the number of items in L.
- **Get(L, i)**: Return the item stored at index i in L.
- **Store(L, i, x)**: Store the item x at index i in L.

This definition of the List ADT is intentionally minimal, specifying only the essential operations for lists. Notice that this description is entirely abstract: it defines the behavior of the data type without specifying how the data is stored or how these operations are implemented.

It is tempting to think of ADTs as the definition of interfaces or abstract classes in programming languages—entities that specify a collection of methods to be implemented (what data and what operations). A data structure, by contrast, is inherently tied to its implementation in code(how the data is stored and how the operations are implemented). It exists as a tangible entity in a program, and when discussing data structures, we focus not only on what they do but also on `how` they do it.

For example, `arrays` can be used to implement the List ADT. Arrays support the operations defined in the List ADT, but they also have specific implementation details:

- Arrays store elements in consecutive memory locations.
- They perform **Get** and **Store** operations in constant time, independent of the length of the array.
- The implementation of **Length** depends on the programming language. In Python, native lists (implemented using arrays) include a member to store their length, at least in the standard CPython implementation.

The key implementation-level detail that is important in this context is the `running time` of an operation. While running time is not part of an ADT's definition, it becomes significant when analyzing how an operation is implemented in a particular data structure.

3.2 What is array?

An `array` is a data structure that stores a collection of elements in contiguous memory locations. Each element in the array is identified by an **index** or a **key**. Arrays are efficient for:

- **Random Access**: Accessing an element by its index is $O(1)$ due to the contiguous memory layout.
- **Iteration**: Traversing through all elements is straightforward.

There are two types of arrays; the `static array` is:

- **Definition**: A static array has a fixed size determined at the time of its creation. Its size cannot be changed during runtime.
- **Advantages**:Simple to use and memory layout is predictable.
- **Disadvantages**:Wasteful if the size is over-allocated and unused and inflexible as size cannot change once allocated.

The `dynamic array` is:

- **Definition**: A dynamic array can grow or shrink in size during runtime. Memory is allocated on the heap and can be resized using specific operations.
- **Advantages**:Flexible as the size can be adjusted as needed, and efficient use of memory.
- **Disadvantages**: Managing memory manually can lead to errors (e.g., memory leaks), and reallocation can involve copying the entire array to a new memory location, which can be costly.

3.2.1 What Kind of Array is Used to Implement Python Lists?

Python lists are implemented using **dynamic arrays** in CPython. The internal structure of a Python list (as described in the CPython source code) is similar to a dynamic array:

- **Growth**: When the list needs to grow, the internal array is resized by allocating a larger array (with over-allocation to minimize frequent resizing) and copying the old elements to the new array.
- **Efficiency**: Although resizing can be expensive ($O(n)$), the **amortized time complexity** for appending elements is $O(1)$, thanks to geometric over-allocation.

3.2.2 Is an Array Primitive Like Pointers?

- In low-level languages like C, arrays are not "primitive" in the strict sense. They are **data structures** built on top of pointers:
  - The name of an array is essentially a pointer to its first element.
  - You can access elements using pointer arithmetic operations.
  - Arrays own storage; pointers only reference storage.

```c
#Example in C
int arr[5] = {1, 2, 3, 4, 5};
printf("%d", *(arr + 2)); // Access the third element (index 2)
```

- In higher-level languages like Python:
  - Arrays (or list-like structures) are **abstracted** and are not directly tied to pointers. Instead, these structures are implemented internally (e.g., Python lists using dynamic arrays).

The Python list implementation is part of the CPython source code, written in C for performance. The relevant implementation is found in the file `listobject.c` in the CPython repository.

3.2.3 Key Structures in `listobject.c`

- **List Object Structure**: The list object is implemented using a dynamic array structure in C.

```c
typedef struct {
    PyObject_VAR_HEAD
    PyObject **ob_item;  /* Array of pointers to PyObject */
    Py_ssize_t allocated;  /* Allocated size of ob_item array */
} PyListObject;
```

  - `PyObject_VAR_HEAD`: Includes size information for variable-sized objects.
  - `ob_item`: Pointer to an array of `PyObject` pointers (the elements of the list).
  - `allocated`: Total number of allocated slots in `ob_item` (may be larger than the current size for efficiency).
- **Dynamic Resizing**: Python lists grow dynamically. When more space is needed, the list grows by over-allocating to minimize future reallocations. The growth strategy is geometric, typically increasing the capacity by 1.125x. The full implementation can be found in the CPython GitHub repository under the function `list_resize()`.

3.2.4 Think More About Dynamic Arrays

A dynamic array maintains the property of storing items in a contiguous memory block while allowing for expansion when the array is full. Expanding the array isn't as simple as adding memory at its endpoint, as the adjacent memory might already be in use. Instead, the solution is to allocate a new, larger memory block, copy the existing elements into it, and then add the new element. In the implementation below, the array data structure has three attributes: `array`, a reference to the actual data; `allocated`, the total allocated spots; and `size`, the number of filled spots. Here, an expansion factor of `two` is used, meaning the allocated memory doubles in size with each expansion.

```python
def insert(A, x):
    # Check whether the current array is full
    if A.size == A.allocated:
        new_array = [None] * (A.allocated * 2)  # Create a new array with double the size
```

```
        for i in range(A.size):
            new_array[i] = A.array[i]  # Copy all elements into the new array
        A.array = new_array
        A.allocated *= 2  # Update the allocated size

    # Insert the new element into the first empty spot
    A.array[A.size] = x
    A.size += 1
```

3.3 Amortized analysis

Before, when we discussed worst-case analysis and average-case analysis, we assumed the "basic operation" was the **same** for each element in a given input, but this assumption does not hold for the above insert function! How do we measure complexity, or say efficiency of the above insert function?

Before defining amortized analysis formally, let us clarify a key point. In dynamic arrays, insertion is rarely slow. It only becomes inefficient when the array is full (i.e., the number of items is a power of two), triggering a resizing operation. Most insertions are fast, and reaching the "slow" state requires a series of efficient insertions beforehand.

For example, consider inserting into a full array of size 1024 (a power of two). Resizing will require 2049 array accesses: copying 1024 elements and inserting one new element. However, to reach this state:

- The array was previously resized from 512 to 1024.
- This required 511 fast insertions, each taking only one array access.

Thus, the expensive operation is preceded by many efficient ones, making the overall process less problematic. *Amortized analysis* **computes the average cost per operation over a sequence of operations, starting from an initial state.** Unlike worst-case or average-case analysis, it focuses on the total cost of multiple operations rather than a single operation in isolation. **Amortized analysis is particularly suitable for algorithms where the worst-case cost is high but occurs periodically, rather than every time the operation is performed.** This periodic nature allows the high cost to be distributed over multiple operations, leading to a lower average cost per operation over time.

There are two ways to think about amortized cost:

1. The **average cost per operation** when multiple operations are performed consecutively. This differs from average-case analysis, which considers operations as isolated events.
2. The **total cost of M operations** as a function of M. For example, M constant-time operations will have a total runtime of $O(M)$. Here, "linear" growth refers to the total runtime of the sequence, not the worst-case time of a single operation.

3.3.1 Aggregate Method

The `aggregate method` is a straightforward approach for amortized analysis. It computes the total cost of a sequence of operations and divides this total by the number of operations to determine the average cost per operation.

Consider a dynamic array starting at size 0 with an allocated space of 1. Suppose we perform a sequence of M insertion operations. The total cost of these insertions is determined as follows:

- For most insertions, the cost is 1 (a single array access).
- When the array size is a power of two, resizing occurs, and the cost becomes $2k+1$, where k is the size of the array before resizing.

The total cost of the M operations is expressed as:

$$\sum_{k=0}^{M-1} T(k),$$

where $T(k)$ is 1 for non-power-of-two insertions and $2k+1$ for power-of-two insertions:

$$
\begin{aligned}
\sum_{k=0}^{M-1} T(k) &= M + \sum_{k=0}^{M-1} T'(k) \\
&= M + \sum_{i=0}^{\lfloor \log(M-1) \rfloor} T'(2^i) && \text{(substituting } k = 2^i\text{)} \\
&= M + \sum_{i=0}^{\lfloor \log(M-1) \rfloor} 2 \cdot 2^i \\
&= M + 2 \cdot \left(2^{\lfloor \log(M-1) \rfloor + 1} - 1\right) && (\sum_{i=0}^{d} 2^i = 2^{d+1} - 1) \\
&= M - 2 + 4 \cdot 2^{\lfloor \log(M-1) \rfloor}.
\end{aligned}
$$

We can further deduce:

$$
\begin{aligned}
\log(M-1) - 1 \leq \lfloor \log(M-1) \rfloor &\leq \log(M-1), \\
2^{\log(M-1)-1} \leq 2^{\lfloor \log(M-1) \rfloor} &\leq 2^{\log(M-1)}, \\
\frac{1}{2}(M-1) \leq 2^{\lfloor \log(M-1) \rfloor} &\leq M-1, \\
2(M-1) + (M-2) \leq \sum_{k=0}^{M-1} T(k) &\leq 4(M-1) + (M-2), \\
3M - 4 \leq \sum_{k=0}^{M-1} T(k) &\leq 5M - 6.
\end{aligned}
$$

The total cost of dynamic array insertion lies between $3M$ and $5M$. While this may seem linear, it is important to note that the key variable here is the number of operations, M, rather than the size of the data structure. In amortized analysis, the size of the data structure is implicitly tied to the number of operations performed. The average cost per operation is between 3 and 5, which is constant. Therefore, we conclude that the amortized cost of dynamic array insertion is $O(1)$. This formalizes the intuition that the rare inefficient insertions are balanced out by the frequent efficient ones.

3.3.2 Accounting (Banker's) Method

The **accounting method**, also known as the **banker's method**, is another approach to amortized analysis. Instead of directly calculating the total cost of M operations, we associate a fixed **charge** with each operation. The total charges must be greater than or equal to the total cost of the operations. This ensures that expensive operations are covered by surplus charges from previous operations.

The relationship between charges and costs can be likened to a payment system:

- **Cost**: The actual expense of an operation (e.g., array accesses during resizing).
- **Charge**: The amount we "pay" for each operation. It can be more than the cost, and the surplus is saved as **credits** to pay for future operations.

The charge for each operation is chosen as a constant c. If the total cost $T(M)$ of M operations satisfies $T(M) \leq cM$, then:

$$
\frac{T(M)}{M} \leq c,
$$

implying that the average (amortized) cost per operation is at most c.

To analyze dynamic array insertion, we use the following charging scheme:

- Assign a charge of 5 credits to each insertion operation.
- For each insertion at index i: (1) 1 credit is used to pay for the insertion (1 array access). (2) The remaining 4 credits are stored for future resizing operations.

When the array is resized (at indices 2^k), the stored credits cover the cost of copying elements. We can have a claim below:

Let $1 \leq i \leq M$. The number of credits removed for insertion i equals the number of array accesses needed for that insertion.

Case 1: i is not a power of two.

- Only 1 array access occurs.
- This is paid by removing 1 credit from the 5 credits assigned to index i.

Case 2: i is a power of two.

- For insertion at index $i = 2^k$, there are $2i + 1$ array accesses.
- 1 credit is removed for the insertion itself, leaving $2i$ accesses to be paid.
- There are four credits removed for each index j such that i is the smallest power of two greater than or equal to j. Let $i = 2^k$. The previous power of two is $2^{k-1} = i/2$. The possible values for j are:

$$\frac{i}{2} + 1, \frac{i}{2} + 2, \ldots, i,$$

  which gives $i/2$ choices. Thus, there are $i/2 \cdot 4 = 2i$ credits removed, accounting for the remaining array accesses.

Each insertion is charged 5 credits, which is sufficient to cover the actual cost, including resizing. Thus, the amortized cost of dynamic array insertion is:

$$O(1).$$

The accounting method allows us to assign charges such that the total charge is always sufficient to cover the total cost. This ensures a constant amortized cost for insertion, even though some operations (e.g., resizing) may have a much higher actual cost. Creative charging schemes balance surplus charges and costs, providing flexibility in amortized analysis.

Now, let's relax. Think about the time complexity for list functions `reverse()`, `max()`, `sum()` and `indexing` operations. Can you figure it out quickly?

3.4 Summary of Analysis Methods

The choice between `worst-case analysis`, `average-case analysis`, and `amortized analysis` depends on the context of the problem, the nature of the algorithm or data structure, and the goals of the analysis.

Analysis Type	Focus	When to Use
Worst-Case	Maximum time/resources for any input of size n.	Real-time systems, safety-critical applications, adversarial inputs.
Average-Case	Expected time/resources for a random input of size n.	Probabilistic inputs, practical performance, randomized algorithms.
Amortized	Average time/resources per operation over a sequence of operations.	Dynamic data structures, sequences of operations, overall efficiency.

- Use **worst-case analysis** for guarantees in critical systems.
- Use **average-case analysis** for known probabilistic or random inputs.
- Use **amortized analysis** for sequences of operations, especially in dynamic data structures.

Each type of analysis has its place, and the choice depends on the problem context and the guarantees or insights you need.

3.5 Binary Search

Before getting to our sorting algorithms, we'll take a quick stop at one topic that both motivates why sorting a list can be useful. Recall the standard way of searching for an item in a list, using a for loop with an early return:

```
1 def search(lst: list, item: Any) -> bool:
2     """Return whether item appears in this list."""
3     for element in lst:
4         if element == item:
5             return True
6     return False
```

This function has a worst-case runtime complexity of $\Theta(n)$, where n is the number of elements in `lst`. This is because, in some cases, the loop must iterate through all n elements, checking each one against the target value. Since the desired element could be located anywhere in the list, the function may have to examine every element to find it.

However, if the list is **sorted**, we can improve efficiency by being strategic about which elements we check instead of scanning each one sequentially. For example, consider a sorted list `lst` containing one million numbers, and suppose we are searching for the number 111. If we check the middle element at index $500,000$, three possible cases arise:

- If $111 =$ `lst`$[500000]$, then we have found the element and can return `True`.
- If $111 <$ `lst`$[500000]$, then we know the number must be in the `left half` of the list, as it is sorted. This allows us to `ignore` all elements with index $\geq 500,000$.
- If $111 >$ `lst`$[500000]$, then it must be in the `right half` of the list, again due to sorting. We can `ignore` all elements with index $< 500,000$.

With just one comparison, we effectively eliminate half of the remaining elements. We can then repeat this process within either the left or right sublist until we either locate the target element or run out of elements to check. This approach is known as **binary search**, where the term "binary" refers to the fact that with each step, we halve the number of elements left to examine.

A key aspect of implementing binary search iteratively is maintaining the **current search range**, which progressively shrinks as new elements are checked. To achieve this, we use two variables, b and e, which represent the boundaries of this range. At any given moment in the algorithm, the search range corresponds to the list slice `lst[b:e]`.

These variables effectively partition the list into three distinct sections:

- `lst[0:b]` consists of elements that are **definitely less than** the target value.
- `lst[b:e]` represents the **current search range**, where elements may be less than, equal to, or greater than the item being searched for.
- `lst[e:len(lst)]` contains elements that are **definitely greater than** the target value.

```
from typing import Any

def binary_search(lst: list, item: Any) -> bool:
    """Return whether item is in lst using the binary search algorithm.

    Preconditions:
        - lst is sorted in non-decreasing order
    """
    b = 0
    e = len(lst)
```

At the beginning of the search, the entire list (`lst[0:len(lst)]`) is considered. The binary search algorithm utilizes a `while` loop to progressively narrow down the search range. The search process involves:

- Computing the midpoint m of the current range.
- Comparing `lst[m]` with the target `item`.
- If `item == lst[m]`, the function returns `True` immediately.
- If `item < lst[m]`, then all elements at indexes $\geq m$ are greater than `item`, so we update e.
- If `item > lst[m]`, then all elements at indexes $\leq m$ are smaller than `item`, so we update b.

```
def binary_search(lst: list, item: Any) -> bool:
    """Return whether item is in lst using the binary search algorithm.

    Preconditions:
        - lst is sorted in non-decreasing order
    """
    b = 0
    e = len(lst)

    while ...:
        m = (b + e) // 2
        if item == lst[m]:
            return True
        elif item < lst[m]:
            e = m
```

```
        else:  # item > lst[m]
            b = m + 1
```

To determine when the loop should stop, we consider when there are no more elements left to check—this occurs when `lst[b:e]` is empty. This happens when $b \geq e$, so negating this gives our loop condition:

```
def binary_search(lst: list, item: Any) -> bool:
    """Return whether item is in lst using the binary search algorithm.

    Preconditions:
        - lst is sorted in non-decreasing order
    """
    b = 0
    e = len(lst)

    while b < e:
        m = (b + e) // 2
        if item == lst[m]:
            return True
        elif item < lst[m]:
            e = m
        else:  # item > lst[m]
            b = m + 1

    # If the loop ends without finding the item, the item is not in the list.
    return False
```

In our binary search algorithm, the condition `b < e` is used instead of `b <= e` because of the way the search range is managed, and using `b < e` simplifies boundary handling in the binary search algorithm.

- `b` is the **inclusive** starting index.
- `e` is the **exclusive** ending index.

The search range consists of elements in `lst[b:e]`. What happens when `b == e`? If `b == e`, the search range is empty because `e` is exclusive (as long as e <= b, lst[b:e] will produce [ ] list.). The loop should stop since there are no more elements to check (indexing error for lst[0] when lst is empty!).

If we use `b <= e`, we must adjust the update rules, that is, adjusting e to stay within bounds:

```
def binary_search(lst: list, item: Any) -> bool:
    b = 0
    e = len(lst) - 1  # e is now inclusive

    while b <= e:
        m = (b + e) // 2
        if item == lst[m]:
            return True
        elif item < lst[m]:
            e = m - 1  # Adjusting e to stay within bounds
        else:  # item > lst[m]
            b = m + 1

    return False
```

Approach	Condition	`e` Update Rule	Interpretation
`b < e`	Stops when `b == e`	`e = m` (exclusive)	Uses a half-open range `[b, e)`
`b <= e`	Stops when `b > e`	`e = m - 1` (inclusive)	Uses a closed range `[b, e]`

With our implementation of `binary_search` complete, we now examine its running time. Earlier, we claimed that binary search is significantly faster than checking every element in the list—let's analyze why.

Although we will not conduct a formal proof, we can outline the key ideas behind the analysis. Our implementation relies on two variables, b and e, which change dynamically based on comparisons between `item` and `lst[m]` at each iteration. By focusing on the difference $e - b$, we can observe the following:

- Initially, $e - b$ is equal to n, the length of the input list.
- The loop terminates when $e - b \leq 0$.
- In each iteration, $e - b$ is reduced by at least a factor of 2.

By combining these observations, we conclude that `binary_search` executes at most $1 + \log_2 n$ iterations, with each iteration requiring constant time. Since the operations outside the loop also run in constant time, the worst-case running time of binary search is $O(\log n)$. This is significantly better than the linear search algorithm.

For large datasets requiring multiple search operations, it is often beneficial to **pre-sort** the list so that each subsequent search can be performed efficiently. As for the sorting process itself—that will be explored in the next section.

Binary search is very popular and powerful; sometimes it will not look exactly like the structure we wrote above. For example, binary search can not only do search but also approximation.

```python
import math
def sqrt_binary_search(x, precision=1e-6):
    if x < 0:
        raise ValueError("Cannot compute the square root of a negative number.")
    if x == 0 or x == 1:
        return x

    low, high = 0, x
    while high - low > precision:
        mid = (low + high) / 2
        if mid * mid < x:
            low = mid
        else:
            high = mid

    return (low + high) / 2  # Final approximation

print(sqrt_binary_search(2))
print(math.sqrt(2))
```

Adjusting `precision` to a smaller number will give you a more accurate approximation!

3.6 Sorting Algorithms

Why are sorting algorithms so important? Can you come up with some real-world applications?

1. Search Engines (Google, Bing): `Ranking Web Pages`: Google's PageRank algorithm sorts search results based on relevance, backlinks, and quality scores.
2. E-Commerce (Amazon, eBay): `Product Listings`: Products are sorted by price (low to high/high to low), ratings, popularity, and relevance.
3. Social Media (Facebook, Twitter, Instagram, TikTok): `News Feed and Trends`: Feeds are sorted by engagement (likes, shares, comments), recency, and user preferences.

3.6.1 Selection Sort

The selection sort algorithm is based on an intuitive principle. Given an unordered collection of elements, the algorithm repeatedly extracts the smallest element and appends it to a growing sorted list. The name `selection sort` originates from the fact that, at each step, the smallest remaining element is selected.

Consider the following list:

$$[3, 7, 2, 5]$$

The sorting process proceeds as follows:

- The smallest element is 2, which is placed in the sorted list: [2].

- The remaining elements are $[3, 7, 5]$. The smallest element is 3, so the sorted list updates to: $[2, 3]$.
- The remaining elements are $[7, 5]$. The smallest element is 5, yielding: $[2, 3, 5]$.
- The only remaining element is $[7]$, which is appended: $[2, 3, 5, 7]$.

This process can be summarized in the following table:

Items to be sorted	**Smallest element**	**Sorted list**
[3, 7, 2, 5]	2	[2]
[3, 7, 5]	3	[2, 3]
[7, 5]	5	[2, 3, 5]
[7]	7	[2, 3, 5, 7]
[]		[2, 3, 5, 7]

A simple implementation of selection sort in Python is given below:

```
def selection_sort_simple(lst: list) -> list:
    """Return a sorted version of lst."""
    sorted_so_far = []

    while lst != []:
        smallest = min(lst)
        lst.remove(smallest)
        sorted_so_far.append(smallest)

    return sorted_so_far
```

In the following section, we will introduce an alternative implementation of selection sort that directly mutates the input list, avoiding the creation of additional lists.

A sorting algorithm is considered *in-place* when it sorts a list by modifying the input directly rather than creating and using additional lists. This means that the algorithm rearranges elements within the given list without allocating extra memory for another list. While in-place algorithms may use a small, constant amount of auxiliary memory for variables such as integers, **this usage does not depend on the size of the input list (That is $O(1)$ memory utilization)**.

To implement an in-place version of selection sort, we do not create a separate list to store sorted elements. Instead, the algorithm swaps elements within the same list such that, at iteration i, the first i elements form the sorted section, while the rest remain unsorted.

The following example demonstrates how the list $[3, 7, 2, 5]$ is sorted using this in-place approach.

1. At iteration $i = 0$, the entire list is unsorted. The algorithm finds the smallest element in the list (which is 2) and swaps it with the item at index 0, resulting in the list:
$$[2, 7, 3, 5]$$
2. At iteration $i = 1$, the elements $[7, 3, 5]$ remain unsorted. The algorithm swaps 3 into index 1, producing:
$$[2, 3, 7, 5]$$
3. At iteration $i = 2$, the elements $[7, 5]$ are unsorted. The algorithm swaps 7 and 5, yielding:
$$[2, 3, 5, 7]$$
4. At iteration $i = 3$, only one element remains in the unsorted section. Since a single element is inherently sorted, no further swaps occur. The list is now fully sorted.

At each iteration i, the algorithm identifies the smallest element in the unsorted section `lst[i:]` and swaps it with `lst[i]`. The following implementation encapsulates this logic:

```
def selection_sort(lst: list) -> None:
    """Sort the given list using the selection sort algorithm.
    """
    for i in range(0, len(lst)):
        # Find the index of the smallest item in lst[i:] and swap that
        # item with the item at index i.
        index_of_smallest = _min_index(lst, i)
```

```
        lst[index_of_smallest], lst[i] = lst[i], lst[index_of_smallest] # tuple in Python

def _min_index(lst: list, i: int) -> int:
    """Return the index of the smallest item in lst[i:].
    """
    index_of_smallest_so_far = i

    for j in range(i + 1, len(lst)):
        if lst[j] < lst[index_of_smallest_so_far]:
            index_of_smallest_so_far = j

    return index_of_smallest_so_far
```

Worst-case analysis -- selection sort

We analyze the time complexity of our in-place selection sort algorithm. The first step is to examine the helper function `_min_index`. Let n denote the length of the input list `lst`.

- The operations outside the loop run in constant time, which we consider as a single step.
- The loop iterates $n - i - 1$ times, where $j = i + 1, \ldots, n - 1$, and since the loop body executes in constant time per iteration, the total loop execution takes $n - i - 1$ steps.
- Therefore, the overall time complexity of `_min_index` is $(n - i - 1) + 1$, which simplifies to $\Theta(n - i)$.

Now, we analyze the function `selection_sort`. Again, let n represent the length of the input list `lst`.

Within the loop body, there are two key operations:

- The first operation is the call to `_min_index`, which executes in $n-i$ steps, where i represents the current iteration index.
- The second operation is swapping elements in `lst`, which takes constant time (1 step).

Since the for-loop runs from $i = 0$ to $i = n - 1$, each iteration contributes $n - i + 1$ steps. Summing over all iterations, we obtain:

$$\sum_{i=0}^{n-1}(n - i + 1) = n(n + 1) - \sum_{i=0}^{n-1} i$$

Using the formula for the sum of the first $n - 1$ integers:

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

we simplify:

$$n(n + 1) - \frac{n(n-1)}{2}$$

$$= \frac{n(n+3)}{2}$$

The running time $\Theta(n - i)$ for `_min_index` has been precisely translated into an exact count of $n - i$ steps within the loop body. Thus, the time complexity of `selection_sort` is $\Theta(n^2)$. This is also equal to the average-case analysis, because no matter what the input is, it will check every element in the unsorted list and do one swap operation, there is no **early stop**!

3.6.2 Insertion Sort

In this section, we examine the **insertion sort** algorithm. The in-place version uses a loop with a variable, i, that marks the boundary between the sorted and unsorted portions of the list, `lst`.

During each loop iteration, the sublist `lst[:i]` is maintained in sorted order. Unlike selection sort—which picks the smallest element from the unsorted portion—insertion sort simply takes the next element, `lst[i]`, and inserts it into its correct position within the sorted sublist.

For example, consider performing in-place insertion sort on the list

$$[3, 0, 1, 8, 7].$$

1. `Iteration 0:`
   $i = 0$. The element at `lst[0]` is 3. Since any single-element sublist is already sorted, no changes are needed. Thus, `lst[:1]` is sorted.
2. `Iteration 1:`
   $i = 1$. The element `lst[1]` is 0, and the sorted sublist is [3]. To sort `lst[:2]`, we swap 0 and 3.
3. `Iteration 2:`
   $i = 2$. The element `lst[2]` is 1, with the sorted portion now being [0, 3]. A swap between 1 and 3 is performed to ensure that `lst[:3]` is sorted.
4. `Iteration 3:`
   $i = 3$. The element `lst[3]` is 8, and the sorted sublist is [0, 1, 3]. Since 8 is greater than the last element in the sorted portion, no swaps are needed, and the sorted part becomes `lst[:4]`.
5. `Iteration 4:`
   $i = 4$. The element `lst[4]` is 7. To sort `lst[:5]`, we swap 7 with 8. After this final iteration, the entire list is sorted.

This step-by-step process demonstrates how insertion sort builds a sorted list one element at a time by inserting each new element into its proper place.

Insertion sort employs a loop structure very similar to that of selection sort and maintains the key invariant that the list is divided into a *sorted* and an *unsorted* section. The following snippet shows the beginning of our algorithm:

```
def insertion_sort(lst: list) -> None:
    """Sort the given list using the insertion sort algorithm.

    Note: This is an in-place (mutating) function.
    """
    for i in range(0, len(lst)):
    # Implementation omitted
```

Unlike selection sort, insertion sort does not ensure that `lst[:i]` contains the i smallest elements overall. Instead, it simply takes the next element in the list (i.e., `lst[i]`) and inserts it into its correct position within the already sorted part.

To complete the insertion sort, we define a helper function `_insert` that performs the actual insertion. Since this helper is a bit more involved than its selection sort counterpart, we detail its specification below:

```
def _insert(lst: list, i: int) -> None:
    """Rearrange lst[i] so that lst[:i + 1] is sorted.

    Preconditions:
        - 0 <= i < len(lst)

    >>> lst = [7, 3, 5, 2]
    >>> _insert(lst, 1)
    >>> lst  # lst[:2] is sorted
    [3, 7, 5, 2]
    """
```

There are several ways to implement this helper function. One efficient approach is to traverse backwards, swapping the element at `lst[i]` with preceding elements until it is no longer less than its predecessor, or until the beginning of the list is reached.

```
#Version 1: Using an Early Return
def _insert(lst: list, i: int) -> None:
    for j in range(i, 0, -1):  # Iterate from i down to 1
        if lst[j - 1] <= lst[j]:
            return
        else:
```

```
            # Swap lst[j - 1] and lst[j]
            lst[j - 1], lst[j] = lst[j], lst[j - 1]
```

```
#Version 2: Using a Compound Loop Condition
def _insert(lst: list, i: int) -> None:
    j = i
    while j != 0 and lst[j - 1] > lst[j]::
        # Swap lst[j - 1] and lst[j]
        lst[j - 1], lst[j] = lst[j], lst[j - 1]
        j -= 1
```

Both versions achieve the same goal but are organized differently. Understanding each method is beneficial as it reinforces various looping techniques.

With the `_insert` helper in place, the main insertion sort loop can be completed by simply invoking `_insert`:

```
def insertion_sort(lst: list) -> None:
    """Sort the given list using the insertion sort algorithm.

    Note: This is an in-place (mutating) function.
    """
    for i in range(0, len(lst)):
        _insert(lst, i)
```

Worst-case analysis -- insertion sort

Let's examine the running time of our `insertion_sort` algorithm. We begin by focusing on the helper function `_insert`, recognizing that while we highlight the `early return` implementation here, the same reasoning applies to its counterpart using a compound loop condition. Because `_insert` may terminate early, its running time can vary across different inputs. We therefore need to investigate its worst-case behavior.

- **`_insert`**: Let $n \in \mathbb{N}$, and let `lst` be an arbitrary list of length n. Suppose we invoke `_insert(lst, i)` for some index $i < n$. Inside `_insert`, the loop can iterate at most i times (decrementing j from i down to 1), and each iteration takes constant time. Consequently, `_insert` requires at most i steps, implying a worst-case time complexity of $O(i)$.

  Consider an input list `lst` in which `lst[i]` is smaller than every element of `lst[:i]`. In this case, the condition `lst[j - 1]` $\leq$ `lst[j]` is false until $j = 0$. Therefore, the loop runs for all i iterations, giving a running time of i steps, or $\Theta(i)$. This matches the upper bound and establishes that the worst-case running time of `_insert` is $\Theta(i)$.

- **`insertion_sort`**: Let $n \in \mathbb{N}$, and let `lst` be a list of length n. The main loop of `insertion_sort` executes n times (for $i = 0, 1, \ldots, n-1$). In each iteration, `_insert(lst, i)` is called and can take up to i steps in the worst case. Each iteration then costs i steps. Summing these from $i = 0$ to $n - 1$ yields

$$\sum_{i=0}^{n-1} i \;=\; \frac{n(n-1)}{2},$$

  which is $O(n^2)$.

  To establish a matching lower bound, consider the list

$$[n-1,\, n-2,\, \ldots,\, 1,\, 0].$$

  In this configuration, every new element to be inserted is smaller than all elements already placed in the sorted portion. Consequently, each call to `_insert` requires i steps. Summing across all i again gives $\frac{n(n-1)}{2}$, which is $\Theta(n^2)$. This confirms that the worst-case running time of `insertion_sort` matches the upper bound.

Both our upper and lower bound analyses converge on the result that the worst-case running time of `insertion_sort` is $\Theta(n^2)$.

3.6.3 Selection Sort vs. Insertion Sort

One way to view each iteration of both selection sort and insertion sort is as having two phases:

1. Choosing which element from the unsorted portion to add to the sorted portion.
2. Adding that chosen element to the sorted portion.

The primary difference lies in how each algorithm balances these steps:

- In **selection sort**, the main effort goes into selecting the next item (the smallest remaining), while inserting that item into the sorted sublist is straightforward.
- In **insertion sort**, choosing which item to insert is trivial (simply the element at index i), but inserting it into the correct position within the sorted sublist requires more work.

At first glance, these approaches look `balanced`, since each one appears to have one straightforward task and one more challenging task. However, our understanding of algorithm analysis reveals deeper distinctions. Although both have worst-case running times on the order of $\mathcal{O}(n^2)$, their actual performance patterns differ significantly:

- **Selection sort** consistently performs $\mathcal{O}(n^2)$ operations, regardless of the input's arrangement.
- **Insertion sort** has a broad spectrum of running times and can be considerably more efficient on many inputs, especially when the list is already sorted or nearly sorted.

Let's do average-case analysis on both sorting algorithms:

`insertion sort`

1. **Key Idea.** In each iteration i, where the sublist of already-sorted elements has length i, the next element to be inserted is equally likely to land in any position among those $i+1$ elements. Consequently, on average, it shifts through about $\frac{i}{2}$ elements.
2. **Expected Cost Per Iteration.** Hence, the expected number of comparisons or shifts in iteration i is approximately $\frac{i}{2}$.
3. **Summation Over All Iterations.** Summing $\frac{i}{2}$ from $i = 1$ to $n-1$ gives:

$$\sum_{i=1}^{n-1} \frac{i}{2} \;=\; \frac{1}{2} \cdot \frac{n(n-1)}{2} \;=\; O(n^2).$$

   Therefore, the average-case running time for insertion sort is $O(n^2)$.

`selection sort`

Based on the previous analysis, its average-case is same to worst-case:

$$\frac{n(n+3)}{2}$$

From the math expression, we should have a conclusion that insertion should be a little bit faster than selection sort, now let's write a simple program to demonstrate it:

```python
# We will test two types of inputs:
# 1. Completely random lists
# 2. Nearly sorted lists
# We will see that insertion sort benefits more from nearly sorted lists.
import time
import random
def time_function(func, data):
    start = time.time()
    func(data)
    end = time.time()
    return end - start

n = 2000  # Change this size as desired for a bigger test

# 1. Generate a random list
random_list_selection = [random.randint(0, 10000) for _ in range(n)]
random_list_insertion = list(random_list_selection)  # Make a copy

# 2. Generate a nearly sorted list
# We can start with a sorted list and then perform a small number of swaps.
nearly_sorted_list = list(range(n))
```

```python
# Randomly swap a small percentage of elements to introduce "slight disorder"
num_swaps = max(1, n // 50)  # e.g., ~2% swaps
for _ in range(num_swaps):
    i = random.randint(0, n-1)
    j = random.randint(0, n-1)
    nearly_sorted_list[i], nearly_sorted_list[j] = nearly_sorted_list[j], nearly_sorted_list[i]

nearly_sorted_list_selection = list(nearly_sorted_list)  # copy
nearly_sorted_list_insertion = list(nearly_sorted_list)  # copy

# --- Time Selection Sort on random list ---
sel_time_random = time_function(selection_sort, random_list_selection)
# --- Time Insertion Sort on random list ---
ins_time_random = time_function(insertion_sort, random_list_insertion)

# --- Time Selection Sort on nearly-sorted list ---
sel_time_nearly = time_function(selection_sort, nearly_sorted_list_selection)
# --- Time Insertion Sort on nearly-sorted list ---
ins_time_nearly = time_function(insertion_sort, nearly_sorted_list_insertion)

# Print results
print(f"List size (n): {n}")
print("--- Random List Performance ---")
print(f"Selection sort: {sel_time_random:.5f} seconds")
print(f"Insertion sort: {ins_time_random:.5f} seconds")

print("\n--- Nearly Sorted List Performance ---")
print(f"Selection sort: {sel_time_nearly:.5f} seconds")
print(f"Insertion sort: {ins_time_nearly:.5f} seconds")
```

The output is

```
List size (n): 2000
--- Random List Performance ---
Selection sort: 0.06518 seconds
Insertion sort: 0.11396 seconds

--- Nearly Sorted List Performance ---
Selection sort: 0.06492 seconds
Insertion sort: 0.00602 seconds
```

In the case of a completely random list, insertion sort actually performs worse! This is because it requires numerous swap operations, leading to a high number of write operations, which are computationally expensive. However, in real-world applications, we often encounter nearly sorted lists, where insertion sort excels. In such scenarios, insertion sort can be up to 10 times faster than selection sort! Consequently, even though these algorithms share conceptual and structural similarities, insertion sort is often the more desirable choice. Its ability to take advantage of partially sorted inputs makes it much more efficient in many practical scenarios.

3.6.4 Merge Sort

Divide and Conquer is a fundamental algorithmic paradigm used to solve complex problems efficiently. It follows a three-step recursive approach:

- `Divide` – Break the original problem into smaller subproblems that are similar to the original problem.
- `Conquer` – Recursively solve each subproblem.
- `Combine` – Merge the results of the subproblems to obtain the final solution.

This approach is especially useful for problems where breaking them into smaller parts significantly reduces the computational complexity.

The first divide-and-conquer sorting algorithm we will study is **merge sort**. This algorithm follows three main steps:

1. Divide the input list into two halves.
2. Recursively sort each half.

3. Merge the sorted halves together.

As evident from this breakdown, the simpler part of merge sort is dividing the list into two halves, while the more complex part is merging the sorted halves into a single, fully sorted list. Below is the specification of the merge sort function in Python. We use a *non-mutating* version, meaning it does not modify the original list but instead returns a new sorted list.

```
def mergesort(lst: list) -> list:
    """
    Return a new sorted list with the same elements as lst.
    This is a *non-mutating* version of mergesort; it does not modify the input list.
    """
```

Since our implementation is recursive, we need a base case. The simplest base case occurs when the list has fewer than two elements, as it is already sorted.

```
def mergesort(lst: list) -> list:
    """
    Return a new sorted list with the same elements as lst.
    This is a *non-mutating* version of mergesort; it does not modify the input list.
    """
    if len(lst) < 2:
        return lst
    else:
        ...
```

For the recursive step, we divide the list into two halves, recursively sort each half, and then merge the sorted halves. The first two steps are straightforward:

```
def mergesort(lst: list) -> list:
    """
    Return a new sorted list with the same elements as lst.
    This is a *non-mutating* version of mergesort; it does not modify the input list.
    """
    if len(lst) < 2:
        return lst
    else:
        # Divide the list into two halves and sort them recursively.
        mid = len(lst) // 2
        left_sorted = mergesort(lst[:mid])
        right_sorted = mergesort(lst[mid:])
        ...
```

The key challenge in mergesort is efficiently merging two sorted lists. To handle this, we define a helper function:

```
def _merge(lst1: list, lst2: list) -> list:
    """Return a sorted list with the elements from lst1 and lst2.

    Preconditions:
        - lst1 is sorted
        - lst2 is sorted
    """
```

The merging process is similar to selection sort: we build a sorted list by repeatedly choosing the smallest element from `lst1` or `lst2`. However, since both lists are already sorted, we can efficiently determine the smallest element by comparing `lst1[0]` and `lst2[0]`.

For example, given the lists:

$$\texttt{lst1} = [-1, 3, 7, 10], \quad \texttt{lst2} = [-3, 0, 2, 6]$$

we can merge them by comparing elements at their respective starting positions. The following table illustrates how elements are merged step by step:

Unmerged items in `lst1`	Unmerged items in `lst2`	Comparison	Sorted list
[-1, 3, 7, 10]	[-3, 0, 2, 6]	-1 vs. -3	[-3]
[-1, 3, 7, 10]	[0, 2, 6]	-1 vs. 0	[-3, -1]
[3, 7, 10]	[0, 2, 6]	3 vs. 0	[-3, -1, 0]
[3, 7, 10]	[2, 6]	3 vs. 2	[-3, -1, 0, 2]
[3, 7, 10]	[6]	3 vs. 6	[-3, -1, 0, 2, 3]
[7, 10]	[6]	7 vs. 6	[-3, -1, 0, 2, 3, 6]
[7, 10]	[]	N/A	[-3, -1, 0, 2, 3, 6, 7, 10]

Once one of the lists is exhausted, we can append the remaining elements, as they are already sorted.

Following this approach, we implement the merge operation using an index-based method:

```
def _merge(lst1: list, lst2: list) -> list:
    """Return a sorted list with the elements in lst1 and lst2.

    Preconditions:
        - lst1 is sorted
        - lst2 is sorted

    >>> _merge([-1, 3, 7, 10], [-3, 0, 2, 6])
    [-3, -1, 0, 2, 3, 6, 7, 10]
    """
    i1, i2 = 0, 0
    sorted_so_far = []

    while i1 < len(lst1) and i2 < len(lst2):

        if lst1[i1] <= lst2[i2]:
            sorted_so_far.append(lst1[i1])
            i1 += 1
        else:
            sorted_so_far.append(lst2[i2])
            i2 += 1

    # When the loop exits, either lst1 or lst2 has been fully merged.
    return sorted_so_far + lst1[i1:] + lst2[i2:]
```

With the merge function complete, we can finalize our mergesort implementation:

```
def mergesort(lst: list) -> list:
    """..."""
    if len(lst) < 2:
        return lst
    else:
        # Divide the list into two halves and sort them recursively.
        mid = len(lst) // 2
        left_sorted = mergesort(lst[:mid])
        right_sorted = mergesort(lst[mid:])

        # Merge the two sorted halves.
        return _merge(left_sorted, right_sorted)
```

If you are still not very clear about recursive function call and steps of merge sort, please check the visualization tool here:

`https://labuladong.online/algo/en/data-structure-basic/merge-sort/`, and expand the section **Algorithm Visualization** in the webpage.

Runtime Analysis -- Merge Sort

To determine the runtime of mergesort, we need to examine the recursive calls. Suppose we begin with a list of size n, where $n > 1$. For simplicity, we assume n is a power of 2. While the analysis here applies when n is not a power of 2, it becomes more complex since the list may not always be divided into equal halves.

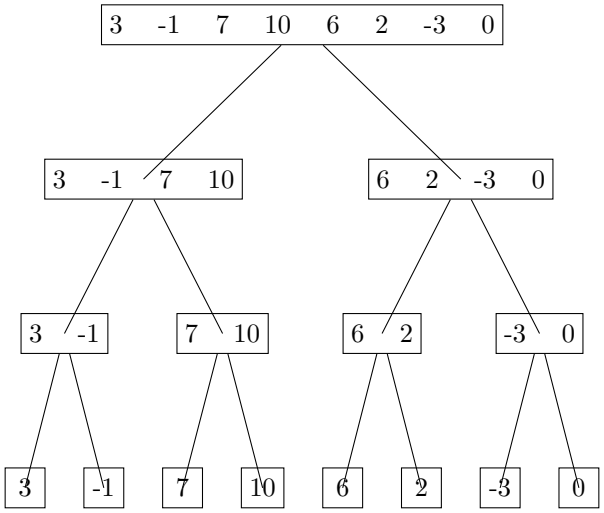
During each recursive step, the list is split into two sublists of length $\frac{n}{2}$, and mergesort is applied to each. This recursive division continues, with each subsequent call further splitting the list into halves. Each of these recursive calls, in turn, divides their sublists, leading to inputs of size $\frac{n}{4}$, and so on.

This pattern persists, where every call to `mergesort` processes a list of size $\frac{n}{2^k}$, for some $k \in \mathbb{N}$. The recursion terminates when mergesort is called on a list of size 1, which serves as the base case.

For instance, consider the sequence of lists passed as inputs to `mergesort` when applied to the list:

$$[3, -1, 7, 10, 6, 2, -3, 0]$$

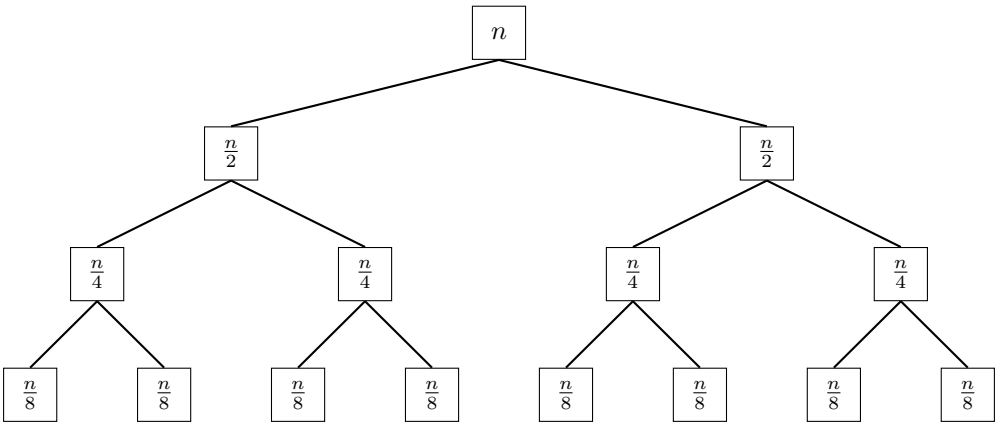
The function will recursively divide the list into smaller sublists until the base case is reached.



In a mergesort process, each recursive call creates a `binary recursion tree`, where every non-base-case call generates two more recursive calls—one for the left half of the input and one for the right half. Additionally, because the input size reduces by a factor of `2` at each recursive step, the height of the recursion tree can be determined.

For an input list of size n, where n is a power of 2, the tree will have exactly $\log_2(n) + 1$ levels.

Next, we analyze the non-recursive `_merge` running time of mergesort. Suppose we have a list of length n, where $n \geq 2$ and is a power of 2. In a recursive step, since both `left` and `right` sublists have size $\frac{n}{2}$, and merging takes $len(left) + len(right)$ steps, thus, the total runtime per `_merge` call is shown in the tree structure below:



With the recursion tree established, the final step is to sum up all the contributions to the total running time. At first glance, this may seem complex, as different levels contain varying numbers of computations. As a result, we cannot simply multiply the number of nodes by the non-recursive running time of a single node.

A crucial observation is that **each level in the recursion tree has the same total running time**. Since the tree follows a binary branching pattern, we can derive a formula for the number of nodes at any given depth.

- At depth d, the tree contains 2^d nodes. - Each node at depth d contributes a cost of $\frac{n}{2^d}$.

Thus, the total work done at depth d is:

$$2^d \cdot \frac{n}{2^d} = n$$

Since the depth starts at 0, with the root node at depth 0, this calculation shows that every level in the recursion tree performs the same amount of work, equal to n.

Since each level contains n total work, and the tree has $\log_2(n) + 1$ levels, the final worst-case runtime of mergesort is:

$$\Theta(n \log n)$$

This is significantly more efficient than the worst-case time complexities of selection sort and insertion sort, which run in $\Theta(n^2)$.

3.7 Quick Sort

To understand Quicksort, consider sorting a group of individuals alphabetically by their last names. We first divide them into two groups: - Those whose last names start with A to L. - Those whose last names start with M to Z.

Although each group is not yet sorted, we already know that all individuals in the A-L group should come before those in the M-Z group. Once we sort each subgroup independently, we can concatenate them to form a completely sorted list.

This concept forms the foundation of Quicksort, where we choose a **pivot**, split the list into two parts, recursively sort each part, and then merge the sorted halves.

To translate this idea into an algorithm, we follow the `divide-and-conquer` paradigm:

1. Select a `pivot` element from the list.
2. Partition the list into two groups:
   - Elements less than or equal to the pivot.
   - Elements greater than the pivot.
3. Recursively sort both partitions.
4. Finally, concatenate the sorted partitions, placing the pivot between them.

We illustrate these steps using the input list:

$$[3, -1, 7, 10, 6, 2, -3, 0]$$

where the first element is chosen as the pivot.

1. We choose 3 as the pivot and partition the list:
2. Divide:
$$[-1, 2, -3, 0] \quad \textbf{Pivot: 3} \quad [7, 10, 6]$$
3. Recursively sort:
$$[-3, -1, 0, 2] \qquad [6, 7, 10]$$
4. Merge:
$$[-3, -1, 0, 2, 3, 6, 7, 10]$$

Below is our implementation of a non-mutating version of quicksort. The structure is quite similar to our mergesort implementation, but the key difference lies in the divide and combine steps. The divide step is more complex and handled by a helper function called `_partition`.

```
def quicksort(lst: list) -> list:
    """
    Return a sorted list with the same elements as lst.
    This is a *non-mutating* version of quicksort; it does not mutate the
    input list.
    """
    if len(lst) < 2:
```

```
        return lst.copy()
    else:
        # Divide the list into two parts by picking a pivot and then partitioning the list.
        # In this implementation, we're choosing the first element as the pivot, but
        # we could have made lots of other choices here (e.g., last, random).
        pivot = lst[0]
        smaller, bigger = _partition(lst[1:], pivot)

        # Sort each part recursively
        smaller_sorted = quicksort(smaller)
        bigger_sorted = quicksort(bigger)

        # Combine the two sorted parts. No need for a helper here!
        return smaller_sorted + [pivot] + bigger_sorted
```

The `_partition` helper function is simpler than the `_merge` helper in mergesort. This is because we can implement partitioning using a single loop through the list:

```
def _partition(lst: list, pivot: Any) -> tuple[list, list]:
    """Return a partition of lst with the chosen pivot.

    Return two lists, where the first contains the items in lst
    that are <= pivot, and the second contains the items in lst that are > pivot.
    """
    smaller = []
    bigger = []

    for item in lst:
        if item <= pivot:
            smaller.append(item)
        else:
            bigger.append(item)

    return (smaller, bigger)
```

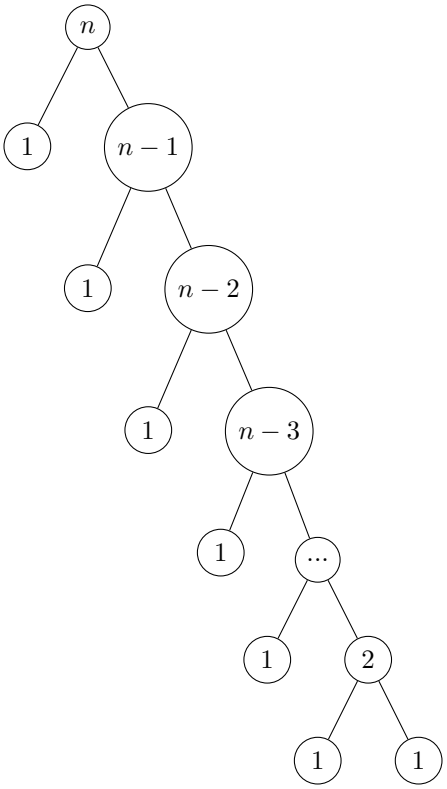
And that's quicksort! At least, this is the non-mutating version. Interestingly, unlike mergesort, quicksort can be implemented in-place without much difficulty. Leave it as a question in your weekly assignment.

Runtime Analysis -- Quick Sort

First let's start with the parts that are similar. The non-recursive running time of **_partition** is also $\Theta(n)$, where n is the length of the input list. The key difference is that with mergesort we know that we're splitting up the list into two equal halves (each of size $\frac{n}{2}$); this isn't necessarily the case with quicksort!

Suppose we get lucky, and at each recursive call we choose the pivot to be the median of the list, so that the two partitions both have size (roughly) $\frac{n}{2}$. In this case, the recursion tree is the same as mergesort, and we get a $\Theta(n \log n)$ runtime.

But this relies on making an assumption about our choice of pivot. Consider another possibility: what if we're always extremely unlucky, and always choose the *smallest* list element? In this case, the partitions are as uneven as possible, with `smaller` having no elements, and `bigger` having all remaining $n - 1$ elements. This results in a very lopsided recursion diagram, with the following non-recursive running times. Note that the "1" nodes down the tree represent the running time of making the recursive calls on an empty list.



In this case, the height of the tree is n, since the size of the `bigger` partition just decreases by 1 at each call. There are $n-1$ recursive calls on empty partitions (`smaller`). Adding the total non-recursive running time gives us:

$$\left(\sum_{i=1}^{n} i\right) + (n-1) = \frac{n(n+1)}{2} + n - 1$$

And so for this case, the running time is actually $\Theta(n^2)$, just like selection sort and insertion sort!

In fact, these two cases mark the extremes of the running time ranges for quicksort. It's possible to show (with a more formal analysis) that the worst-case running time for this implementation of quicksort is $\Theta(n^2)$ and the best-case (i.e., minimum) running time is $\Theta(n \log n)$. So it seems like quicksort isn't actually so quick after all, at least in the worst case. The choice of pivot is extremely important, because if we repeatedly choose pivots that result in uneven partitions, the running time increases significantly. In our naive implementation of quicksort, where the first list item is chosen as the pivot, this actually means that getting an already-sorted list as input results in this poor $\Theta(n^2)$ running time — not great!

If you are still not very clear about recursive function call and steps of quick sort, please check the visualization tool here:

`https://labuladong.online/algo/en/data-structure-basic/quick-sort/`, and expand the section **Algorithm Visualization** in the webpage.

3.8 Mergesort vs. Quicksort

After analyzing the running times of mergesort and quicksort, one might wonder why quicksort is often preferred in practice despite its higher worst-case complexity. The answer lies in real-world performance factors that go beyond theoretical asymptotic analysis.

One of the strengths of asymptotic Theta notation is that it simplifies our running time analysis by ignoring constant factors. However, this simplification also has a side effect—it tends to flatten running time differences, making all $\Theta(n \log n)$ algorithms appear similar. **In practice, an in-place implementation of quicksort is often faster than mergesort for most inputs.** This is because quicksort generally has smaller constant factors, performs fewer low-level operations, and exhibits better cache efficiency compared to mergesort.

A common question arises: What sorting algorithm does Python's built-in `sorted()` function and `list.sort()` use?

Python employs a highly optimized sorting algorithm known as **Timsort**, invented by Tim Peters in 2002 specifically for Python. Timsort is essentially a refined version of mergesort, optimized for real-world data. It merges sorted sublists efficiently but also incorporates additional optimizations, such as:

- Using insertion sort for sorting small sublists, which speeds up performance.
- Dynamically adapting to patterns in the input data, making it significantly faster in practice.

This highlights the fact that worst-case asymptotic complexity does not always tell the full story when it comes to real-world sorting performance.

3.9 Summary of Sorting Algorithms

Algorithm	**Best Case**	**Average Case**	**Worst Case**	**Space Complexity**
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$ (In-place)
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$ (In-place)
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$ (Not In-place)
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$ (In-place)

4 Stack

The Stack ADT is a simple and fundamental data structure. A stack consists of zero or more items. When an item is added, it is placed on top of the stack (this operation is known as *pushing*). Similarly, when an item is removed, it is taken from the top (this operation is known as *popping*).

The term *stack* is a metaphor derived from a stack of books on a table. The first item added is the last one removed, a principle known as **Last-In-First-Out (LIFO)**.

4.1 Stack ADT Definition

Data: A collection of items.

Basic Operations:

- Check if the stack is empty.
- Add an item to the stack (push).
- Remove the most recently added item (pop).

4.2 Implementation in Python

Below is an example implementation of a Stack in Python:

```
from typing import Any
class Stack:
    """A last-in-first-out (LIFO) stack of items.

    Stores data in last-in, first-out order. When removing an item from the
    stack, the most recently-added item is the one that is removed.

    """
    def __init__(self) -> None:
        """Initialize a new empty stack."""

    def is_empty(self) -> bool:
        """Return whether this stack contains no items."""

    def push(self, item:Any) -> None:
        """Add a new element to the top of this stack."""

    def pop(self) -> Any:
        """Remove and return the element at the top of this stack.

        Preconditions:
            - not self.is_empty()
        """

    def peek(self):
        """Return the item at the top of the stack without removing it.

        Raise an EmptyStackError if this stack is empty.

        Preconditions:
            - not self.is_empty()
        """
```

Abstraction allows us to differentiate the conceptual definition of the Stack ADT from its implementation. Despite their limited operations, stacks are highly useful. Below are some notable applications:

- **Call Stack in Programming:** When a function `f` calls `g`, which in turn calls `h`, execution returns to `g` when `h` completes, and then back to `f`. This is managed via a call stack, with function frames stacked and removed in LIFO order.
- **Undo Functionality:** Many applications implement undo functionality using a stack. The most recent action is undone first, making the Stack ADT ideal for tracking action history.
- **Web Browser Back Navigation:** Browsers store visited pages in a stack, allowing users to navigate back to the most recently visited page.

4.3 Implementation of the Stack ADT Using Python Lists

In this section, we will implement the Stack Abstract Data Type (ADT) using Python's built-in list data structure. **We choose to use the end of the list to represent the top of the stack.** Since Python lists support efficient appends and pops from the end, this implementation ensures good performance for stack operations.

```
from typing import Any
class Stack:
    """A last-in-first-out (LIFO) stack of items.

    Stores data in a last-in, first-out order. When removing an item from the
    stack, the most recently added item is the one that is removed.

    Instance Attributes:
        - items: The items stored in the stack. The end of the list represents
          the top of the stack.
    """
    items: list

    def __init__(self) -> None:
        """Initialize a new empty stack."""
        self.items = []

    def is_empty(self) -> bool:
        """Return whether this stack contains no items."""
        return self.items == []

    def push(self, item:Any) -> None:
        """Add a new element to the top of this stack."""
        self.items.append(item)

    def pop(self) -> Any:
        """Remove and return the element at the top of this stack.

        Preconditions:
            - not self.is_empty()
        """
        return self.items.pop()

    def peek(self):
        """Return the item at the top of the stack without removing it.

        Raise an EmptyStackError if this stack is empty.

        Preconditions:
            - not self.is_empty()
        """

        return self.items[-1]

```

The current implementation of the `Stack` class is functionally correct but has a subtle distinction from the Stack ADT it aims to represent. While users can instantiate a `Stack` object and use its methods `push` and `pop` as intended, they also have direct access to its `items` attribute. This means users can access, modify, or manipulate the stack's contents directly, potentially altering its behavior in unintended ways.

4.4 Public vs. Private Attributes

In our current implementation, the instance attribute `items` is part of the class's public interface, meaning users can reasonably expect to interact with it. To designate an attribute as not part of the public interface, we prefix its name with an underscore (_). Attributes with an underscore prefix are referred to as *private instance attributes*, while those without are considered *public instance attributes*.

Public attributes are part of the class interface and intended for external use, whereas private attributes are meant for internal use within the class. In professional software development, attributes are often categorized into three access

levels: public, private, and protected. To simplify, we will only consider the first two, though our definition of *private* partially overlaps with the common interpretation of *protected*.

Below is the modified `Stack` implementation, where `items` has been changed to a private attribute:

```
from typing import Any
class Stack:
    """A last-in-first-out (LIFO) stack of items.

    Stores data in first-in, last-out order. When removing an item from the
    stack, the most recently-added item is the one that is removed.

    """

    # Private Instance Attributes:
    #   - _items: The items stored in the stack. The end of the list represents
    #     the top of the stack.
    _items: list

    def __init__(self) -> None:
        """Initialize a new empty stack."""
        self._items = []

    def is_empty(self) -> bool:
        """Return whether this stack contains no items."""
        return self._items == []

    def push(self, item:Any) -> None:
        """Add a new element to the top of this stack."""
        self._items.append(item)

    def pop(self) -> Any:
        """Remove and return the element at the top of this stack.

        Preconditions:
            - not self.is_empty()
        """
        return self._items.pop()

    def peek(self):
        """Return the item at the top of the stack without removing it.

        Raise an EmptyStackError if this stack is empty.

        Preconditions:
            - not self.is_empty()
        """

        return self._items[-1]

```

4.5 Revisit Merge Sort

One of the widely used applications of the Stack data structure is the Call Stack in computer systems. Now, let's utilize our Stack implementation to create a Call Stack that tracks the data flow within a merge sort.

```
from src.stack import Stack

call_stack = Stack()

def mergesort_with_stack(lst):
    call_stack.push(("mergesort", lst))
    if len(lst) < 2:
        return lst
```

```
    else:
        mid = len(lst) // 2

        left_sorted = mergesort_with_stack(lst[:mid])

        right_sorted = mergesort_with_stack(lst[mid:])


        return _merge(left_sorted, right_sorted)

def _merge(left, right):
    call_stack.push(("merge", (left, right)))
    sorted_lst = []
    l = 0
    r = 0

    while l < len(left) and r < len(right):
        if left[l] <= right[r]:
            sorted_lst.append(left[l])
            l += 1
        else:
            sorted_lst.append(right[r])
            r += 1
    return sorted_lst + left[l:] + right[r:]

# Example usage

lst = [64, 34, 25, 12, 22, 11, 90]
sorted_array = mergesort_with_stack(lst)

while not call_stack.is_empty():
    func_name, args = call_stack.pop()

    if func_name == "mergesort":
        print(f"mergesort({args})")
    else:  # merge
        print(f"merge({args[0]}, {args[1]})")
```

4.6 Handle the Preconditions

The stack implementations we studied in the previous section included a precondition on their `pop` method specifying that the stack must not be empty. Consider this version of `Stack.pop`, which removes the precondition but keeps the same implementation:

```
def pop(self) -> Any:
    """Remove and return the element at the top of this stack."""
    return self._items.pop()
```

When calling `pop` on an empty stack, the following error occurs:

```
>>> s = Stack()
>>> s.pop()
Traceback (most recent call last):
  File "./main.py", line 38, in <module>
    call_stack.pop()
  File "./src/stack.py", line 33, in pop
    return self._items.pop()
IndexError: pop from empty list
```

From the perspective of client code, it is helpful to see an exception as an indication that something has gone

wrong. However, the error message refers to a list (`IndexError: pop from empty list`) and a private attribute (`self._items`), which the client code should have no knowledge of.

A better solution is to raise a custom exception that is descriptive yet does not reveal any implementation details. This can be achieved easily in Python by defining a new exception class:

```
class EmptyStackError(Exception):
    """Exception raised when calling pop on an empty stack."""
```

The syntax `(Exception)` following the class name allows us to create a new type of exception. This mechanism, known as *inheritance*.

```
def pop(self) -> Any:
    """Remove and return the element at the top of this stack.

    Raise an EmptyStackError if this stack is empty.
    """
    if self.is_empty():
        raise EmptyStackError
    else:
        return self._items.pop()
```

When calling `pop` on an empty stack, we now get:

```
>>> s = Stack()
>>> s.pop()
Traceback (most recent call last):
  File "./main.py", line 38, in <module>
    call_stack.pop()
  File "./src/stack.py", line 34, in pop
    raise EmptyStackError
src.stack.EmptyStackError
```

As before, an exception is raised, but this time, the error message does not reference implementation details such as `self._items`. Instead, it clearly indicates that an `EmptyStackError` occurred, as documented in the method docstring.

A limitation of the above approach is that simply using the exception class name does not convey much information. To provide a more informative error message, we can define a special `__str__` method in our exception class:

```
class EmptyStackError(Exception):
    """Exception raised when calling pop on an empty stack."""

    def __str__(self) -> str:
        """Return a string representation of this error."""
        return 'pop may not be called on an empty stack'
```

Now, when we attempt to pop from an empty stack:

```
>>> s = Stack()
>>> s.pop()
Traceback (most recent call last):
  File "./main.py", line 38, in <module>
    call_stack.pop()
  File "./src/stack.py", line 34, in pop
    raise EmptyStackError
src.stack.EmptyStackError: pop may not be called on an empty stack
```

This approach provides a more user-friendly error message, helping users understand the issue without exposing internal implementation details. We did the same modification to our `peek()` function:

```
def peek(self):
    """Return the item at the top of the stack without removing it.

    Raise an EmptyStackError if this stack is empty.
    """
    if not self.is_empty():
        return self._items[-1]
    raise EmptyStackError
```

4.7 Classic Problem: Valid Parentheses Check

Given a string containing only the characters '(', ')', '', '', '[' and ']', determine if the input string is valid. A string is valid if:

- An open bracket must have a corresponding closing bracket of the same type.
- A closing bracket must follow its corresponding open bracket in the correct order.
- A closing bracket cannot appear without a preceding unmatched open bracket of the same type.
- The string is valid if all open brackets are properly closed, leaving no unmatched brackets.

```
Example 1:
Input: "{[()]}"
Output: True

Example 2:
Input: "{[(])}"
Output: False
```

```
def is_valid_parentheses(s: str) -> bool:
    stack = Stack()
    for char in s:
        if char in "({[":
            stack.push(char)
        elif char == ")" and not stack.is_empty() and stack.pop() == "(":
            continue
        elif char == "}" and not stack.is_empty() and stack.pop() == "{":
            continue
        elif char == "]" and not stack.is_empty() and stack.pop() == "[":
            continue
        else:
            return False
    return stack.is_empty()  # Valid if stack is empty (all matched)
```

5 Queue

A queue is another collection of data that adds and removes items in a fixed order. Unlike a stack, items come out of a queue in the order in which they entered. We call this behavior **First-In-First-Out (FIFO)**.

ADT: Queue

- **Data:** A collection of items.
- **Operations:**
  - Determine whether the queue is empty.
  - Add an item (enqueue).
  - Remove the least recently added item (dequeue).

```
from typing import Any

class Queue:
    """A first-in-first-out (FIFO) queue of items.

    Stores data in a first-in, first-out order. When removing an item from the
    queue, the most recently-added item is the one that is removed.
    """

    def __init__(self) -> None:
        """Initialize a new empty queue."""

    def is_empty(self) -> bool:
        """Return whether this queue contains no items."""

    def enqueue(self, item: Any) -> None:
        """Add <item> to the back of this queue."""

    def dequeue(self) -> Any:
        """Remove and return the item at the front of this queue.

        Raise an EmptyQueueError if this queue is empty.
        """

    def peek(self) -> Any:
        """Return the item at the front of the queue without removing it.

        Raise an EmptyQueueError if this queue is empty.
        """

class EmptyQueueError(Exception):
    """Exception raised when calling dequeue on an empty queue."""

    def __str__(self) -> str:
        """Return a string representation of this error."""
        return 'dequeue may not be called on an empty queue'
```

5.1 List-based implementation

In the following implementation, we use a Python list that is hidden from the user. We have chosen that the beginning of the list (i.e., index 0) represents the front of the queue. This means that new items that are enqueued will be added at the end of the list, and items that are dequeued will be removed from the beginning of the list.

```
class Queue:
    """A first-in-first-out (FIFO) queue of items.

    Stores data in a first-in, first-out order. When removing an item from the
    queue, the most recently-added item is the one that is removed.
    """
```

```
    _items: list

    def __init__(self) -> None:
        """Initialize a new empty queue."""
        self._items = []

    def is_empty(self) -> bool:
        """Return whether this queue contains no items."""
        return self._items == []

    def enqueue(self, item: Any) -> None:
        """Add <item> to the back of this queue."""
        self._items.append(item)

    def dequeue(self) -> Any:
        """Remove and return the item at the front of this queue.

        Raise an EmptyQueueError if this queue is empty.
        """
        if self.is_empty():
            raise EmptyQueueError
        else:
            return self._items.pop(0)

    def peek(self) -> Any:
        """Return the item at the front of the queue without removing it.

        Raise an EmptyQueueError if this queue is empty.
        """
        if self.is_empty():
            raise EmptyQueueError
        else:
            return self._items[-1]
```

Our `Queue.enqueue` implementation calls `list.append`, which we know operates in constant $O(1)$ time. However, the `Queue.dequeue` method calls `self._items.pop(0)`, which takes $O(n)$ time (where n is the number of items stored in the queue). If we instead designated the front of the queue as the end of the list (rather than the beginning), we would simply swap these running times. This presents a trade-off: using an array-based list, we can either have an efficient enqueue operation or an efficient dequeue operation, but not both.

5.2 Stack-based Implementation

`pop(0)` is too expensive; is there a better way to speed up this `dequeue` process? We use two stacks:

- **Stack 1 (Input Stack)**: Used for enqueue operations (pushing elements).
- **Stack 2 (Output Stack)**: Used for dequeue operations (popping elements).

The operations are:

- **Enqueue (Push operation)**: Always push new elements onto **Stack 1**.
- **Dequeue (Pop operation)**:
  - If **Stack 2** is empty:
    - Move all elements from **Stack 1** to **Stack 2**.
    - Pop the top element from **Stack 2**.
  - If **Stack 2** is not empty:
    - Directly pop the top element.

```
from src.stack import Stack,EmptyQueueError
from typing import Any


class Queue:
```

```
    """A first-in-first-out (FIFO) queue of items.

    Stores data in a first-in, first-out order. When removing an item from the
    queue, the most recently-added item is the one that is removed.
    """

    def __init__(self) -> None:
        """Initialize a new empty queue."""
        self._stack1 = Stack()  # Input stack
        self._stack2 = Stack()  # Output stack

    def is_empty(self) -> bool:
        """Return whether this queue contains no items."""
        return self._stack1.is_empty() and self._stack2.is_empty()

    def enqueue(self, item: Any) -> None:
        """Add <item> to the back of this queue."""
        self._stack1.push(item)

    def dequeue(self) -> Any:
        """Remove and return the item at the front of this queue.

        Raise an EmptyQueueError if this queue is empty.
        """
        if self._stack2.is_empty():
            while not self._stack1.is_empty():
                self._stack2.push(self._stack1.pop())
        if self._stack2.is_empty():
            raise EmptyQueueError
        return self._stack2.pop()

    def peek(self) -> Any:
        """Return the item at the front of the queue without removing it.

        Raise an EmptyQueueError if this queue is empty.
        """
        if self._stack2.is_empty():
            while not self._stack1.is_empty():
                self._stack2.push(self._stack1.pop())

        if self._stack2.is_empty():
            raise EmptyQueueError

        return self._stack2.peek()  # Assuming Stack has a peek() method
```

Using **two stacks** to implement a queue is significantly more efficient than using a Python list with `pop(0)`. Because:

- Using a Python list as a queue with `pop(0)` is inefficient because it requires shifting all elements, making it $O(n)$.
- In the two-stack queue implementation, we use only `append()` and `pop()` operations, which are $O(1)$ in amortized time.
- The expensive move operation from **Stack 1** to **Stack 2** happens only when **Stack 2** is empty, leading to an **amortized** $O(1)$ dequeue operation.

Operation	**Two Stacks (Amortized)**	**Python List (Queue)**
Enqueue ($enqueue(x)$)	$O(1)$	$O(1)$
Dequeue ($dequeue()$)	Amortized $O(1)$, Worst-case $O(n)$	$O(n)$ (due to $pop(0)$ shifting)
Is Empty ($is\_empty()$)	$O(1)$	$O(1)$

5.3 Classic Problem: Round-robin Scheduler

In a **round-robin scheduling** system, each task gets a **fixed time slice (quantum)** to run. If a task requires more time, it is **paused** and placed **back into the queue** until it is fully executed.

- A list of tasks, each with a required execution time.

- A **fixed time slice (quantum)** for each task.
- If a task needs more than the allotted time, it will be **re-enqueued**.

Input:

```
tasks = [("Task1", 6), ("Task2", 3), ("Task3", 4)]
time_slice = 2
```

Expected Output:

```
Processing Task1 for 2 units
Processing Task2 for 2 units
Processing Task3 for 2 units
Processing Task1 for 2 units
Processing Task2 for 1 units (Completed)
Processing Task3 for 2 units (Completed)
Processing Task1 for 2 units (Completed)
All tasks completed!
```

```python
def scheduler(tasks, time_slice) -> None:
    queue = Queue()

    # Enqueue all tasks
    for task in tasks:
        queue.enqueue(task)

    while not queue.is_empty():
        task_name, time_required = queue.dequeue()

        if time_required > time_slice:
            print(f"Processing {task_name} for {time_slice} units")
            queue.enqueue((task_name, time_required - time_slice))  # Re-enqueue with remaining time
        else:
            print(f"Processing {task_name} for {time_required} units (Completed)")

    print("All tasks completed!")

# Test Case
tasks = [("Task1", 6), ("Task2", 3), ("Task3", 4)]
time_slice = 2
scheduler(tasks, time_slice)
```

5.4 Priority Queue

The **Priority Queue Abstract Data Type (ADT)** is similar to the standard Queue ADT, but each item is associated with a certain **priority**. Items are dequeued based on their priority rather than their insertion order. In cases where two items share the same priority, the item that was added first will be removed first, following the **First-In-First-Out (FIFO)** principle.

Priority Queue

- **Data:** A collection of items, each with an associated priority.
- **Operations:**
  - Determine whether the priority queue is empty.
  - Add an item along with a priority (enqueue).
  - Remove and return the highest-priority item (dequeue).

One important aspect of this ADT is how priorities are represented. In this section, priorities are represented as **integers**, where larger integers correspond to higher priorities.

```python
from typing import Any

class PriorityQueue:
```

```
    """A collection of items that are removed in priority order.

    When removing an item from the queue, the item with the highest priority
    is dequeued first.
    """

    def __init__(self) -> None:
        """Initialize a new and empty priority queue."""

    def is_empty(self) -> bool:
        """Return whether this priority queue contains no items."""

    def enqueue(self, priority: int, item: Any) -> None:
        """Add the given item with the given priority to this priority queue."""

    def dequeue(self) -> Any:
        """Remove and return the item with the highest priority.

        Raise an EmptyPriorityQueueError when the priority queue is empty.
        """

    def peek(self) -> Any:
        """Return the item at the front of the priority queue without removing it.

        Raise an EmptyPriorityQueueError if this priority queue is empty.
        """

class EmptyPriorityQueueError(Exception):
    """Exception raised when calling dequeue on an empty priority queue."""

    def __str__(self) -> str:
        """Return a string representation of this error."""
        return 'You called dequeue on an empty priority queue.'
```

Our implementation idea here is to use a private attribute that is a list of tuples, where each tuple is a `(priority, item)` pair. Our list will also be sorted with respect to priority (breaking ties by insertion order), so that the last element in the list is always the next item to be removed from the priority queue.

With this idea, three of the four `PriorityQueue` methods are straightforward to implement:

```
class PriorityQueue:
    """A queue of items that can be dequeued in priority order.

    When removing an item from the queue, the highest-priority item is the one
    that is removed.
    """
    # Private Instance Attributes:
    #   - _items: a list of the items in this priority queue
    _items: list[tuple[int, Any]]

    def __init__(self) -> None:
        """Initialize a new and empty priority queue."""
        self._items = []

    def is_empty(self) -> bool:
        """Return whether this priority queue contains no items."""
        return self._items == []

    def dequeue(self) -> Any:
        """Remove and return the item with the highest priority.

        Raise an EmptyPriorityQueueError when the priority queue is empty.
        """
        if self.is_empty():
```

```
            raise EmptyPriorityQueueError
        else:
            _priority, item = self._items.pop()
            return item

    def peek(self) -> Any:
        """Return the item at the front of the priority queue without removing it.

        Raise an EmptyPriorityQueueError if this priority queue is empty.
        """

        if self.is_empty():
            raise EmptyPriorityQueueError
        else:
            return self._items[-1]
```

But what about `PriorityQueue.enqueue`? An initial approach might be to first insert the new priority and item into the list, and then sort the list by priority. However, this is inefficient: we shouldn't need to re-sort the entire list if we start with a sorted list and are simply inserting one new item.

Insert an element in a sorted list – Binary search! But the question is, how to keep FIFO order for the items with the same priority?

```
def binary_search(lst: list, priority: int) -> int:
    """Return the index where `priority` should be inserted in `lst` using binary search.

    Preconditions:
        - `lst` is sorted in non-decreasing order of priority.
    """
    b, e = 0, len(lst)

    while b < e:
        m = (b + e) // 2
        if priority <= lst[m][0]:  # Find the first occurrence of `priority`
            e = m
        else:
            b = m + 1
    return b  # Correct index ensuring FIFO order
```

This ensures FIFO ordering by inserting new items at the first occurrence of priority. Because when the items have same priority, we update our left boundary, so we keep searching for the first occurrence from the left side!

```
from typing import Any


class PriorityQueue:
    """A queue of items that can be dequeued in priority order.

    When removing an item from the queue, the highest-priority item is the one
    that is removed.
    """
    # Private Instance Attributes:
    #   - _items: a list of the items in this priority queue
    _items: list[tuple[int, Any]]

    def __init__(self) -> None:
        """Initialize a new and empty priority queue."""
        self._items = []

    def is_empty(self) -> bool:
        """Return whether this priority queue contains no items."""
        return self._items == []

```

```
    def dequeue(self) -> Any:
        """Remove and return the item with the highest priority.

        Raise an EmptyPriorityQueueError when the priority queue is empty.
        """
        if self.is_empty():
            raise EmptyPriorityQueueError
        else:
            _priority, item = self._items.pop()
            return item

    def _binary_search(self, lst: list, priority: int) -> int:
        """Return the index where `priority` should be inserted in `lst` using binary search.

        Preconditions:
            - `lst` is sorted in non-decreasing order of priority.
        """
        b, e = 0, len(lst)

        while b < e:
            m = (b + e) // 2
            if priority <= lst[m][0]:  # Find the first occurrence of `priority`
                e = m
            else:
                b = m + 1
        return b  # Correct index ensuring FIFO order

    def enqueue(self, priority: int, item: Any) -> None:
        """Insert an item into the priority queue while maintaining sorted order and FIFO.

        Uses binary search to find the correct position.
        """
        index = self._binary_search(self._items, priority)  # Find correct insertion index
        self._items.insert(index, (priority, item))  # Insert at correct index

    def peek(self) -> Any:
        """Return the item at the front of the priority queue without removing it.

        Raise an EmptyPriorityQueueError if this priority queue is empty.
        """

        if self.is_empty():
            raise EmptyPriorityQueueError
        else:
            return self._items[-1]

class EmptyPriorityQueueError(Exception):
    """Exception raised when calling dequeue on an empty priority queue."""

    def __str__(self) -> str:
        """Return a string representation of this error."""
        return 'You called dequeue on an empty priority queue.'
```

To do a test:

```
pq = PriorityQueue()
pq.enqueue(2, 'taskA')
pq.enqueue(1, 'taskB')
pq.enqueue(2, 'taskC')  # Same priority as 'taskA'
pq.enqueue(2, 'taskD')  # Same priority as 'taskA' and 'taskC'
pq.enqueue(3, 'taskE')

while not pq.is_empty():
    print(pq.dequeue())
```

```
10
```

Later, we will use a different data structure, the Heap, to implement the Priority Queue (PQ) more efficiently. A list is not ideal for frequent insert operations because it requires a significant number of shift operations, which are highly expensive!

6 Linked List

We have learned that Python lists are an array-based implementation of the List Abstract Data Type (ADT). As a result, they have certain efficiency limitations: inserting and deleting elements in a Python list may require shifting multiple elements in memory. In the worst case, inserting or deleting at the beginning of a Python list takes $O(n)$ time, where n is the length of the list, since every element needs to be shifted. The reason Python list operations often require shifting elements back and forth is that the (memory addresses of) list elements are stored in contiguous locations in memory.

In this section, we will explore an alternative implementation of the List ADT designed to mitigate this efficiency issue. To achieve this, we will introduce a new data structure known as the linked list. The linked list does not enforce memory continuity. The question is, if we had a variable pointing to the first element of a list, how could we determine the locations of the remaining elements? This issue can be resolved by storing, along with each element, a reference to the next element in the list.

6.1 _Node Class

This combination of data—an element along with a reference to the next element—suggests the need for a new data type that encapsulates both pieces of information. We will call this data type a **node** and implement it in Python as follows. We use a preceding underscore in the class name to indicate that this class is private: it should not be accessed directly by client code but is intended for use within the "main" class described later in this section.

```
class _Node:
    """A node in a linked list.

    Instance Attributes:
      - item: The data stored in this node.
      - next: The next node in the list, if any.
    """

    def __init__(self, item, next = None):
        """Initialize a node with a given item and an optional reference to the next node."""
        self.item = item
        self.next = next  # Defaults to None if no next node is provided

    def __str__(self):
        """print the node info"""
        return f"Node: item is {self.item}, next is {None if self.next == None else self.__class__.__name__}"
```

An instance of `_Node` represents a single element in a list. To represent a list of n elements, we need n `_Node` instances. The references in their `next` attributes link these nodes together in sequence, even though they are not stored in consecutive memory locations. These references, or "links," are what give linked lists their name.

6.2 LinkedList Class

The second class in our linked list implementation is `LinkedList`, which represents the list itself. This class serves as the interface for client code. The `LinkedList` class has a private attribute, `head`, which refers to the first node in the list. This attribute determines the starting point of the list. Our initial version of this class includes a simple initializer that always creates an empty linked list:

```
class LinkedList:
    """A linked list implementation of the List ADT, hiding the internal node structure."""

    def __init__(self) -> None:
        """Initialize an empty linked list."""
        self.head = None
```

To perform meaningful operations on linked lists, we need to construct lists of arbitrary length. For now, we will directly manipulate the private attributes to build a linked list.

```
llst = LinkedList()  # Create an empty linked list
node1 = _Node(111)   # Create a new node with item 111
```

```
node2 = _Node(211)    # Create a new node with item 211
node3 = _Node(241)  # Create a new node with item 241

print("node1.item: ", node1.item)
print("node1.next: ", node1.next)

# Manually setting links
node1.next = node2
node2.next = node3

print("node1.next: ",node1.next)
print("node1.next.item: ",node1.next.item)
print("node1.next.next: ", node1.next.next)
print("node1.next.next.item: ", node1.next.next.item)

# Setting the linked list's head node
llst.head = node1
print("llst.head.item: ", llst.head.item)
print("llst.head.next.item: ", llst.head.next.item)
print("llst.head.next.next.item: ", llst.head.next.next.item)
```

One of the most frequent mistakes when learning linked lists is confusing a `_Node` object with the item it stores. In the example above, there is a significant difference between `node3` and `node3.item`: - `node3` is a `_Node` object that contains the value `241`. - `node3.item` is the integer `241` itself.

When working with linked lists, it is essential to distinguish between operating on nodes versus operating on items. Maintaining this distinction is key to successfully implementing and debugging linked list operations.

6.3 Traversing Linked List

In the previous section, we created a `LinkedList` object containing three elements and used the following expressions to access each item sequentially:

```
>>> llst.head.item
111
>>> llst.head.next.item
211
>>> llst.head.next.next.item
241
```

However, this approach is inefficient and cumbersome. In this section, we explore how to **traverse** a linked list—writing code that systematically visits each element, regardless of the list's length. Since linked lists do not support indexing, we instead use a loop variable to reference the **current `_Node` object**:

```
curr = llst.head              # 1. Initialize curr to the head node.
while curr is not None:       # 2. Stop when curr reaches the end (None).
    ... curr.item ...         # 3. Process the current element, curr.item.
    curr = curr.next          # 4. Move to the next node.
```

The following method converts a linked list into a Python list:

```
class LinkedList:
    def to_list(self) -> list:
        """Return a Python list containing the elements of this linked list."""
        items_so_far = []
        curr = self.head
        while curr is not None:
            items_so_far.append(curr.item)
            curr = curr.next
        return items_so_far
```

Calling `linky.to_list()` on our example linked list returns `[111, 211, 241]`.

6.4 Looping with An Index

Now, let's consider a slightly different problem: accessing an element in a linked list at a specific index, similar to using `my_list[3]` in Python. The method we will implement for this functionality is as follows:

```
class LinkedList:
    def __getitem__(self, i: int) -> Any:
        """Return the item stored at index i in this linked list.

        Raise an IndexError if index i is out of bounds.

        Preconditions:
            - i >= 0
        """
```

We begin by utilizing our linked list traversal pattern, with an additional variable to track the current index:

```
class LinkedList:
    def __getitem__(self, i: int) -> Any:
        """Return the item stored at index i in this linked list.

        Raise an IndexError if index i is out of bounds.

        Preconditions:
            - i >= 0
        """
        curr = self.head
        curr_index = 0

        while curr is not None:
            ... curr.item ...

            curr = curr.next
            curr_index = curr_index + 1
```

By updating both `curr` and `curr_index` simultaneously, we maintain the following loop invariant:

curr refers to the node at index curr_index in the linked list.

the `__getitem__` special method is also called automatically by the Python interpreter when we use **`square bracket notation`** for sequence indexing/dictionary key lookup. That means that we can use the following syntax to index into our LinkedList object, just as we would a Python list:

```
llst[0]
>>>111
```

6.4.1 Early Stop Method

One way to complete this method is by using an early return statement, exiting the loop as soon as we reach the desired index:

```
from typing import Any

class LinkedList:
    def __getitem__(self, i: int) -> Any:
        """..."""
        curr = self.head
        curr_index = 0

```

```
        while curr is not None:
            if curr_index == i:
                return curr.item

            curr = curr.next
            curr_index = curr_index + 1

        # If the loop ends and no item has been returned,
        # the given index is out of bounds.
        raise IndexError
```

This approach follows the early return loop pattern, making the code simple and easy to understand.

6.4.2 Compound Loop Condition Method

An alternative approach is to modify the while loop condition so that it stops when either the end of the list is reached or the correct index is found. Compound loop conditions (i.e., conditions with multiple logical expressions) can be tricky to reason about. A helpful strategy is to first define the stopping condition and then negate it to determine the loop condition.

In this case, the stopping condition is:

1. `curr` is `None` (end of the list), or
2. `curr_index == i` (correct index found).

With this in mind, the while loop condition is written as the negation of the stopping condition:

```
from typing import Any

class LinkedList:
    def __getitem__(self, i: int) -> Any:
        """... """
        curr = self.head
        curr_index = 0

        while not (curr is None or curr_index == i):
            curr = curr.next
            curr_index = curr_index + 1

        if curr is None:
            # Index is out of bounds
            raise IndexError
        else:
            # curr_index == i, so curr is the node at index i
            return curr.item
```

The second implementation is more complex than the first, but it is important to understand both methods.

6.5 Mutating Linked List

Previously, we constructed a linked list by creating individual nodes separately and manually linking them together. However, since `_Node` should be a private class within LinkedList, it must remain hidden from the user. If `_Node` is encapsulated within LinkedList, how can we create and modify a linked list instance?

6.5.1 `append()` Method

Since a `LinkedList` object only contains a reference to the first node, appending an item requires us to locate the current last node before adding the new node at the end. We begin by creating a new node for the item and employing a traversal pattern to find the last node. To determine when we reach the last node, we recall that a `_Node` object's `next` attribute is `None` when it is the final node.

```
from typing import Any
class LinkedList:
    def append(self, item: Any) -> None:
        """Add the given item to the end of this linked list."""
        new_node = _Node(item)  # The node to insert into the linked list

        curr = self.head
        while curr.next is not None:
            curr = curr.next
```

A potential issue arises if `curr` is initially `None`, meaning the list is empty. If this occurs, the loop condition will attempt to access `curr.next`, resulting in an error. Instead of addressing this immediately, we insert a `TODO` comment and an `if` statement:

```
class LinkedList:
    def append(self, item: Any) -> None:
        """..."""
        new_node = _Node(item)  # The node to insert into the linked list

        if self.head is None:
            ...  # TODO: Handle this case
        else:
            curr = self.head
            while curr.next is not None:
                curr = curr.next
```

After reaching the last node, we update its `next` attribute to point to the new node, effectively appending it to the linked list.

```
class LinkedList:
    def append(self, item: Any) -> None:
        """..."""
        new_node = _Node(item)  # The node to insert into the linked list

        if self.head is None:
            ...  # TODO: Handle this case
        else:
            curr = self.head
            while curr.next is not None:
                curr = curr.next

            # After the loop, curr is the last node in the LinkedList.
            curr.next = new_node
```

Now, we handle the case where the list is initially empty. The only possible way this occurs is when `self.head` is `None`. In this case, the new node becomes the first node in the linked list.

```
class LinkedList:
    def append(self, item: Any) -> None:
        """..."""
        new_node = _Node(item)

        if self.head is None:
            self.head = new_node
        else:
            curr = self.head
            while curr.next is not None:
                curr = curr.next

            # After the loop, curr is the last node in the LinkedList.
```

```
            assert curr is not None and curr.next is None
            curr.next = new_node
```

With the `append` method implemented, we can now avoid manually modifying linked list attributes. Instead, we enhance the initializer to accept an iterable collection of values (works with `for` loops), appending them one by one.

```
from typing import Iterable

class LinkedList:
    def __init__(self, items) -> None:
        """Initialize a new linked list containing the given items."""
        self.head = None
        for item in items:
            self.append(item)
```

Although this implementation is functionally correct, it is inefficient. Now, we can reorganize our code as below:

```
from typing import Any, Iterable


class LinkedList:
    """A linked list implementation of the List ADT, hiding the internal node structure."""

    class _Node:
        """A node in a linked list.

        Instance Attributes:
          - item: The data stored in this node.
          - next: The next node in the list, if any.
        """

        def __init__(self, item, next=None):
            """Initialize a node with a given item and an optional reference to the next node."""
            self.item = item
            self.next = next  # Defaults to None if no next node is provided

        def __str__(self):
            return f"Node: item is {self.item}, next is {None if self.next == None else self.__class__.__name__}"

    def __init__(self, items: Iterable) -> None:
        """Initialize a new linked list containing the given items."""
        self.head = None
        for item in items:
            self.append(item)

    def to_list(self) -> list:
        """Return a Python list containing the elements of this linked list."""
        items_so_far = []
        curr = self.head
        while curr is not None:
            items_so_far.append(curr.item)
            curr = curr.next
        return items_so_far

    def __getitem__(self, i: int) -> Any:
        """Return the item stored at index i in this linked list.

        Raise an IndexError if index i is out of bounds.

        Preconditions:
            - i >= 0
```

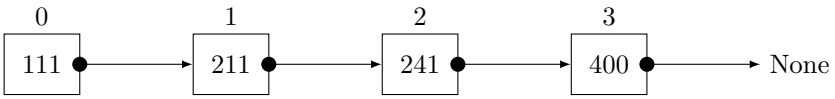
```
        """
        curr = self.head
        curr_index = 0

        while not (curr is None or curr_index == i):
            curr = curr.next
            curr_index = curr_index + 1

        if curr is None:
            # Index is out of bounds
            raise IndexError
        else:
            # curr_index == i, so curr is the node at index i
            return curr.item

    def append(self, item: Any) -> None:
        """Add the given item to the end of this linked list."""
        new_node = self._Node(item)

        if self.head is None:
            self.head = new_node
        else:
            curr = self.head
            while curr.next is not None:
                curr = curr.next

            # After the loop, curr is the last node in the LinkedList.
            assert curr is not None and curr.next is None
            curr.next = new_node
```

6.5.2 `insert()` Method

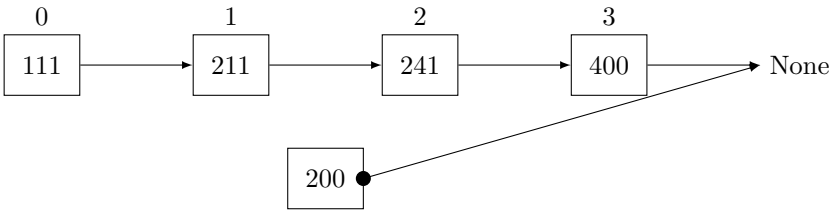
In the previous section, we explored how to implement a basic append method, which adds an item to the end of a linked list. Now, let's consider a more general insertion method that allows users to specify the exact index where the new item should be inserted.

```
class LinkedList:
    def insert(self, i: int, item: Any) -> None:
        """Insert the given item at index i in this linked list.

        Raise IndexError if i exceeds the length of the list.

        If i *equals* the length of the list, the item is added at
        the end, equivalent to LinkedList.append.

        Preconditions:
            - i >= 0
        """
```

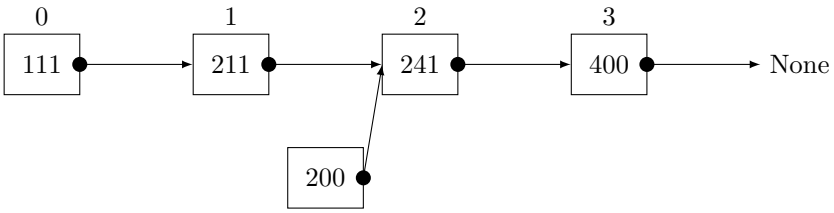
Before diving into the implementation, let's first examine some diagrams to illustrate the process.



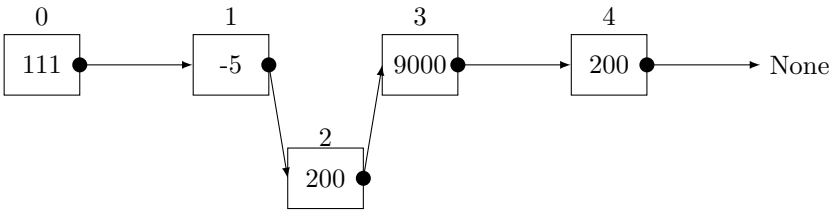
We want to insert `_Node` 200 at index 2 in this list. Like LinkedList.append, we can start by creating a new node, which by default links to None:



To insert the new node into the linked list, we need to modify some links. Because we want the new node to be at index 2, we need to modify the new node to point to next node of the current node:



After that, we need to let the current node point to the new node. Here is our final diagram:



To insert a node at position i, we need to access the node at position $i - 1$. We can achieve this using the same approach as the `LinkedList.__getitem__` method studied earlier. Here, we utilize the compound loop condition approach instead of the early return approach.

```
class LinkedList:
    def insert(self, i: int, item: Any) -> None:
        """..."""
        new_node = _Node(item)
        curr = self.head
        curr_index = 0

        while not (curr is None or curr_index == i - 1):
            curr = curr.next
            curr_index = curr_index + 1

        # After the loop, either we've reached the end of the list
        # or curr is the (i - 1)-th node.

        if curr is None:
            ...
        else:  # curr_index == i - 1
            ...
```

If `curr` is `None` when the loop terminates, it means the list does not contain a node at position $i - 1$, making i out of bounds. In this case, we should raise an `IndexError`.

On the other hand, if `curr` is not `None`, we have reached the desired position and can insert the new node. The insertion follows the same approach as `LinkedList.append`, except we must also update the `next` attribute of the new node.

```
class LinkedList:
    def insert(self, i: int, item: Any) -> None:
        """..."""
        new_node = _Node(item)
        curr = self.head
        curr_index = 0

```

```
        while not (curr is None or curr_index == i - 1):
            curr = curr.next
            curr_index = curr_index + 1

        # After the loop, either we've reached the end of the list
        # or curr is the (i - 1)-th node.

        if curr is None:
            # i - 1 is out of bounds. The item cannot be inserted.
            raise IndexError
        else:  # curr_index == i - 1
            # i - 1 is valid. Insert the new item.
            new_node.next = curr.next
            curr.next = new_node
```

When writing mutating methods for linked lists, we frequently update the `next` attributes of individual nodes to insert or remove elements. The **order** in which these links are updated is crucial, as an incorrect sequence can lead to unexpected behavior.

For example, the following incorrect update order results in an unintended reference cycle:

```
curr.next = new_node
new_node.next = curr.next  # Incorrect!
```

On the second line, `curr.next` has already been modified, and its original value is lost. This effectively sets `new_node.next` to itself, which is clearly incorrect. A safer approach is to use parallel assignment, which ensures that the two assignments are executed in a single step:

```
curr.next, new_node.next = new_node, curr.next
```

Our `LinkedList.insert` implementation has one issue: what if $i = 0$? In this case, iterating to position $i - 1$ does not make sense! Similar to `LinkedList.append`, this special case must be handled separately by updating `self.head`.

```
class LinkedList:
    def insert(self, i: int, item: Any) -> None:
        """..."""
        new_node = self._Node(item)

        if i == 0:
            # Insert the new node at the beginning of the list.
            self.head, new_node.next = new_node, self.head
        else:
            curr = self.head
            curr_index = 0

            while not (curr is None or curr_index == i - 1):
                curr = curr.next
                curr_index = curr_index + 1

            # After the loop, either we've reached the end of the list
            # or curr is the (i - 1)-th node.

            if curr is None:
                # i - 1 is out of bounds. The item cannot be inserted.
                raise IndexError
            else:  # curr_index == i - 1
                # i - 1 is valid. Insert the new item.
                curr.next, new_node.next = new_node, curr.next
```

6.6 List vs. LinkedList

Let us compare the running time analysis of our `LinkedList` implementation against Python's built-in `list.insert` method. Assuming $0 \leq i < n$, we have $\min(n, i) = i$, which means the running time of `LinkedList.insert` is $\Theta(i)$.

This result is subtle: notice how n (our traditional variable for "input size") no longer appears in the Theta expression! While $\Theta(\min(n, i))$ represents the most general expression without assuming any relationship between the list length and index i, our simplified expression $\Theta(i)$ requires the additional assumption that $i < n$.

In essence, when i is small relative to the list size, the algorithm's running time depends only on the index position, not the list length. The most striking example is when $i = 0$ (insertion at the front), which takes constant time $\Theta(1)$, independent of list length.

The same analysis applies to `LinkedList.__getitem__` and `LinkedList.pop`, as they both require traversing to the node at index i. We can summarize the running times as follows:

Operation (assuming $0 \leq i < n$)	`list` Runtime	`LinkedList` Runtime
Indexing (`lst[i]`)	$\Theta(1)$	$\Theta(i)$
Insert at index i	$\Theta(n - i)$	$\Theta(i)$
Remove at index i	$\Theta(n - i)$	$\Theta(i)$

For indexing operations, linked lists exhibit inferior performance: access time is proportional to the index position rather than constant. This highlights the primary advantage of array-based implementations, which benefit from contiguous memory storage. However, for insertion and deletion, linked lists show precisely opposite performance characteristics! Front insertion in a linked list takes $\Theta(1)$ time, while back insertion requires $\Theta(n)$ time.

While this performance tradeoff might seem problematic, it actually exemplifies a common pattern in computer science: different implementations of the same interface often excel in some operations while performing worse in others. The choice between implementations ultimately falls to the programmer, who must prioritize the efficiency of specific operations based on their application's needs:

- Choose `list` when constant-time indexing is crucial and modifications primarily occur at the list's end
- Choose `LinkedList` when frequent front insertions/deletions are required and random access is minimal

6.7 Revisit `append()` Method

Can we do further modifications to make our `append` method more efficient? Yes, we can optimize the `append` method using a tail node.

```
class LinkedList:
    def __init__(self, items: Iterable) -> None:
        """Initialize a new linked list containing the given items in O(n) time."""
        self.head = None
        self.tail = None  # Maintain a tail pointer for efficient appends
        for item in items:
            self.append(item)

    def append(self, item: Any) -> None:
        """Append an item to the end of the linked list in O(1) time."""
        new_node = _Node(item)

        if self.head is None:
            # If the list is empty, set both head and tail to the new node
            self.head = new_node
            self.tail = new_node
        else:
            # Directly update the tail's next pointer instead of looping
            self.tail.next, self.tail = new_node, new_node
```

Of course, the `insert()` function is needed to be modified, because we probably will insert a new node to the end of the list, and we need to update tail pointer corrospondingly:

```python
class LinkedList:
    def insert(self, i: int, item: Any) -> None:
        """Insert item at index i."""
        new_node = self._Node(item)

        # Case 1: Insert at the beginning
        if i == 0:
            new_node.next = self.head
            self.head = new_node

            # If list was empty, update tail as well
            if self.tail is None:
                self.tail = new_node
            return

        # Case 2: Insert at index > 0
        curr = self.head
        curr_index = 0

        while curr is not None and curr_index < i - 1:
            curr = curr.next
            curr_index += 1

        if curr is None:
            raise IndexError("Index out of bounds")

        # Insert after curr
        new_node.next = curr.next
        curr.next = new_node

        # If inserted at the end, update tail
        if new_node.next is None:
            self.tail = new_node
```

6.8 pop() Method

pop() function should be able to remove the item at a specified index position:

- By default, return and remove the last node.
- If the list is empty, you know for sure that index is out of bounds.
- Else if i is 0, remove the first node and return its item.
- Else iterate to the (i-1)-th node and update links to remove the node at position index. But don't forget to return the item!

```python
    def pop(self, index: int = None) -> Any:
        """
        Remove and return an item from the linked list.

        Args:
            index: The index of the item to remove. If None (default),
                   removes the last item.

        Returns:
            The removed item.

        Raises:
            IndexError: If the linked list is empty or index is out of range.
        """
        if self.head is None:
            raise IndexError("Cannot pop from an empty linked list")

        # Case 1: Remove the last item (default behavior)
```

```
        if index is None:
            # If there's only one node
            if self.head == self.tail:
                value = self.head.item
                self.head = None
                self.tail = None
                return value

            # Find the node before the tail
            current = self.head
            while current.next != self.tail:
                current = current.next

            # Store the value to return
            value = self.tail.item

            # Update the tail
            self.tail = current
            current.next = None

            return value

        # Case 2: Remove from the beginning (index=0)
        if index == 0:
            value = self.head.item
            self.head = self.head.next

            # If we removed the last node, update the tail
            if self.head is None:
                self.tail = None

            return value

        # Case 3: Remove from a specific index
        current = self.head
        position = 0

        while current.next and position < index - 1:
            current = current.next
            position += 1

        # Check if index is out of range
        if current.next is None:
            raise IndexError(f"Index {index} is out of range")

        # Store the value to return
        value = current.next.item

        # Special case: removing the tail
        if current.next == self.tail:
            self.tail = current

        # Remove the node
        current.next = current.next.next
        return value
```

7 Tree Structure

While the List abstract data type is extremely common and useful, not all data has a natural linear order. Family trees, corporate organization charts, and file storage systems on computers all follow a **hierarchical structure** where each entity is linked to multiple entities below it.

In computer science, we use a **tree data structure** to represent this type of hierarchical data. Trees are a recursive data structure, with the following recursive definition: A tree is either:

- empty, or
- has a root value connected to any number of other trees, called the subtrees of the tree.

One noteworthy aspect of our definition is that **subtrees are allowed to be empty**. We predominantly work with non-empty trees when representing real-world data, but there are cases when allowing empty subtrees is useful or necessary.

We generally draw the root at the top of the tree; the rest of the tree consists of subtrees that are attached to the root.

- **Size**: The size of a tree is the number of values in the tree.
- **Leaf**: A leaf is a value with no subtrees.
- **Internal value**: The opposite of a leaf is an internal value, which is a value that has at least one subtree.
- **Height**: The height of a tree is the length of the longest path from its root to one of its leaves, counting the number of values on the path.
- **Children**: The children of a value are all values directly connected underneath that value.
- **Descendants**: The descendants of a value are its children, the children of its children, etc. This can be defined recursively as "the descendants of a value are its children, and the descendants of its children."
- **Parent**: The parent of a value is the value immediately above and connected to it; each value in a tree has exactly one parent, except the root, which has no parent.
- **Ancestors**: The ancestors of a value are its parent, the parent of its parent, etc. This too can be defined recursively: "the ancestors of a value are its parent, and the ancestors of its parent."

Note: sometimes, it will be convenient to say that descendants/ancestors of a value include the value itself; we'll make it explicit whether to include the value or not when it comes up. However, a value is never a child or parent of itself.

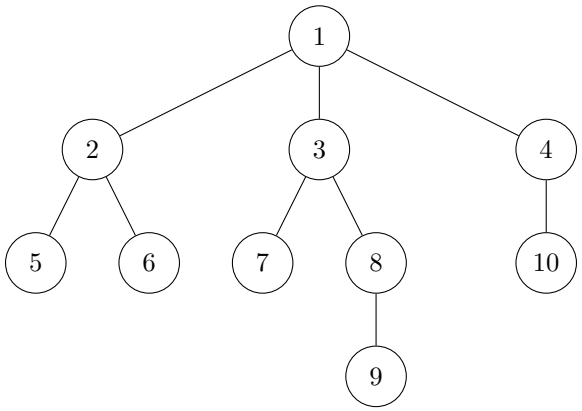


Figure 1: Tree structure Example

7.1 A Tree Implementation

Here is a simple implementation of a tree in Python. As with linked lists, we'll start with a very simple implementation and then develop additional methods:

```
class Tree:
    """A recursive tree data structure.
    CopyRepresentation Invariants:
        - self._root is not None or self._subtrees == []
    """
```

```
    # Private Instance Attributes:
    #   - _root:
    #       The item stored at this tree's root, or None if the tree is empty.
    #   - _subtrees:
    #       The list of subtrees of this tree. This attribute is empty when
    #       self.root is None (representing an empty tree). However, this attribute
    #       may be empty when self.root is not None, which represents a tree consisting
    #       of just one item.

    def __init__(self, root, subtrees):
        """Initialize a new Tree with the given root value and subtrees.

        If root is None, the tree is empty.

        Preconditions:
            - root is not none or subtrees == []
        """
        self.root = root
        self.subtrees = subtrees

    def is_empty(self) -> bool:
        """Return whether this tree is empty.
        """
        return self.root is None
```

Our `Tree` initializer creates either an empty tree (when `root` is `None`), or a tree with a root value and the given subtrees. Note that it is possible for `root` to not be `None`, but `subtrees` to still be empty: this represents a tree with a single root value, and no subtrees. The empty tree and single value cases are generally the base cases when writing recursive code that operates on trees.

7.2 Why Do We Define Linkedlist Interatively But Tree Recursively?

We actually can define linked lists recursively, and in many theoretical discussions and functional programming languages (You will see the example in CSC242), linked lists are indeed defined recursively. A recursive definition of a linked list would be: A linked list is either:

Empty, or A value and a reference to another linked list (the rest of the list)

This recursive definition is conceptually similar to the tree definition. However, there are a few key reasons why we tend to think of and implement linked lists differently than trees in practice:

- **Simpler structure**: Linked lists have a linear structure with each node pointing to at most one other node. This simplicity allows us to conceptualize them iteratively more easily than trees. We can simply "walk" from one node to the next, and also in our discussion, it belongs to List ADT.
- **Implementation efficiency**: Iterative implementations of linked list operations are often more efficient and **consume less stack space than recursive ones**, especially for long lists.
- **Practical Usage**: The iterative model of a linked list (moving from head to tail) often matches better with how we use linked lists in algorithms and data processing.

In contrast, trees inherently have a branching structure where each node can have multiple children, making recursion a more natural fit for both conceptualization and implementation. **The recursive approach handles the variable number of branches elegantly.**

7.3 Tree Recursion

Understanding the relationship between a tree and its subtrees—that is, its recursive structure—allows us to write elegant recursive code for processing trees.

Let's consider a first example: calculating the size of a tree. We can approach this by following the recursive definition of a tree, which is either empty or a root connected to a list of subtrees. Let T be a tree, and let $size$ be a function mapping a tree to its size.

- If T is empty, then its size is 0: we can write $size(T) = 0$.

- If T is non-empty, then it consists of a root value and collection of subtrees $T_0, T_1, \ldots, T_n - 1$ (for some $n > 0$). In this case, the size of T is the sum of the sizes of its subtrees, plus 1 for the root: $size(T) = 1 + size(T_0) + size(T_1) + \ldots + size(T_n - 1)$

We can combine these observations to write a recursive mathematical definition for our *size* function:

$$size(T) = \begin{cases} 0 & \text{if } T \text{ is empty} \\ 1 + \sum_{i=0}^{n-1} size(T_i) & \text{otherwise, where } T_0, \ldots, T_{n-1} \text{ are the subtrees of } T \end{cases} \quad (1)$$

This definition translates naturally into a recursive Python method `Tree.__len__`:

```
class Tree:
    def __len__(self) -> int:
        """
        Return the number of items contained in this tree.
        """
        if self.is_empty():
            return 0
        else:
            return 1 + sum(subtree.__len__() for subtree in self.subtrees)
```

The base case is when `self.is_empty()`, and the recursive step occurs when the tree is non-empty, requiring recursion on each subtree. If you do not like comprehension method, here's how we could have written our `else` branch using a `for` loop:

```
else:
    size_so_far = 1
    for subtree in self.subtrees:
        size_so_far += subtree.__len__()
    return size_so_far
```

Once you implement `__len__()`, you are able to directly use built-in `len()` method to produce the size of the tree object:

```
tree0 = Tree(0,[Tree(1,[]),Tree(2,[]),Tree(3,[])])
print(len(tree0)) # output is 4
```

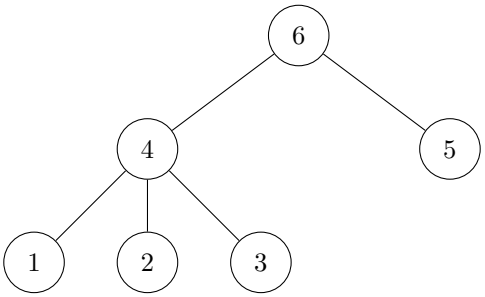
7.4 Tree Structure Visualization

Because the elements of a list have a natural order, lists are straightforward to traverse. With trees, there is a non-linear ordering on the elements. How might we write a `Tree.__str__` method?

Here's an approach: start with the value of the root, then recursively add the `__str__` for each of the subtrees. That's easy to implement. The base case is when the tree is empty, and in this case the method returns an empty string.

```
class Tree:
    def __str__(self):
        """
        Return a string representation of this tree.
        """
        if self.is_empty():
            return ''
        else:
            # We use newlines ('\n') to separate the different values.
            str_so_far = f'{self.root}\n'
            for subtree in self.subtrees:
                str_so_far += subtree.__str__()  # equivalent to str(subtree)
            return str_so_far
```

Consider what happens when we call this method on the following tree structure:




```
t1 = Tree(1, [])
t2 = Tree(2, [])
t3 = Tree(3, [])
t4 = Tree(4, [t1, t2, t3])
t5 = Tree(5, [])
t6 = Tree(6, [t4, t5])
print(t6)  # This automatically calls t6.__str__ and prints the result
6
4
1
2
3
5
```

We know that 6 is the root of the tree, but it's ambiguous how many children it has. While the items in the tree are correctly included, we lose the structure of the tree itself.

Drawing inspiration from how file explorers display folder structures, we can use `indentation` to differentiate between the different levels of a tree. For our example tree, we want `__str__` to produce:

```
6
    4
        1
        2
        3
    5
```

We want `Tree.__str__` to return a string with 0 indents before the root value, 1 indent before its children's values, 2 indents before their children's values, and so on. We need the recursive calls to return strings with more indentation the deeper down in the tree they are. We'll pass an extra parameter for the depth of the current tree, which will be used to add the corresponding number of indents before each value. We can define a helper method that has this extra parameter:

```
class Tree:
    def _str_indented(self, depth=0):
        """
        Return an indented string representation of this tree.
        The indentation level is specified by the <depth> parameter.
        """

        if self.is_empty():
            return ''
        else:
            str_so_far = '  ' * depth + f'{self.root}\n'
            for subtree in self.subtrees:
                # Note that the 'depth' argument to the recursive call is modified.
                str_so_far += subtree._str_indented(depth + 1)
            return str_so_far

    def __str__(self):
        """
```

```
        Return a string representation of this tree.
        """
        return self._str_indented()
```

7.5 Traversal Orders

When processing trees, the order in which we visit nodes is crucial for many algorithms. There are three fundamental ways to traverse a tree systematically: **preorder, inorder, and postorder traversals.** For general trees (non-binary), a true inorder traversal is not well-defined. So now, we just discuss preorder and postorder traversals.

7.5.1 Preorder Traversal

In a preorder traversal, we:

1. Visit the root node first
2. Recursively traverse each subtree from left to right. (By convention, we think of the subtrees list as being ordered from left to right.)

This traversal is useful when we need to explore nodes before their children, such as when creating a copy of a tree or evaluating a prefix expression. The implementation of `_str_indented()` is using preorder traversal because everytime we access the root value first. We can further define a function `preorder_traversal()` to return not only the string representation, but also a list of all items in this tree using preorder traversal..

```
class Tree:
    def preorder_traversal(self, depth=0):
        """
        Returns:
            a list of all items in this tree using preorder traversal
            a string representation using preorder traversal
        """
        if self.is_empty():
            return [], ''
        else:
            lst_so_far, str_so_far = [self.root], '  ' * depth + f'{self.root}\n'
            for subtree in self.subtrees:
                # Call preorder_traversal once per subtree and unpack the results
                subtree_lst, subtree_str = subtree.preorder_traversal(depth + 1)
                lst_so_far.extend(subtree_lst)
                str_so_far += subtree_str
            return lst_so_far, str_so_far
```

7.5.2 Postorder Traversal

In a postorder traversal, we:

1. Recursively traverse each subtree from left to right. (By convention, we think of the subtrees list as being ordered from left to right.)
2. Visit the root node last

This traversal is particularly useful when we need to process all children before their parent, such as when deleting a tree or evaluating a postfix expression.

```
class Tree:
    def postorder_traversal(self, depth=0):
        """
        Returns:
            a list of all items in this tree using postorder traversal
            a string representation using postorder traversal
        """

        if self.is_empty():
            return [], ''
```

```
        else:
            lst_so_far, str_so_far = [], ''
            for subtree in self.subtrees:
                # Changed preorder_traversal to postorder_traversal for correct traversal order
                subtree_lst, subtree_str = subtree.postorder_traversal(depth + 1)
                lst_so_far.extend(subtree_lst)
                str_so_far += subtree_str

            # Add root after processing all subtrees (postorder)
            lst_so_far.append(self.root)
            str_so_far += '  ' * depth + f'{self.root}\n'

            return lst_so_far, str_so_far
```

7.6 Mutating Trees

There are two fundamental mutating operations for trees: insertion and deletion.

7.6.1 Value-Based Deletion

Our goal is to implement the following method:

```
class Tree:
    def remove(self, item):
    """
    Delete one occurrence of the given item from this tree.
    Do nothing if the item is not in this tree.

    Return whether the given item was deleted successfully or not.
    """
```

For this method, we'll need to consider two cases: empty vs. non-empty tree. The base case when the tree is empty is straightforward:

```
class Tree:
    def remove(self, item):
        """..."""
        if self.is_empty():
            return False            # The item isn't in the tree
```

In the recursive step, we first check whether the item is equal to the root. If it is, we only need to remove the root; if not, we recurse on the subtrees to continue searching for the item:

```
class Tree:
    def remove(self, item):
        """..."""
        if self.is_empty():
            return False
        elif self.root == item:
            self._delete_root()   # Delete the root
            return True
        else:
            for subtree in self.subtrees:
                subtree.remove(item)
```

Deleting the root is challenging, so we'll defer that by writing a call to a helper method (`Tree._delete_root`) that we'll implement later. The inner `else` branch has two problems:

1. It doesn't return anything, violating the method's type contract.

2. If one recursive call successfully finds and deletes the item, no further subtrees should be modified or recursed on.

The solution to both problems is to use the return values of the recursive calls to determine whether the item was deleted, which tells us whether to continue to the next subtree:

```
class Tree:
    def remove(self, item):
        """..."""
        if self.is_empty():
            return False
        elif self.root == item:
            self._delete_root()   # Delete the root
            return True
        else:
            for subtree in self._subtrees:
                if subtree.remove(item):
                    return True
            return False
```

The final step is to implement `Tree._delete_root`:

```
class Tree:
    def _delete_root(self):
    """
    Remove the root item of this tree.
    Preconditions:
            - not self.is_empty()
    """
        self.root = None
```

This has a significant problem: if the tree has subtrees, setting `root` to `None` violates our representation invariant that states if `self.root` is `None`, then `self.subtrees == []`. We should only assign `self.root` to `None` when there are no subtrees:

```
class Tree:
    def _delete_root(self):
        """..."""
        if self.subtrees == []:
            self.root = None
        else:
            ...
```

There are many ways to handle the non-empty subtrees case. One approach is to select the **rightmost subtree** and **promote** its root and subtrees by moving them up a level in the tree:

```
class Tree:
    def _delete_root(self):
        """..."""
        if self.subtrees == []:
            self.root = None
        else:
            # Get the last subtree in this tree.
            chosen_subtree = self.subtrees.pop()
            self.root = chosen_subtree.root
            self.subtrees.extend(chosen_subtree.subtrees)
```

This implementation is technically correct, satisfying the function specification without violating any representation invariants. However, it might not be entirely satisfying because it significantly alters the structure of the original tree just to delete a single element.

We still have an issue to address. In our current implementation of `Tree.remove`, if we delete an item that is a leaf of the tree, we'll successfully delete that item, but we'll end up with an empty tree in the parent's subtrees list. For example:

```
t = Tree(10, [Tree(1, []), Tree(2, []), Tree(3, [])])  # A tree with leaves 1, 2, and 3
t.remove(1)
t.remove(2)
t.remove(3)

print(t.subtrees)
>>>[<__main__.Tree object at 0x105282c10>, <__main__.Tree object at 0x105282bb0>, <__main__.Tree object at 0x105282b50>]

print([subtree.is_empty() for subtree in t._subtrees])
>>>[True, True, True]
```

Our tree `t` now has three empty subtrees. These empty subtrees waste memory and slow down subtree iteration and counterintuitive.

If we detect that we've deleted a leaf, we should remove the now-empty subtree from its parent's subtree list. This requires a small modification to `Tree.remove`:

```
class Tree:
    def remove(self, item):
        """..."""
        if self.is_empty():
            return False
        elif self.root == item:
            self._delete_root()   # Delete the root
            return True
        else:
            for subtree in self.subtrees:
                deleted = subtree.remove(item)
                if deleted and subtree.is_empty():
                    # Note that mutating a list while looping through it is
                    # EXTREMELY DANGEROUS!
                    # We are only doing it because we return immediately
                    # afterwards, and so no more loop iterations occur.
                    self.subtrees.remove(subtree)
                    return True
                elif deleted:
                    return True
            return False
```

Generally, removing an object from a list while iterating through it is extremely dangerous, as it interferes with the loop's iteration. **We avoid this problem because immediately after removing the subtree, we stop the method by returning `True`.**

7.6.2 Value-Based Insertion

Unlike deletion, insertion requires us to decide where to place the new item, as there are multiple valid positions in a tree where an item could be inserted. Let's implement an insertion method that takes a parent value and inserts the new item as a child of the first occurrence of that parent:

```
class Tree:
    def insert(self, parent, item):
        """
        Insert the given item as a child of the first occurrence of parent.
        If the tree is empty, insert the item as the root.

        Return True if successful.
        """
        if self.is_empty():
```

```
            # For an empty tree, set the item as the root
            self._root = item
            return True

        if self.root == parent:
            # Found the parent, insert as its child
            self.subtrees.append(Tree(item, []))
            return True

        # Recursively search for the parent in the subtrees
        for subtree in self.subtrees:
            if subtree.insert(parent, item):  # Note: changed from insert_under to match method name
                return True

        # Parent not found in this tree
        return False
```

8 Binary Search Trees

Binary search tree (BST), a specialized tree structure for efficiently storing and retrieving data. A tree where each node has two (possibly empty) subtrees, designated as left and right subtrees. A node in a binary tree satisfies this property when its value is:

- Greater than or equal to all values in its left subtree
- Less than or equal to all values in its right subtree

Note: In this discussion, duplicates of the root are permitted in either subtree.

Binary search trees function as "sorted trees" that maintain data in a sorted fashion even when input isn't sorted. This makes BSTs extremely efficient for operations like searching, while also maintaining efficient insertion and deletion.

8.1 Binary Search Tree Implementation

We'll define a `BinarySearchTree` class based on our previous `Tree` class, with important adaptations:

```
class BinarySearchTree:
    """
    Binary Search Tree class.

    #   - root:
    #       The item stored at the root of this tree, or None if this tree is empty.
    #   - left:
    #       The left subtree, or None if this tree is empty.
    #   - right:
    #       The right subtree, or None if this tree is empty.
    """
```

Key differences from our general `Tree` class:

- We use explicit `left` and `right` attributes instead of a general `subtrees` list
- For empty trees, `root`, `left`, and `right` are all `None`
- For non-empty trees, `left` and `right` are always `BinarySearchTree` objects (which might be empty)

Here are the initializer and `is_empty` methods:

```
def __init__(self, root):
    """
    Initialize a new BST containing only the given root value.

    If <root> is None, initialize an empty BST.
```

```
    """
    if root is None:
        self.root = None
        self.left = None
        self.right = None
    else:
        self.root = root
        self.left = BinarySearchTree(None)   # self._left is an empty BST
        self.right = BinarySearchTree(None)  # self._right is an empty BST

def is_empty(self):
    """
    Return whether this BST is empty.
    """
    return self.root is None
```

Note that we don't allow client code to directly pass left and right subtrees to the initializer, since BSTs have strict requirements for value placement. Instead, we'll use a separate method to insert values while preserving the BST property.

8.2 Searching a Binary Search Tree

The key advantage of BSTs becomes apparent in the search operation. Unlike general trees where we must potentially search every node, BSTs allow us to eliminate half the search space at each step:

```
def __contains__(self, item):
    """
    Return whether <item> is in this BST.
    """
    if self.is_empty():
        return False
    elif item == self.root:
        return True
    elif item < self.root:
        return self.left.__contains__(item)
    else:
        return self.right.__contains__(item)
```

This implementation makes only one recursive call at each step, rather than recursing on all subtrees as we would with a general tree. When searching for a value (e.g., 111):

- If the root is 50, we only need to search the right subtree (since $111 > 50$)
- If the root is 9000, we only need to search the left subtree (since $111 < 9000$)

This directed search approach is what gives BSTs their logarithmic time complexity for search operations.

8.3 Inorder Traversal

An inorder traversal of a binary search tree visits the nodes in the following order:

1. First visit all nodes in the left subtree (recursively)
2. Then visit the root node
3. Finally visit all nodes in the right subtree (recursively)

For a BST, this traversal produces the values in sorted (ascending) order, which is one of the key properties of a binary search tree.

```
def inorder_traversal(self):
    """
    Return a list of all items in this BST in sorted (inorder) order.
    """
    if self.is_empty():
```

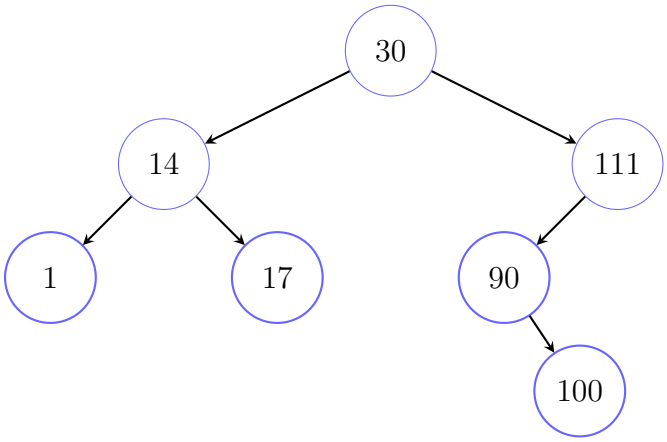
```
        return []
    else:
        # Get all values from left subtree
        left_values = self.left.inorder_traversal()

        # Add the root value
        root_value = [self.root]

        # Get all values from right subtree
        right_values = self.right.inorder_traversal()

        # Combine all values in the correct order
        return left_values + root_value + right_values
```

8.4 Insertion

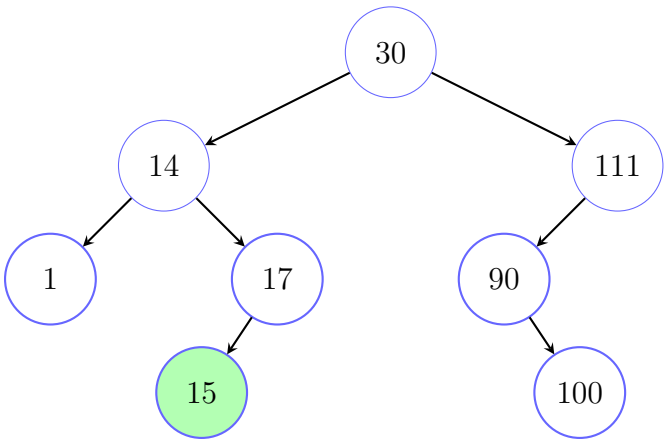
The most straightforward approach for BST insertion preserves the existing tree structure by placing new items at the "bottom" of the tree as leaves. This can be implemented recursively with the following logic:

- If the BST is empty, make the new item the root
- Otherwise, insert the item into either the left or right subtree while maintaining the BST property

For example, consider inserting 15 into this BST:



The only valid leaf position for 15 would be to the left of node 17.



The implementation follows a pattern similar to `__contains__`:

```
def insert(self, item):
    """Insert <item> into this BST."""
    if self.is_empty():
```

```
        # Make this node a leaf containing the item
        self.root = item
        self.left = BinarySearchTree(None)
        self.right = BinarySearchTree(None)
    elif item <= self.root:
        # Insert item in the left subtree
        self.left.insert(item)
    else:
        # Insert item in the right subtree
        self.right.insert(item)
```

8.5 Deletion

Deletion in a BST is more complex. We search for the item and, upon finding it at the root of some subtree, we remove it:

```
def remove(self, item):
    """
    Remove *one* occurrence of <item> from this BST.

    Do nothing if <item> is not in the BST.
    """
    if self.is_empty():
        pass
    elif self.root == item:
        self._remove_root()
    elif item < self.root:
        self.left.remove(item)
    else:
        self.right.remove(item)
```

We use a helper method `_remove_root` to handle the complexity of removing the root while maintaining the BST property.

Case 1: Leaf Node If the root has no children (i.e., both subtrees are empty), we simply set the root to `None` and ensure both subtrees are also `None` to maintain our representation invariant:

```
def _remove_root(self):
    """Remove the root of this tree.
    Preconditions:
      - not self.is_empty()
    """
    if self.left.is_empty() and self.right.is_empty():
        self.root = None
        self.left = None
        self.right = None
```

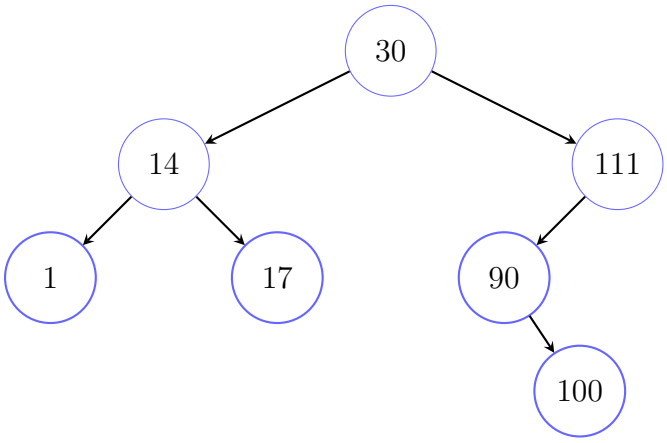
Case 2: One Empty Subtree When one subtree is empty and the other isn't, we can "promote" the non-empty subtree:

```
    elif self.left.is_empty():
        # "Promote" the right subtree.
        self.root, self.left, self.right = \
            self.right.root, self.right.left, self.right.right
    elif self.right.is_empty():
        # "Promote" the left subtree.
        self.root, self.left, self.right = \
            self.left.root, self.left.left, self.left.right
```

Case 3: Two Non-Empty Subtrees The most complex case arises when both subtrees contain elements. To maintain the BST property, we must replace the root with either:

- The maximum value from the left subtree, or
- The minimum value from the right subtree

Consider removing the root value 30 from this BST, where the maximum in left subtree is 17 and minimum in right subtree is 90:



Our implementation chooses to extract the maximum from the left subtree:

```
    else:
        self.root = self.left._extract_max()
```

Extracting the Maximum Value To find the maximum value in a BST, we follow right children until we reach a node with no right child. This node must have at most one child (on the left) by the BST property:

```
def _extract_max(self):
    """
    Remove and return the maximum item stored in this tree.
    Preconditions:
      - not self.is_empty()
    """
    if self.right.is_empty():
        max_item = self.root
        # Like remove_root, "promote" the left subtree.
        self.root, self.left, self.right = \
            self.left.root, self.left.left, self.left.right
        return max_item
    else:
        return self.right._extract_max()
```

The base case handles two scenarios: when the maximum node has a left child (but no right child) and when it has no children at all (a leaf). Both scenarios are properly managed by promoting any existing left subtree.

8.6 Binary Search Tree Efficiency

The efficiency of BST operations depends critically on the shape of the tree. In particular, the height of the tree (the length of the longest path from the root to a leaf) determines the worst-case complexity of most operations.

8.6.1 Best Case

The best case for search occurs when the item we're looking for is at the root of the tree. In this case, the search operation takes constant time, $O(1)$. For insertion, the best case occurs when we're inserting into an empty tree, which also takes $O(1)$ time.

Operation	Best Case	Average Case	Worst Case
Search	$O(1)$	$O(\log n)$	$O(n)$
Insert	$O(1)$	$O(\log n)$	$O(n)$
Delete	$O(1)$	$O(\log n)$	$O(n)$

Table 1: Time complexity of BST operations

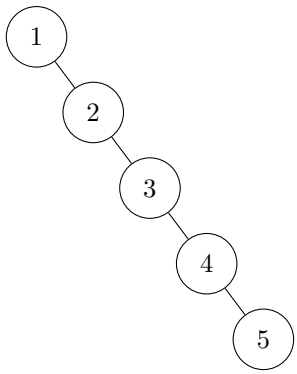
8.6.2 Average Case

When a BST is relatively balanced, the height of the tree is approximately $\log_2 n$, where n is the number of nodes. In this case, operations like search, insert, and delete are very efficient, with time complexity $O(\log n)$.

For example, a balanced BST with 1,000,000 elements would have a height of approximately 20, meaning that at most 20 comparison operations would be needed to find any element.

8.6.3 Worst Case

The worst case occurs when the tree becomes highly unbalanced or degenerate, resembling a linked list. This happens when elements are inserted in sorted (or reverse sorted) order.



In this case, the height of the tree becomes n, and the time complexity of all operations degrades to $O(n)$, which is no better than a simple linked list.

9 Heap

Definition 9 (heap property)**.** *A tree satisfies the heap property if and only if for each node in the tree, the value of that node is greater than or equal to the value of all its descendants.*

This property is less restrictive than the BST property: given a node satisfying the heap property, we cannot determine any relationship between its left and right subtrees. Consequently, it's much easier to create a compact, balanced binary tree that satisfies the heap property compared to one satisfying the BST property. In fact, we can enforce the strongest possible balancing for our data structure.

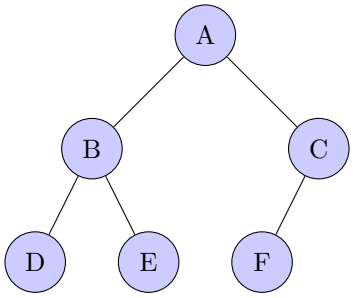
Definition 10 (complete binary tree)**.** *A binary tree is complete if and only if it satisfies the following conditions:*

- *All levels are full, except possibly the bottom one. Every leaf must be in one of the two bottommost levels.*
- *All nodes in the bottom level are positioned as far left as possible.*

Complete trees are the most "compact" trees possible. A complete tree with n nodes has height $\lfloor \log(n+1) \rfloor$, which is the minimum possible height for a tree with this number of nodes.

Furthermore, because nodes in the bottom layer must be as far left as possible, there's never any ambiguity about where the "empty" spots in a complete tree are located. **There exists only one complete tree shape for each number of nodes.**

Due to this property, we don't need to allocate space to store references between nodes, as in standard binary tree implementations. Instead, we establish a conventional ordering of nodes in the heap, then record the items according to that order. The ordering we use is called **level order**, as it arranges elements based on their depth in the tree: **the first node is the tree's root, followed by its children (at depth 1) in left-to-right order, then all nodes at depth 2 in left-to-right order, and so on.**



The array representation of the above complete binary tree.

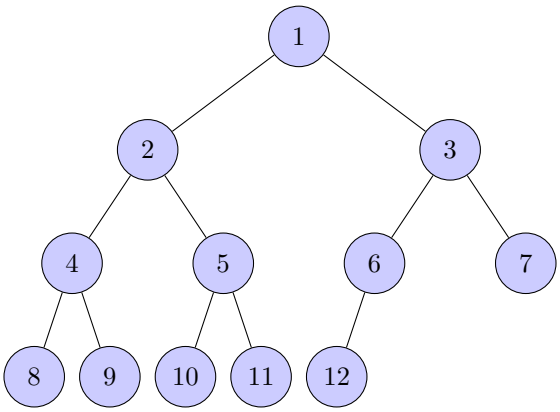
A	B	C	D	E	F

With these definitions established, we can now define a heap:

Definition 11 (heap)**.** *A heap is a binary tree that satisfies both the heap property and completeness.*

We implement a heap in programs as an **array**, **where array items correspond to the level order of the actual binary tree**. We can use simple array to do this because of the following observation:

Assuming items are stored starting at index 1 rather than 0, we observe the following pattern: **for a node at index i, its left child is stored at index $2i$, and its right child at index $2i+1$. Conversely, the parent of a node at index i (where $i > 1$) is stored at index $\lfloor i/2 \rfloor$.** This relationship between indices will prove valuable in the next section when we perform more complex heap operations.



We are showing the indices where each node would be stored in the array.

9.1 Define A Max Heap

Now, based on the definitions above, let's define our max heap data structure:

- Elements are stored as list of (item, priority) tuples
- The priority value determines the ordering in the heap
- Higher priority values will be closer to the root

```
class MaxHeap:
    """
    A MaxHeap implementation using a list with 1-based indexing.
    Each element in the heap is a tuple (item, priority), where 'priority' determines the order.
    """

    def __init__(self, items=None):
        """
```

```
        Initializes the MaxHeap. If a list of (item, priority) tuples is provided, builds the heap.
        """
        self.heap = [None]  # Using 1-based indexing
        self.size = 0
        if items:
            self.build_heap(items)
```

The None at index 0 is a placeholder that's never used, which simplifies the parent-child index calculations.

9.2 Build A Max Heap

Let's understand the two key methods in a MaxHeap implementation: `build_heap` and `_bubble_down`.

```
def build_heap(self, items):
    """
    Builds a max heap from a list of (item, priority) tuples.
    """
    self.heap = [None] + items  # Add a dummy element at index 0
    self.size = len(items)
    for i in range(self.size // 2, 0, -1):  # Start heapifying from the last parent node
        self._bubble_down(i)
```

The `build_heap` method transforms an arbitrary list of (item, priority) tuples into a valid max heap. Here's how it works step by step:

1. **Initialization:**
   - `self.heap = [None] + items`: Creates a new list where index 0 contains `None` (unused), and indices 1 through n contain the input items. This enables 1-based indexing which simplifies the parent-child index relationships.
   - `self.size = len(items)`: Updates the heap size to reflect the number of actual elements.
2. **Bottom-up heap construction:**
   - `for i in range(self.size // 2, 0, -1)`: Iterates from $\lfloor n/2 \rfloor$ down to 1.
   - Why start at $\lfloor n/2 \rfloor$? This is the index of the last non-leaf node in the heap. Leaf nodes (which have no children) already satisfy the heap property trivially (heap property only constrains parent nodes).
   - The iteration proceeds in reverse order, moving from the lowest levels of the tree upward toward the root.
3. **Heapification:**
   - `self._bubble_down(i)`: For each parent node, this ensures that the subtree rooted at that node satisfies the max-heap property.
   - By the time we reach the root (index 1), the entire tree will satisfy the max-heap property.

```
def _bubble_down(self, i):
    """
    Moves an element down the heap to maintain the max heap property after extraction.
    """
    while 2 * i <= self.size:
        left = 2 * i
        right = 2 * i + 1
        largest = i

        # Check if left child has higher priority than current node
        if left <= self.size and self.heap[left][1] > self.heap[largest][1]:
            largest = left
        # Check if right child has higher priority than the largest so far
        if right <= self.size and self.heap[right][1] > self.heap[largest][1]:
            largest = right

        if largest == i:  # If heap property is maintained, stop
```

```
            break

        self.heap[i], self.heap[largest] = self.heap[largest], self.heap[i]  # Swap with larger child
        i = largest
```

The `_bubble_down` method restores the max-heap property when a node might be smaller than its children. Here's how it works:

1. **Loop condition:**
   - `while 2 * i <= self.size`: Continue as long as the current node has at least a left child. If $2i > n$, then the node is a leaf, and we're done (if a node only has one child, then it must be left because we ensure it is compact and every node is as far left as possible).
2. **Find the largest among the node and its children:**
   - `left = 2 * i` and `right = 2 * i + 1`: Calculate the indices of left and right children.
   - `largest = i`: Initially assume the current node has the highest priority.
   - Compare with the left child: If the left child exists (`left <= self.size`) and has higher priority than the current largest, update `largest = left`.
   - Compare with the right child: If the right child exists (`right <= self.size`) and has higher priority than the current largest, update `largest = right`.
3. **Check if swapping is needed:**
   - `if largest == i`: If the current node is still the largest, the heap property is satisfied for this subtree, so break out of the loop.
   - Otherwise, the heap property is violated, and a swap is needed.
4. **Swap and continue:**
   - `self.heap[i], self.heap[largest] = self.heap[largest], self.heap[i]`: Swap the current node with its largest child.
   - `i = largest`: Move down to the position of the largest child and continue the process.
   - This continues until either the node reaches a position where it's larger than both children or becomes a leaf node.

The time complexity analysis of `build_heap` is interesting. A naive analysis suggests $O(n \log n)$, but a tighter bound shows that it's actually $O(n)$. This is because the operation on nodes with many descendants (near the top of the tree) is $O(\log n)$, but there are few such nodes. Nodes near the bottom have $O(1)$ operations, and there are many of them.

9.3 Extract Max

The `extract_max` method is one of the fundamental operations in a max heap data structure. It removes and returns the item with the highest priority (the root of the heap). Let's examine how this operation works line by line:

```
def extract_max(self):
    """
    Removes and returns the item with the highest priority (the root of the heap).
    """
    if self.size == 0:
        raise IndexError("Extract from an empty heap")

    max_item = self.heap[1]  # The root element
    self.heap[1] = self.heap[self.size]  # Replace root with last element
    self.size -= 1
    self.heap.pop()  # Remove last element
    self._bubble_down(1)  # Restore heap property

    return max_item
```

1. **Error Handling**:

   First, we check if the heap is empty (`self.size` = 0). If it is, we raise an `IndexError` since we cannot extract from an empty heap.

2. **Store the Maximum Element**:

   Since the root of the heap (at index 1) always contains the element with the highest priority in a max heap, we store this value to return later.

3. **Replace the Root with the Last Element**:

   We replace the root element with the last element in the heap. This maintains the **complete binary tree property** but might violate the max heap property.

4. **Update Size and Remove Last Element**:

   We decrement the size of the heap and remove the last element from the array (which is now a duplicate since we've already moved it to the root position).

5. **Restore Heap Property**:

   After replacing the root with the last element, the heap property may be violated. The `_bubble_down` method is called to restore the max heap property starting from the root node (index 1). This method compares the node with its children and swaps it with the larger child if necessary, continuing this process down the heap until the heap property is restored.

The time complexity of `extract_max` is $O(\log n)$, where n is the number of elements in the heap. This is because:

- Accessing the maximum element (the root) is $O(1)$.
- Replacing the root with the last element is $O(1)$.
- Removing the last element is $O(1)$.
- The `_bubble_down` operation takes $O(\log n)$ time in the worst case, as it may need to traverse from the root to a leaf along a single path, and the height of a complete binary tree with n nodes is $\lfloor \log_2 n \rfloor$.

This approach ensures that we don't have to shift elements as in an array, which would be an $O(n)$ operation. Instead, we get an efficient $O(\log n)$ operation.

9.4 Insert

Let's examine the `insert` method of the MaxHeap class:

```
def insert(self, item, priority):
    """
    Inserts a new (item, priority) tuple into the heap and maintains heap order.
    """
    self.size += 1
    self.heap.append((item, priority))
    self._bubble_up(self.size)
```

The `insert` method adds a new element to the heap while maintaining the heap property (every parent node has a higher or equal priority than its children). Here's how it works:

1. **Increment the size counter:** `self.size += 1`
   This updates the count of elements in our heap.

2. **Add the new element:** `self.heap.append((item, priority))`
   The new element is added as a tuple containing the actual item and its priority to the end of the list.

3. **Restore heap property:** `self._bubble_up(self.size)`
   After adding the new element at the end of the heap (as a leaf node), we need to ensure the max-heap property is preserved. The `_bubble_up` method compares the newly added element with its parent and swaps them if the child has higher priority, repeating this process up the tree until the element reaches its proper position.

```
def _bubble_up(self, i):
    """
    Moves an element up the heap to maintain the max heap property after insertion.
    """
```

```
    while i > 1:
        parent = i // 2
        if self.heap[i][1] <= self.heap[parent][1]:  # If parent has higher priority, stop
            break
        self.heap[i], self.heap[parent] = self.heap[parent], self.heap[i]  # Swap with parent
        i = parent
```

The `_bubble_up` method is where the key logic for maintaining the max heap property happens after an insertion. Here's how it works:

1. `while i > 1:` - We continue the process as long as we haven't reached the root of the heap (index 1). Copy
2. `parent = i // 2` - Calculate the index of the parent node. In a binary heap represented as an array with 1-based indexing, the parent of node i is at position $\lfloor i/2 \rfloor$.
3. `if self.heap[i][1] <= self.heap[parent][1]: break` - This compares the priority of the current node with its parent's priority, if the parent's priority is greater than or equal to the current node's priority, the max heap property is satisfied, and we can stop the process
4. `self.heap[i], self.heap[parent] = self.heap[parent], self.heap[i]` - If we get here, it means the current node has higher priority than its parent, violating the max heap property. We swap the current node with its parent to restore the property.
5. `i = parent` - Finally, we update our index to be the parent's index, as we now need to check if the node (which we just moved up) needs to be bubbled up further.

Insertion has a time complexity of $O(\log n)$ where n is the number of elements in the heap, because in the worst case, an element might need to move from the bottom to the top of the heap, and the heap's height is $\log n$.

9.5 Handle Equal Priority Items

Looking at the MaxHeap implementation earlier, it does not guarantee FIFO (First-In-First-Out) ordering for items with the same priority. The issue is that when two items have the same priority value, their relative ordering in the heap becomes unpredictable and depends on their positions in the heap structure. This means that when extracting items with equal priorities, there's no guarantee that the one inserted first will be extracted first.

We apply a **Composite Priority Key**: Instead of storing just the priority, we now store a tuple of (priority, -timestamp) where the timestamp is a counter that increments with each insertion. **The negative sign ensures that earlier items (smaller timestamps) get higher precedence in the max heap.**

Further, Python handles tuple comparisons automatically in a lexicographical manner:

- It first compares the first elements of each tuple
- If the first elements are equal, it then compares the second elements
- And so on for longer tuples

This gives us the convenience to do a very slight modification:

1. **Added a timestamp mechanism**
   - Introduced `self.timestamp = itertools.count()` in the `__init__` method.
   - This generates a unique increasing number for each inserted item.
   - Ensures that items with the same priority maintain FIFO order.
2. **Updated the `insert` method**
   - Changed the insertion process from:

     ```
     self.heap.append((item, priority))
     ```

   - To:

     ```
     self.heap.append((item, (priority, -next(self.timestamp))))
     ```

   - The `next(self.timestamp)` function assigns a unique timestamp.
3. **Updated the `build_heap` method**

- When constructing the heap from an initial list, changed:

```
self.heap = [None] + items
```

- To:

```
self.heap = [None] + [(item, (priority, -next(self.timestamp)))
                                                    for item, priority in items]
```

- Ensures each item receives a timestamp at initialization.

4. **Updated the `extract_max` method**
   - The extracted max item originally returned as:

```
return max_item
```

   - Was changed to:

```
return max_item[0], max_item[1][0]  # Stripping out the timestamp
```

   - Ensures that only the `(item, priority)` is returned, not the timestamp.
5. **No changes were made to `_bubble_up` and `_bubble_down`**
   - Since tuples in Python are compared lexicographically, comparisons still work as expected.
   - The heap property is maintained while ensuring FIFO ordering for equal-priority items.

```python
import itertools

class MaxHeap:
    """
    A MaxHeap implementation using a list with 1-based indexing.
    Each element in the heap is a tuple (item, (priority, time)), ensuring FIFO order for items with the same priority.
    """

    def __init__(self, items=None):
        """
        Initializes the MaxHeap. If a list of (item, priority) tuples is provided, builds the heap.
        """
        self.heap = [None]  # Using 1-based indexing
        self.size = 0
        self.timestamp = itertools.count()  # Generates unique timestamps
        if items:
            self.build_heap(items)

    def extract_max(self):
        """
        Removes and returns the item with the highest priority (the root of the heap).
        """
        if self.size == 0:
            raise IndexError("Extract from an empty heap")

        max_item = self.heap[1]  # The root element
        self.heap[1] = self.heap[self.size]  # Replace root with last element
        self.size -= 1
        self.heap.pop()  # Remove last element
        self._bubble_down(1)  # Restore heap property

        return max_item[0], max_item[1][0]  # Return item and priority (excluding timestamp)

    def insert(self, item, priority):
        """
        Inserts a new (item, priority) tuple into the heap and maintains heap order.
        """
        self.size += 1
```

```
        self.heap.append((item, (priority, -next(self.timestamp))))  # Assign a unique timestamp
        self._bubble_up(self.size)

    def build_heap(self, items):
        """
        Builds a max heap from a list of (item, priority) tuples.
        """
        self.heap = [None] + [(item, (priority, -next(self.timestamp))) for item, priority in items]
        self.size = len(items)
        for i in range(self.size // 2, 0, -1):  # Start heapifying from the last parent node
            self._bubble_down(i)

    def _bubble_up(self, i):
        """
        Moves an element up the heap to maintain the max heap property after insertion.
        """
        while i > 1:
            parent = i // 2
            if self.heap[i][1] <= self.heap[parent][1]:  # Compare priority, then timestamp
                break
            self.heap[i], self.heap[parent] = self.heap[parent], self.heap[i]  # Swap with parent
            i = parent

    def _bubble_down(self, i):
        """
        Moves an element down the heap to maintain the max heap property after extraction.
        """
        while 2 * i <= self.size:
            left = 2 * i
            right = 2 * i + 1
            largest = i

            # Check if left child has higher priority than current node
            if left <= self.size and self.heap[left][1] > self.heap[largest][1]:
                largest = left
            # Check if right child has higher priority than the largest so far
            if right <= self.size and self.heap[right][1] > self.heap[largest][1]:
                largest = right

            if largest == i:  # If heap property is maintained, stop
                break

            self.heap[i], self.heap[largest] = self.heap[largest], self.heap[i]  # Swap with larger child
            i = largest

```

9.6 Heap Sort

Given a max heap, the largest element is always at the root. Heap Sort takes advantage of this property with the following steps:

1. Start with a valid max heap (where the largest element is at the root)
2. Repeatedly:
   (a) Swap the root (maximum element) with the last unsorted element
   (b) Reduce the heap size by 1 (to exclude the sorted element)
   (c) Restore the heap property by bubbling down the new root
3. After all elements are processed, the array will be sorted in ascending order

Let's analyze the `heapsort` method from our `MaxHeap` class:

```
def heapsort(self):
    """
```

```
    Sorts the elements in the heap in-place using the heap sort algorithm.
    Returns the sorted list of (item, priority) tuples in ascending order.
    """
    # Repeated build up a sorted list from the back of the list, in place
    sorted_index = self.size
    original_size = self.size
    while sorted_index > 1:
        # Swap the root (maximum element) with the last unsorted element
        self.heap[sorted_index], self.heap[1] = self.heap[1], self.heap[sorted_index]
        # Decrease the size of the heap (excludes the sorted part)
        sorted_index = sorted_index - 1

        # Temporarily adjust the heap size to exclude the sorted part
        self.size = sorted_index

        # Restore the heap property for the reduced heap
        self._bubble_down(1)

    # Restore the original size
    self.size = original_size
    # Return the sorted heap (note: it will be in ascending order)
    return self.heap[1:]
```

1. **Initialization**: The algorithm starts with `sorted_index` equal to the heap size. This index represents the boundary between the unsorted portion (the active heap) and the sorted portion at the end of the array.
2. **Main Loop**:
   (a) **Loop Condition**: The main loop continues as long as `sorted_index > 1`, meaning there are still unsorted elements to process
   (b) **Swap Root with Last Unsorted Element**: This operation places the maximum element (at the root) at position `sorted_index`, which is the end of the unsorted portion. Since we're using a max heap, this element is guaranteed to be the largest among the unsorted elements.
   (c) **Adjust Boundaries**: We decrement `sorted_index` to shrink the unsorted portion and expand the sorted portion.
   (d) **Temporarily Adjust Heap Size**: Before restoring the heap property, we temporarily adjust the heap size to match `sorted_index`. This ensures that `_bubble_down` only operates on the unsorted portion and doesn't touch the sorted elements.
   (e) **Restore Heap Property**: After swapping, the element at the root may not be the maximum, violating the max-heap property. We call `_bubble_down` on the root to restore this property for the reduced heap.
3. **Restore Original Size**: We restore the original heap size after the heap sort completion.
4. **Return Sorted List**: Once the loop completes, all elements have been sorted in ascending order (smallest to largest) in the heap array, so we return the heap without the dummy element at index 0.

`Compared with MergeSort and QuickSort:`

- **Guaranteed Performance**: Unlike quick sort, heap sort maintains O(n log n) complexity even in worst-case scenarios.
- **Memory Efficiency**: Heap sort is in-place, using only constant extra space, making it more memory-efficient than merge sort.

9.7 Why Build-Heap is O(n)?

At first glance, it may seem that building a heap should take $O(n \log n)$, since we call `heapify` on many nodes and each call may cost $O(\log n)$. However, this is not the case.

The key observation is that not every node requires $O(\log n)$ work. In a complete binary tree, most nodes are located near the leaves, and those nodes can move down only a few levels. Only a small number of nodes near the root can move down a large distance.

More precisely, if a node is at height h from the bottom, then `heapify` on that node costs at most $O(h)$. Also, the number of nodes at height h is at most $\frac{n}{2^{h+1}}$:

$$\#(\text{nodes of height } h) \;\le\; \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

For $n = 9$ and $h = 0$, this gives

$$\left\lceil \frac{9}{2} \right\rceil = 5,$$

which matches the actual **number of leaves** in a 9-node heap.

Example: heap of size 9. A complete binary tree with 9 nodes has the following number of nodes at each level:

$$1,\ 2,\ 4,\ 2.$$

If we count height from the bottom, then:

$$\begin{aligned}&\text{height } 0 : 5 \text{ nodes},\\ &\text{height } 1 : 2 \text{ nodes},\\ &\text{height } 2 : 1 \text{ node},\\ &\text{height } 3 : 1 \text{ node}.\end{aligned}$$

Tsherefore, the total cost can be written as

$$\sum_{h=0}^{\lfloor \log n \rfloor} O\Big(h \left\lceil \frac{n}{2^{h+1}} \right\rceil\Big),$$

which is still $O(n)$.

Let

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} O\Big(h \left\lceil \frac{n}{2^{h+1}} \right\rceil\Big).$$

We will show that $T(n) = O(n)$.

Set

$$L = \lfloor \log n \rfloor.$$

First, consider the terms with $0 \le h \le L - 1$. For such h, we have

$$\frac{n}{2^{h+1}} \ge 1.$$

Hence,

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil \le 2 \cdot \frac{n}{2^{h+1}} = \frac{n}{2^h}.$$

Therefore,

$$\sum_{h=0}^{L-1} O\Big(h \left\lceil \frac{n}{2^{h+1}} \right\rceil\Big) \le \sum_{h=0}^{L-1} O\Big(h \cdot \frac{n}{2^h}\Big) = O\left(n \sum_{h=0}^{L-1} \frac{h}{2^h}\right).$$

Now observe that the infinite series

$$\sum_{h=0}^{\infty} \frac{h}{2^h}$$

converges to a constant. Thus,

$$\sum_{h=0}^{L-1} \frac{h}{2^h} = O(1),$$

and so

$$\sum_{h=0}^{L-1} O\Big(h \left\lceil \frac{n}{2^{h+1}} \right\rceil\Big) = O(n).$$

Next, consider the remaining term with $h = L$. Since $L = \lfloor \log n \rfloor$, we have

$$\frac{n}{2^{L+1}} < 1.$$

Therefore,

$$\left\lceil \frac{n}{2^{L+1}} \right\rceil \leq 1,$$

so the last term is

$$O\Big(L \left\lceil \frac{n}{2^{L+1}} \right\rceil\Big) = O(L) = O(\log n).$$

Combining the two parts, we obtain

$$T(n) = \sum_{h=0}^{L-1} O\Big(h \left\lceil \frac{n}{2^{h+1}} \right\rceil\Big) + O\Big(L \left\lceil \frac{n}{2^{L+1}} \right\rceil\Big) = O(n) + O(\log n).$$

Since $O(\log n) \subseteq O(n)$, it follows that

$$T(n) = O(n).$$

The reason this works is that nodes near the root may require more work, but there are very few of them, while the many nodes near the bottom have very small height. As a result, the total weighted sum grows only linearly with n.

10 Graph

Consider a social network: it consists of you and thousands or millions of other users, each possessing individual accounts and data. What makes a social network truly *social* is the existence of links—relationships—between users, which may take various forms such as friends, followers, subscribers, clients, partners, and others.

Although social media applications and airports serve vastly different purposes in our daily lives, both rely on representing data as a *network*, comprised of entities and the connections between them. Throughout this chapter, we will examine the formal data type that computer scientists employ to represent networks, exploring both the theoretical properties of this data type and implementing it in Python.

Let us start with some basic definitions.

Definition 12. *A **graph** is a pair of sets (V, E), which are defined as follows:*

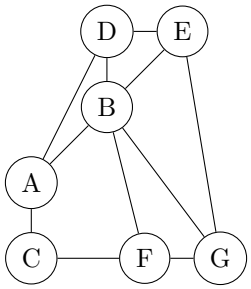
- *V is a set of objects. Each element of V is called a **vertex** of the graph, and V itself is called the set of **vertices** of the graph.*
- *E is a set of pairs of objects from V, where each pair $\{v_1, v_2\}$ is a set consisting of two distinct vertices—i.e., $v_1, v_2 \in V$ and $v_1 \neq v_2$—and is called an **edge** of the graph.*

  Order does not matter in the pairs, and so $\{v_1, v_2\}$ and $\{v_2, v_1\}$ represent the same edge.

The conventional notation to introduce a graph is to write $G = (V, E)$, where G is the graph itself, V is its vertex set, and E is its edge set.

Intuitively, the set of vertices of a graph represents a collection of objects, and the set of edges of a graph represent the relationships between those objects. For example, if we wanted to use the terminology of graphs to describe Facebook, we could say that each Facebook user is a *vertex*, while each friendship between two Facebook users is an *edge* between the corresponding vertices.

Let's consider an example to practice the above terminology. Consider the graph below. This graph consists of *seven* vertices (the circles with letters A through G) and *eleven* edges.



You'll notice that graph diagrams look similar to the tree diagrams. And indeed there are many similarities between graphs and trees, right now, you can view graphs as a generalization of trees: rather than enforcing a strict parent-child hierarchy on the data, graphs support any vertex being joined by an edge to any other vertex.

Our next set of definitions introduces one of the key properties of a vertex in a graph: how many edges that vertex is a part of.

Definition 13. *Let $G = (V, E)$, and let $v_1, v_2 \in V$. We say that v_1 and v_2 are **adjacent** if and only if there exists an edge between them, i.e., $\{v_1, v_2\} \in E$. Equivalently, we can also say that v_1 and v_2 are **neighbours**.*

Definition 14. *Let $G = (V, E)$, and let $v \in V$. We say that the **degree** of v, denoted $d(v)$, is its number of neighbours, or equivalently, how many edges v is a part of.*

For example, in our example graph above, we could say that the vertices A and B are adjacent, but A and G are not. The degree of vertex A is 3, since it has three neighbours: B, C, and D.

Often when we use graphs in modelling the real world, it is not sufficient to capture just a single relationship between entities. Our goal now is to use individual edges, which represent some sort of relationship between vertices, to build up extended, indirect connections between vertices. In a social network, for example, we want to be able to go from friends to "friends of friends," and even "friends of friends of friends of friends."

Definition 15. *Let $G = (V, E)$ and let $u, u' \in V$. A **path** between u and u' is a sequence of distinct vertices $v_0, v_1, v_2, \ldots, v_k \in V$ which satisfy the following properties:*

- *$v_0 = u$ and $v_k = u'$. (The endpoints of the path are u and u'.)*

- *Each consecutive pair of vertices are adjacent. (So v_0 and v_1 are adjacent, and so are v_1 and v_2, and v_2 and v_3, etc.)*

We allow k to be zero; this path would be just a single vertex v_0.

*The **length** of a path is one less than the number of vertices in the sequence (so the above sequence would have length k); more intuitively, the length of the path is the number of edges which are used by this sequence.*

We say that graph G is **connected** when for all pairs of vertices $u, v \in V$, u and v are connected.

Being connected is a fundamental property of graphs. Imagine, for example, a geographical representation where each graph vertex is a city, and each edge a road between two cities. If this graph is not connected, then there is at least one pair of cities for which it is not possible to get from one to the other by road.

10.1 Representing Graphs in Python

Intuitively, each vertex of a graph will correspond to a `node object`, and the edges of a graph will be represented implicitly by having each node store references to its neighbours.

Here is the start of a _Vertex class, which represents a single vertex in a graph.

```
class _Vertex:
    """A vertex in a graph.

    Instance Attributes:
    - item: The data stored in this vertex.
    - neighbours: The vertices that are adjacent to this vertex.
    """
    def __init__(self, item, neighbours):
        """Initialize a new vertex with the given item and neighbours."""
        self.item = item
        self.neighbours = neighbours
```

This definition is fairly straightforward: the `item` instance attribute stores the `data` in each vertex, which may be an integer, a string, or a custom data type that we define. The `neighbours` attribute is how we encode the graph edges: it stores a set of other _Vertex objects. This attribute is structurally similar to the `Tree` _subtrees attribute, but now we're back to treating the class as representing a single element of the graph, rather than the whole graph itself. Also note that this collection is **unordered**.

For example, here is how we could represent a "triangle": three vertices that are all adjacent to each other.

```
v1 = _Vertex('a', set())
v2 = _Vertex('b', set())
v3 = _Vertex('c', set())
v1.neighbours = {v2, v3}
v2.neighbours = {v1, v3}
v3.neighbours = {v1, v2}
```

Recall that graph edges have two important restrictions: we cannot have a vertex with an edge to itself, and all edges are symmetric (that is, the edge $\{u, v\}$ is equivalent to the edge $\{v, u\}$).

As you're probably expecting by now, we can enforce these properties by adding two representation invariants to our _Vertex class:

- self not in self.neighbours
- all(self in u.neighbours for u in self.neighbours)

You've probably noticed that the _Vertex class is pretty similar to the _Node class we developed for linked lists. The key difference in representation is the "link" attribute: whereas _Node s has a `next` attribute that referred to a single other _Node, now our _Vertex's `neighbours` attribute can refer—or link—to multiple other _Vertex objects.

Now, you might point out that our Tree class also had an attribute _subtrees that could refer to multiple other Trees. The main difference here is that we won't enforce any tree-like hierarchy on the nodes in `neighbours`, and instead allow cycles and other forms of vertex "interconnectedness" that would have been logically inconsistent with how we viewed trees.

Next, we can define a Graph class that simply consists of a collection of _Vertex objects.

```
class Graph:
    """A graph.

    - _vertices: A collection of the vertices contained in this graph.
    #Maps item to _Vertex instance.
    """

    def __init__(self):
        """Initialize an empty graph (no vertices or edges)."""
        self._vertices = {}
```

Similar to our other collection data types, our initializer is very restricted: we only create empty graphs. In order to build up a larger graph, we'll provide mutating methods to insert new vertices and edges. Here are two such methods:

```
class Graph:
    def add_vertex(self, item):
        """Add a vertex with the given item to this graph.

        The new vertex is not adjacent to any other vertices.

        Preconditions:
        - item not in self._vertices
        """
        self._vertices[item] = _Vertex(item, set())

    def add_edge(self, item1, item2):
        """Add an edge between the two vertices with the given items in this graph.
        Raise a ValueError if item1 or item2 do not appear as vertices in this graph.

        Preconditions:
        - item1 != item2
        """
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            v2 = self._vertices[item2]

            # Add the new edge
            v1.neighbours.add(v2)
            v2.neighbours.add(v1)
        else:
            # We didn't find an existing vertex for both items.
            raise ValueError
```

With this, we can create Graph objects and populate them with vertices and edges:

```
g = Graph()
g.add_vertex('a')
g.add_vertex('b')
g.add_vertex('c')
g.add_edge('a', 'b')
g.add_edge('a', 'c')
g.add_edge('b', 'c')
```

One subtlety about this Python representation of graphs is how it represents edges. Even though we defined a graph formally to Graphs as a collection of vertices and edges, our Graph class only has one instance attribute, `vertices`. That's because the edges are stored implicitly through each _Vertex's `neighbours` attribute. As a result of this representation, for example, we can easily iterate over all vertices in a graph, but not easily iterate over all edges in a graph.

One of the most common operations on graphs is asking two questions about adjacency: *Are these two items adjacent?* and more generally, *What items are adjacent to this item?* We can implement two Graph methods that answer these questions, using the same techniques as the previous section.

```
class Graph:
    def adjacent(self, item1, item2):
        """Return whether item1 and item2 are adjacent vertices in this graph.

        Return False if item1 or item2 do not appear as vertices in this graph.
        """
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            return any(v2.item == item2 for v2 in v1.neighbours)
        else:
            # We didn't find an existing vertex for both items.
            return False

    def get_neighbours(self, item):
        """Return a set of the neighbours of the given item.

        Note that the *items* are returned, not the _Vertex objects themselves.

        Raise a ValueError if item does not appear as a vertex in this graph.
        """
        if item in self._vertices:
            v = self._vertices[item]
            return {neighbour.item for neighbour in v.neighbours}
        else:
            raise ValueError
```

And to wrap up, here's a doctest example putting together all of the methods we developed in this section:

```
g = Graph()
g.add_vertex('a')
g.add_vertex('b')
g.add_vertex('c')
g.add_vertex('d')
g.add_edge('a', 'b')
g.add_edge('b', 'c')
print(g.adjacent('a', 'b')) #True
print(g.adjacent('a', 'd')) #False
print(g.get_neighbours('b') == {'a', 'c'}) #True
print(g.get_neighbours('d') == set()) #True
```

10.2 Connectivity and Recursive Graph Traversal

Now that we have covered the basics of a Python representation of graphs, we are ready to perform some interesting graph computations! First, recall this graph `adjacent` method in Python:

```
class Graph:
    def adjacent(self, item1, item2):
        """Return whether item1 and item2 are adjacent vertices in this graph.
        Return False if item1 or item2 do not appear as vertices in this graph.
        """
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            return any(v2.item == item2 for v2 in v1.neighbours)
        else:
            # We didn't find an existing vertex for both items.
            return False
```

Our goal in this section will be to implement a generalization of this method that goes from checking whether two vertices are adjacent to whether two vertices are connected:

```
class Graph:
    def connected(self, item1, item2):
        """Return whether item1 and item2 are connected vertices in this graph.
        Return False if item1 or item2 do not appear as vertices in this graph.
        """
```

We can start implementing this method in the same fashion as `Graph.adjacent`, by finding the vertex corresponding to `item1`:

```
class Graph:
    def connected(self, item1, item2):
        """..."""
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            ...
        else:
            return False
```

The problem, of course, is in the final `else` branch. We can't simply check whether `item2` is an immediate neighbour of `item1`, since there are many other possible cases for two vertices being connected: `item2` could equal `item1` itself, or it could be a neighbour, or the neighbour of a neighbour, or the neighbour of a neighbour of a neighbour, etc.

So you might be tempted to start writing code that tries to cover all these cases:

```
class Graph:
    def connected(self, item1, item2):
        """..."""
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            if v1.item == item2:
                return True
            elif any(u.item == item2 for u in v1.neighbours):
                return True
            elif any(any(u1.item == item2 for u1 in u.neighbours)
                     for u in v1.neighbours):
                return True
            ...
        else:
            return False
```

You can see why this gets unmanageable very quickly, and fact cannot possibly cover all possibilities! Because we need to handle an arbitrary number of "neighbour" links, we can't write a fixed amount of code that enumerates the possibilities; instead, we either need to use a loop or recursion. We'll develop a recursive approach although it is possible to translate this approach into one that uses loops as well.

```
class Graph:
    def connected(self, item1, item2):
        """..."""
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            return v1.check_connected(item2)
        else:
            return False
```

Note that our helper method, `check_connected` is called with `vertex` to the left of the dot—that means that it's a method of the `_Vertex` class, not `Graph`. Now let's recall our definition of connectedness: two vertices are connected

when there exists a path between them. The challenge with this definition is that it only requires that a path exists, but we don't know how long it is. But there's another, equivalent way of defining connectedness recursively.

Definition 16 (connectivity, recursive)**.** *Let $G = (V, E)$ be a graph, and let $v_1, v_2 \in V$. We say that v_1 and v_2 are connected when:*

- *$v_1 = v_2$, or*
- *there exists a neighbour u of v_1 such that u and v_2 are connected.*

This definition maps onto the recursive definition structure we've seen so far, with the first point being the base case and the second being the recursive part. However, this definition is different from all the previous recursive definitions in one important respect: it isn't exactly structural, since it doesn't break down the data type—a graph—into a smaller instance with the same structure. We'll come back to this point further down below.

But first, well, let's take this recursive definition and run with it! Even though we're still new to graphs, everything we've learned about translating recursive definitions into Python still applies, and our code actually turns out to be surprisingly simple:

```
class _Vertex:
    def check_connected(self, target_item):
        """Return whether this vertex is connected to a vertex corresponding to the target_item.
        """
        if self.item == target_item:
            # Our base case: the target_item is the current vertex
            return True
        else:
            for u in self.neighbours:
                if u.check_connected(target_item):
                    return True
            return False
```

Pretty simple! But unfortunately, our implementation has a flaw that leads to a new kind of error:

```
g = Graph()
g.add_vertex(1)
g.add_vertex(2)
g.add_vertex(3)
g.add_edge(1, 2)
g.connected(1, 3) # Should return False!
```

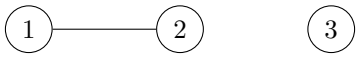
leading to an error:

```
Traceback (most recent call last):
  File "./src/graph.py", line 103, in <module>
    g.connected(1, 3) # Should return False!
  File "./src/graph.py", line 94, in connected
    return v1.check_connected(item2)
  File "./src/graph.py", line 29, in check_connected
    if u.check_connected(target_item):
  File "./src/graph.py", line 29, in check_connected
    if u.check_connected(target_item):
  File "./src/graph.py", line 29, in check_connected
    if u.check_connected(target_item):
  [Previous line repeated 994 more times]
  File "./src/graph.py", line 24, in check_connected
    if self.item == target_item:
RecursionError: maximum recursion depth exceeded in comparison
```

Our implementation for `_Vertex.check_connected` is simple, and exactly follows the recursive definition of connectedness we gave above. So what we're running into here is not an error in our logic, but a limitation of how we've expressed that logic in our code. To understand what's gone wrong, let's do a bit of manual tracing of our example above.

First, we have a graph `g` with three vertices, containing the items 1, 2, and 3. In our initial call to `g.connected(1, 3)`, we first find the vertex corresponding to `1`, which we'll name v_1. Then our code calls `_Vertex.check_connected` on v_1 and `3`. This is where the problem begins.

1. For v_1, `self.item == 1`, which is not `3`. We enter the `else` branch, where the loop (`for u in self.neighbours`) executes. The only neighbour is the vertex containing `2`, which we'll name v_2. We make a recursive call to `_Vertex.check_connected` on v_2 and `3`.
2. For v_2, `self.item == 2`, which is not `3`. We enter the `else` branch, where the loop (`for u in self.neighbours`) executes. The only neighbour is v_1. We make a recursive call to `_Vertex.check_connected` on v_1 and `3`.
3. For v_1, `self.item == 1`, which is not `3`. We enter the `else` branch...



Hey, wait a second! We're back to the first step, repeating a call to `_Vertex.check_connected` with arguments v_1 and `3`. That's not good, because we know what happens in this case: we recurse on v_2.

This is an instance of *infinite recursion*, meaning we've expressed a recursive computation that does not stop by reaching a base case. In our example, the Python interpreter alternates between calling `_Vertex.check_connected` with v_1 and `3` and with v_2 and `3`, and never stops.

From a technical point of view, each of these function calls adds a new stack frame onto the function call stack, which uses up more and more computer memory. To prevent your computer from running out of memory, the Python interpreter enforces a limit on the total number of stack frames that can be on the call stack at any one point in time—and when that limit is reached, the Python interpreter raises a `RecursionError`.

Okay, so now that we understand what the problem is, how do we fix it? Intuitively, we need to prevent repeated recursive calls to the same node. Before changing our code, let's first revisit our recursive definition of connectedness.

Given two vertices v_1 and v_2, they are connected when:

- $v_1 = v_2$, or
- there exists a neighbour u of v_1 such that u and v_2 are connected.

Our problem is in the recursive step. When we say that "u and v_2 are connected", we're allowing for the possibility that these two vertices are connected by a path going through v_1. But that's not necessary: if v_1 and v_2 are connected, then we should be able to find a path between a neighbour u and v_2 that doesn't use v_1, and then add v_1 to the start of that path.

So we can modify this definition as follows.

Definition 17 (connectivity, recursive, modified)**.** *Let $G = (V, E)$ be a graph, and let $v_1, v_2 \in V$. We say that v_1 and v_2 are connected when:*

- *$v_1 = v_2$, or*
- *there exists a neighbour u of v_1 such that u and v_2 are connected by a path that does not use v_1.*

One way to visualize is that in the recursive step, we should be able to remove v_1 from the graph and still find a path between u and v_2. And this is how we can take our original recursive definition and impose some structure on it: we take the original graph and make it smaller by "removing" the vertex v_1, and then recursively checking for the connectivity of each neighbour u and v_2.

Now, let's take this idea and apply it to fix our code. It might be tempting to actually mutate our `Graph` object to remove the "v_1" vertex at each recursive call, but we don't actually want to mutate the original graph when we call `Graph.connected`. So instead, we'll use a common technique when traversing graphs: keep track of the items that have been already visited by our algorithm, so that we don't visit the same vertex more than once.

Here is our new `_Vertex.check_connected` specification:

```
class _Vertex:
    def check_connected(self, target_item, visited):
        """Return whether this vertex is connected to a vertex corresponding to the target_item,
        WITHOUT using any of the vertices in visited.

        Preconditions:
        - self not in visited
        """
```

And before getting to our updated implementation, let's update how we call this helper method in `Graph.connected`. When we make the initial call to `_Vertex.check_connected`, we haven't yet visited any vertices:

```
class Graph:
    def connected(self, item1, item2):
        """..."""
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            return v1.check_connected(item2, set())  # Pass in an empty "visited" set
        else:
            return False
```

And finally, let's modify our implementation of `_Vertex.check_connected` to use this new parameter. We need to make two changes: first, we add `self` to `visited` before making any recursive calls (to indicate that the current `_Vertex` has been visited by our algorithm); second, when looping over `self.neighbours`, we only make recursive calls into nodes that have not yet been visited.

```
class _Vertex:
    def check_connected(self, target_item, visited):
        """Return whether this vertex is connected to a vertex corresponding to the target_item,
        WITHOUT using any of the vertices in visited.

        Preconditions:
        - self not in visited
        """
        if self.item == target_item:
            # Our base case: the target_item is the current vertex
            return True
        else:
            new_visited = visited.union({self})  # Add self to the set of visited vertices
            for u in self.neighbours:
                if u not in new_visited:  # Only recurse on vertices that haven't been visited
                    if u.check_connected(target_item, new_visited):
                        return True
            return False
```

And with this version, we've eliminated our infinite recursion error:

```
g = Graph()
g.add_vertex(1)
g.add_vertex(2)
g.add_vertex(3)
g.add_edge(1, 2)
print(g.connected(1, 3)) # False
```

One last thing: you might have noticed that in our final implementation of `_Vertex.check_connected`, we created a new set of visited vertices (`new_visited`) rather than mutating `visited`, which would have been simpler and more efficient. Consider this alternate version:

```
class _Vertex:
    def check_connected(self, target_item, visited):
        """Return whether this vertex is connected to a vertex corresponding to the target_item,
        WITHOUT using any of the vertices in visited.

        Preconditions:
        - self not in visited
        """
        if self.item == target_item:
            # Our base case: the target_item is the current vertex
            return True
```

```
        else:
            visited.add(self)  # Add self to the set of visited vertices
            for u in self.neighbours:
                if u not in visited:  # Only recurse on vertices that haven't been visited
                    if u.check_connected(target_item, visited):
                        return True
            return False
```

This call uses our familiar `set.add` method, which is indeed constant time. However, there is an important difference with this version: now there is only one `visited` set object that gets shared and mutated across all recursive calls, including ones that return `False`.

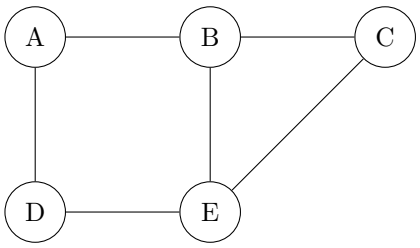
So for example, suppose `self` has two neighbours `u0` and `u1`, and we first recursive on `u0`. If `u0` is not connected to the target vertex, then the `check_connected` call will return `False`, as expected. But `u0` would still be in `visited` after that recursive call ends, meaning the subsequent `check_connected` call with `u1` will never visit `u0`. This is somewhat surprising, but technically still correct: once we've established that `u0` is not connected to the target item, we shouldn't recurse on it ever again.

But even though this implementation is correct, and much more efficient than the previous one, it does come with an important warning we want you to remember for the future. Whenever you use recursion with a mutable argument, be very careful when choosing whether to mutate that argument or create a modified copy—if you choose to mutate the argument, know that all recursive calls will mutate it as well!

Then you probably will start to think, how about I remove it if there are no connections?

```
class _Vertex:
    def __repr__(self):
        return f" {repr(self.item)}"

    def check_connected(self, target, visited):
        if self.item == target:
            return True
        else:
            visited.add(self)
            print(visited)
            # curr = visited.union({self}) # curr = copy.deepcopy(visited)
                                           # curr.add(self)
            for u in self.neighbours:
                if u not in visited:
                    if u.check_connected(target, visited):
                        return True
            visited.remove(self)
            return False
```

Consider this simple graph:



If both vertices exist, then this methods works exactly same to the previous implementation. But if one of the vertices does not exist, you will see the impact of `visited.remove(self)`:

```
g = Graph()
g.add_vertex(1)
g.add_vertex(2)
g.add_vertex(3)
g.add_vertex(4)
g.add_vertex(5)
```

```
g.add_edge(1, 2)
g.add_edge(1, 4)
g.add_edge(2, 3)
g.add_edge(2, 5)
g.add_edge(3, 5)
g.add_edge(4, 5)
print(g.connected(1, 6)) # False

# without visited.remove(self), every node will be visited at most once.
{ 1}
{ 2,  1}
{ 2,  5,  1}
{ 2,  5,  3,  1}
{ 5,  4,  2,  3,  1}
False

# with visited.remove(self), we actually explore all possible paths
{ 1}
{ 2,  1}
{ 2,  5,  1}
{ 2,  5,  3,  1}
{ 5,  4,  2,  3,  1}
{ 2,  1,  3}
{ 5,  2,  1,  3}
{ 4,  2,  1,  3}
{ 4,  1}
{ 4,  1,  3}
{ 4,  2,  1,  3}
{ 5,  4,  2,  1,  3}
{ 5,  4,  1,  3}
{ 5,  4,  2,  1,  3}
False
```

This is actually the idea of backtracking, you will explore further about backtracking in `CSC242` and `CSC243` later.

Now we have ability to explore all possible paths, could we write a function to find all possible paths? In the Graph class, you need to write a very simple function first:

```
class Graph:
    def all_paths(self, item1, item2):
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            return v1.paths(item2, set())
        else:
            raise ValueError
```

Then, we slightly modify the `check_coonected` function defined in `_Vertex` class to produce a new function which returns all paths:

```
class _Vertex:
    def paths(self, target, visited=None):
        # Add current node to visited set
        visited.add(self)
        # Initialize paths list
        paths = []
        # When target is found, add current path to paths
        if self.item == target:
            # Create path from items in visited nodes
            current_path = [node.item for node in visited]
            paths = [current_path]
        else:
            # Check all neighbors
            for u in self.neighbours:
```

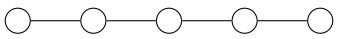
```
                if u not in visited:
                    # Directly extend paths with recursive results
                    paths.extend(u.paths(target, visited))

        # Remove current node from visited before returning (backtracking)
        visited.remove(self)

        # Return all paths found
        return paths
```

10.3 Cycles And Trees

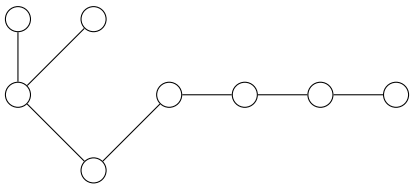
Graph connectivity is one of the most fundamental properties that a graph can have, and so it is useful in practice to be able to efficiently determine whether a graph is connected or not. In the previous section, we developed an algorithm for checking whether two vertices in a graph are connected. It is possible to extend this to an algorithm that checks whether an entire graph is connected, and a good exercise to do so!

Suppose we have a graph with n vertices. We'll start by asking the following question: *how many edges (in terms of n) must this graph have in order to be connected?* Or put another way, *what is the minimum number of edges required in a connected graph?*

We might consider some simple examples to gain some intuition here. For example, suppose we have a graph with n vertices which is just a path, as shown on the right. This has $n - 1$ edges, and if you remove any edge from it, the resulting graph will be disconnected.



But this isn't the only possible configuration for such a graph. The graph below certainly isn't a path, but removing any edge from this graph disconnects it as well. And you might notice that the number of edges is again one fewer than the number of vertices.



It turns out that these examples do give us the right intuition: any connected graph must have $|E| \geq |V| - 1$.

Definition 18. *Let $G = (V, E)$ be a graph. A **cycle** in G is a sequence of vertices $v_0, \ldots, v_k$ satisfying the following conditions:*

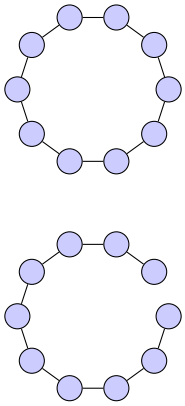
- *$k \geq 3$, $v_0 = v_k$, and all other vertices are distinct from each other and v_0*
- *each consecutive pair of vertices is adjacent*
- a cycle is like a path, except it starts and ends at the same vertex
- The length of a cycle is the number of edges used by the sequence, which is also the number of distinct vertices in the sequence.
- Cycles must have length at least three
- Two adjacent vertices are not considered to form a cycle.

Getting back to our motivation, cycles are a form of **connectedness redundancy** in a graph. Vertices in a cycle are all obviously connected to each other, but even if one edge is removed, the result is a path. In this case, the cycle's vertices are still connected to each other—albeit with possibly a much longer path to travel.

Definition 19. *Let $G = (V, E)$ be a graph and $e \in E$. If G is connected and e is in a cycle of G, then the graph obtained by removing e from G is still connected.*

The previous lemma told us that if we have a connected graph with a cycle, it is always possible to remove an edge from the cycle and still keep the graph connected. Since we are interested in talking about the minimum number of edges necessary for connecting a graph, we'll now think about graphs which don't have any cycles.

Definition 20. *Let $G = (V, E)$ be a graph. We say that G is a **tree** when it is connected and has no cycles.*



Wait a second, we already defined trees before! It turns out that **we can view trees as just a special type of graph, one that has no cycles**. The one subtle difference is that the trees we studied earlier had a root, representing the "start" or "top" of a tree. With this graph-based definition of trees, we don't have a designated root element, and won't necessarily draw trees top-down, as illustrated by the diagram on the right.

Now, let us return to our original motivation of counting edges to prove the following remarkable result, which says that the number of edges in a tree depends only on the number of vertices.

Definition 21 (Number of edges in a tree)**.** *Let $G = (V, E)$ be a tree. Then $|E| = |V| - 1$.*

Definition 22. *Let $G = (V, E)$ be a tree. If $|V| \geq 2$, then G has a vertex with degree one.*

Combining everything together, we can conclude the following required number of edges for any connected graph. Since every connected graph contains at least one tree (just keep removing edges in cycles until you cannot remove any more), this constraint on the numbers of edges in a tree translates immediately into a lower bound on the number of edges in any connected graph (in terms of the number of vertices of that graph).

Definition 23. *Let $G = (V, E)$ be a graph. If G is connected, then $|E| \geq |V| - 1$.*

So for example, suppose we have 100 cities that we're connecting by airplane routes, and we want to be able to travel from one city to any other city (possibly by taking multiple flights). In this case, our previous theorem tells us that we'll need at least 99 routes between cities to make this possible. We might, of course, decide to open more routes, but we cannot open fewer without disconnecting cities from each other!

```python
class Graph:
    def connected_graph(self):
        """Determine if the graph is connected using the edge count property.

        A connected graph with |V| vertices must have at least |V|-1 edges.
        This is a necessary but not sufficient condition - we still need to verify
        that all vertices are actually reachable from a single source.

        Returns:
            bool: True if the graph is connected, False otherwise.
        """
        # Get the number of vertices and edges
        num_vertices = len(self._vertices)

        # Handle empty graph case
        if num_vertices == 0:
            return True

        # Count edges (divide by 2 since each edge is counted twice in an undirected graph)
        num_edges = sum(len(v.neighbours) for v in self._vertices.values()) // 2

        # Check necessary condition: |E| >= |V|-1
        if num_edges < num_vertices - 1:
            # If we have fewer edges than |V|-1, the graph cannot be connected
            return False
        else:
            return True
```

10.4 Spanning Tree

We saw that given a graph $G = (V, E)$, if G is a connected graph, then $|E| \geq |V| - 1$. We said that we could take any connected graph and repeatedly remove edges that are part of a cycle until we end up with a tree—a connected graph with no cycles.

One computational question that arises naturally from this analysis is how can we design and implement an algorithm that finds such a tree. In this section, we'll tie together both theoretical and computational threads in this chapter to develop an algorithm that accomplishes this efficiently.

Definition. Let $G = (V, E)$ be a *connected* graph. Let $G' = (V, E')$ be another graph with the same vertex set as G, and where $E' \subseteq E$, i.e., its edges are a subset of the edges of G. We say that G' is a **spanning tree** of G when G' is a tree.

Every connected graph has at least one spanning tree, but can generally have more than one. In fact, it is possible to prove that a connected graph G has a *unique* spanning tree if and only if G is itself a tree.

Our goal is to implement a `Graph` method that computes a spanning tree for the graph. To keep things simple, we'll focus here on just returning a *subset of the edges*, although we could modify our approach to return a brand-new `Graph` object as well.

```
class Graph:
    def spanning_tree(self):
        """Return a subset of the edges of this graph that form a spanning tree.
        The edges are returned as a list of sets, where each set contains the two
        Vertices corresponding to an edge. Each returned edge is in this graph
        (i.e., this function doesn't create new edges!).

        Preconditions:
        - this graph is connected
        """
```

10.4.1 A brute force approach

The argument we gave at the end of the last section translates naturally into an algorithm for computing a spanning tree of a graph. Given a connected graph, we can start with all of its edges and then try finding a cycle. If we find a cycle, remove one of its edges, and repeat until there are no more cycles. Here is some pseudocode for this approach:

```
class Graph:
    def spanning_tree(self) -> list[set]:
        edges_so_far = all edges in self
        while edges_so_far contains a cycle:
            remove an edge in the cycle from edges_so_far
        return edges_so_far
```

While this approach is correct, it's quite inefficient, as finding a cycle (the while loop condition) is relatively complex. It turns out that we can do better by taking a similar approach to the `Graph.connected` and `_Vertex.check_connected` methods.

To start, recall our `_Vertex.check_connected` method:

```
class _Vertex:
    def check_connected(self, target_item, visited):
        """Return whether this vertex is connected to a vertex corresponding to the target_item,
        WITHOUT using any of the vertices in visited.

        Preconditions:
        - self not in visited
        """
        if self.item == target_item:
            # Our base case: the target_item is the current vertex
            return True
        else:
            visited.add(self)  # Add self to the set of visited vertices
            for u in self.neighbours:
```

```
                if u not in visited:  # Only recurse on vertices that haven't been visited
                    if u.check_connected(target_item, visited):
                        return True
            return False
```

We used this method to check whether a given vertex was connected to some target vertex. This implementation uses a loop with an early return, stopping if one of the recursive calls finds the target vertex. However, suppose we drop the early return, or are searching for a vertex that `self` isn't connected to. In this case, we expect that this method will recurse on every vertex that `self` is connected to.

This means that we can take this same code structure and modify it to, for example, print out all of the vertices that `self` is connected to:

```
class _Vertex:
    def print_all_connected(self, visited):
        """Print all items that this vertex is connected to, WITHOUT using any of the vertices in visited.

        Preconditions:
        - self not in visited
        """
        print(self.item)
        visited.add(self)
        for u in self.neighbours:
            if u not in visited:  # Only recurse on vertices that haven't been visited
                u.print_all_connected(visited)
```

Here's an example of this method in action:

```
g = Graph()
for i in range(0, 5):
    g.add_vertex(i)
g.add_edge(0, 1)
g.add_edge(0, 3)
g.add_edge(1, 2)
g.add_edge(1, 4)
g.add_edge(2, 4)
g._vertices[0].print_all_connected(set())
>>> 0 1 4 2 3
```

Okay, we can see all of the items printed out, so that's a start. But how do we turn this into a spanning tree?

Let's make one modification to our code to make it easier to see what's going on. We'll use the same technique that we saw when printing out a Tree in previous tree chapter and add some indentation to our output.

```
class _Vertex:
    def print_all_connected_indented(self, visited: set[_Vertex], d: int) -> None:
        """Print all items that this vertex is connected to, WITHOUT using any of the vertices
        in visited.

        Print the items with indentation level d.

        Preconditions:
        - self not in visited
        - d >= 0
        """
        print(' ' * d + str(self.item))
        visited.add(self)
        for u in self.neighbours:
            if u not in visited:  # Only recurse on vertices that haven't been visited
                u.print_all_connected_indented(visited, d + 1)
```

By increasing this depth parameter `d`, we can now see the recursive call structure that this method traces:

```
>>> # Using same graph g as above
>>> g._vertices[0].print_all_connected_indented(set(), 0)
0
 1
  2
  4
 3
```

So this is our key insight: the recursive call structure of `_Vertex.check_connected` forms a tree that spans all of the vertices that the starting vertex is connected to.

10.4.2 A spanning tree algorithm

We can now use this insight to compute a spanning tree of a connected graph. Essentially, what we need to do is convert our recursive traversal algorithm from one that prints items to one that returns a list of edges.

There are two key changes we'll need to make:

1. Each recursive call will now return a list of edges, so we'll need an accumulator to keep track of them.
2. To "handle the root", we'll also need to add edges between `self` and each vertex where we make a recursive call.

```
class _Vertex:
    def spanning_tree(self, visited):
        """Return a list of edges that form a spanning tree of all vertices that are
        connected to this vertex WITHOUT using any of the vertices in visited.

        The edges are returned as a list of sets, where each set contains the two
        ITEMS corresponding to an edge.

        Preconditions:
        - self not in visited
        """
        edges_so_far = []
        visited.add(self)
        for u in self.neighbours:
            if u not in visited:  # Only recurse on vertices that haven't been visited
                edges_so_far.append({self.item, u.item})
                edges_so_far.extend(u.spanning_tree(visited))
        return edges_so_far
```

And we can call this method as follows:

```
g._vertices[0].spanning_tree(set())
[{0, 3}, {0, 1}, {1, 4}, {1, 2}]
```

Finally, we can use this method as a helper to implement our original `Graph.spanning_tree` method. But how will we choose a starting vertex? We can choose any starting vertex we want, because as long as we assume our graph is connected, any vertex we pick will be connected to every other vertex.

```
class Graph:
    def spanning_tree(self):
        """Return a subset of the edges of this graph that form a spanning tree.
        The edges are returned as a list of sets, where each set contains the two
        ITEMS corresponding to an edge. Each returned edge is in this graph
        (i.e., this function doesn't create new edges!).

        Preconditions:
        - this graph is connected
```

```
        """
        # Pick a vertex to start
        all_vertices = list(self._vertices.values())
        start_vertex = all_vertices[0]
        # Use our helper _Vertex method!
        return start_vertex.spanning_tree(set())
```

The idea of spanning tree has many real-world applications:

- **Network design optimization**: In telecommunications, power grids, and transportation networks, spanning trees help design networks with minimal infrastructure while ensuring all points are connected.
- **Control flow graph**: In build systems and package managers, spanning trees help resolve dependencies efficiently.

11 Dictionary

We'll now explore an approach to dictionary implementation.

Consider a simple scenario where all key-value pairs have keys that are integers from 0 to $K-1$. In this case, we could store these pairs in an array of length K, achieving constant time search, insertion, and deletion by simply storing each value at the array index matching its key.

This approach, called **direct addressing**, works efficiently when the number of possible keys is small. However, since array addressing is only constant time when the entire array is allocated in memory, as K increases, the space requirement grows linearly with K, even if operations remain constant time. This space constraint makes direct addressing impractical for many real-world applications.

11.1 Hash Function

Hashing, a generalized version of direct addressing that allows us to store key-value pairs with arbitrary keys using significantly less space than would be required to accommodate all possible keys.

Let U denote the set of all possible keys we might want to store. These keys aren't necessarily numeric—they could be strings, floating-point numbers, or custom data types defined by our application. Like direct addressing, we'll store these keys in an array, but with size m, which may differ from the size of U.

Definition 24 (Hash Function, Hash Table)**.** *A hash function is a function $h : U \to \{0, 1, \ldots, m-1\}$ that takes a key and calculates the array position where that key should be placed.*

A hash table is a data structure consisting of an array of length m and a hash function.

The basic operations can be implemented as follows:

```
def Search(hash_table, key):
    hash = h(key)
    return hash_table[hash]

def Insert(hash_table, key, value):
    hash = h(key)
    hash_table[hash] = value

def Delete(hash_table, key):
    hash = h(key)
    hash_table[hash] = None
```

While this implementation is efficient—all operations run in $\Theta(1)$ time—it isn't guaranteed to be correct. We haven't made assumptions about the relationship between m and $|U|$, nor have we specified what h actually does. **Our implementation assumes no collisions in the hash function, meaning each key maps to a unique array position.** If two keys hash to the same index i, the second insertion would overwrite the first value.

If $m \geq |U|$, there always exists a hash function without collisions, called a **perfect hash function**. Such functions allow our naive implementation to work correctly. However, finding and computing perfect hash functions can be non-trivial for large sets of keys.

In practice, the assumption that $m \geq |U|$ is often unrealistic, as we typically expect the number of possible keys to be extremely large while keeping the array size reasonable. If $m < |U|$, at least one collision is guaranteed, causing our implementation to fail.

11.2 Closed Addressing

In **closed addressing**, each element in the array doesn't directly store a key-value pair. Instead, it contains a pointer to a linked list that holds multiple key-value pairs. This approach, also called "**chaining**," elegantly handles collisions by simply adding both the key and its associated value to the linked list at the calculated array position. The key must be stored alongside the value to enable proper identification during searches.

The search and deletion operations are slightly more complex than insertion, as they require traversing the linked list to locate the specific key-value pair.

```
def Search(hash_table, key):
    hash = h(key)
    linked_list = hash_table[hash]
    if linked_list contains key:
        return corresponding value from linked_list
```

```
    else:
        return None

def Insert(hash_table, key, value):
    hash = h(key)
    linked_list = hash_table[hash]
    if linked_list contains key: (Search function)
        update the value associated with key in linked_list
    else:
        insert key, value at the head of linked_list

def Delete(hash_table, key):
    hash = h(key)
    linked_list = hash_table[hash]
    delete key from linked_list
```

This property ensures correctness. Since the algorithms only examine one specific linked list determined by the hash function, they can be quite efficient.

However, performance issues arise if numerous keys map to the same index. In such cases, search and deletion operations degrade to the worst-case performance of linked list operations: $\Theta(n)$, where n represents the number of entries in the hash table. Unfortunately, we cannot design a hash function that completely prevents this situation. For any hash function h, when the universe of possible keys significantly exceeds the array size, there must exist at least one index to which multiple keys map.

This type of counting argument is extremely common in algorithm analysis. If you have a set of numbers whose average is x, then one of the numbers must have value $\geq x$.

The time complexity of the `Search` operation can be broken down into two steps:

1. Computing the hash value: $O(1)$ assuming a well-designed hash function.
2. Searching for the key in the linked list: $O(L)$ where L is the length of the linked list at index $h(key)$.

The total time complexity is therefore $O(1 + L)$, which simplifies to $O(L)$.

In the best case, either:

- The linked list at index $h(key)$ is empty: $O(1)$
- The key is at the head of the linked list: $O(1)$

In the worst case, all n keys in the hash table are mapped to the same index, creating a linked list of length n. The Search operation then degrades to a linear search through this linked list:

- Worst case time complexity: $O(n)$

For a hash table with m array slots and n keys, assuming uniform hashing (each key is equally likely to hash to any slot), the average length of a linked list is $\alpha = n/m$, which is called the load factor.

- Average case time complexity: $O(1 + \alpha) = O(1 + n/m)$

If we maintain α as a constant by resizing the hash table when necessary, the average time complexity of Search is $O(1)$, which makes hash tables very efficient for lookup operations.

11.3 Open Addressing

Open addressing is an alternative collision resolution strategy that requires each array element to contain only a single key, while allowing keys to be assigned to different indices when their initial position is already taken. This approach eliminates the space overhead needed for linked list references in closed addressing, but it imposes the constraint that the load factor α must remain below 1, meaning the hash table cannot store more keys than its array length.

In this hashing approach, we utilize a parameterized hash function h that accepts two parameters: a key and a positive integer. For a key k, the initial hash value is $h(k, 0)$, and if this position is occupied, subsequent positions are determined by $h(k, 1)$, $h(k, 2)$, and so on.

For implementation simplicity, we've omitted boundary checks ($i < m$) that would be necessary in practice to prevent infinite loops.

```
def Insert(hash_table, key, value):
    i = 0
    while hash_table[h(key, i)] is not None:
        i = i + 1
```

```
    hash_table[h(key, i)].key = key
    hash_table[h(key, i)].value = value
```

Searching requires examining multiple positions rather than just one. When searching for key k, we traverse the same sequence of indices $h(k, 0)$, $h(k, 1)$, etc. until either finding the key or encountering a None value.

```
def Search(hash_table, key):
    i = 0
    while hash_table[h(key, i)] is not None and hash_table[h(key, i)].key != key:
        i = i + 1

    if hash_table[h(key, i)].key == key:
        return hash_table[h(key, i)].value
    else:
        return None
```

Deletion is more complex than simply replacing an item with None. Consider a situation where key k is stored at $h(k, 1)$ because $h(k, 0)$ was occupied. If we later delete the item at $h(k, 0)$ and replace it with None, subsequent searches for k would stop at $h(k, 0)$ upon seeing None, never checking $h(k, 1)$.

To address this issue, we use a special `Deleted` marker instead of None when removing items. This allows the Search algorithm to continue past deleted positions. Modifying the Insert algorithm to utilize these `Deleted` positions is necessary.

```
def Delete(hash_table, key):
    i = 0
    while hash_table[h(key, i)] is not None and hash_table[h(key, i)].key != key:
        i = i + 1

    if hash_table[h(key, i)].key == key:
        hash_table[h(key, i)] = Deleted
```

Further discussion is needed regarding the implementation of different parameterized hash functions and the properties of their generated probe sequences $[h(k, 0), h(k, 1), \ldots]$. **A key question is how probe sequences for different keys overlap and how these overlaps affect open addressing efficiency, since all dictionary operations require traversing these sequences until reaching an empty position.**

11.3.1 Linear Probing

Linear probing is the most straightforward implementation of open addressing. This technique begins at a specific hash value and repeatedly adds a fixed offset to the index until finding an unoccupied position.

Definition 25 (Linear probing)**.** *Given a hash function* $hash : U \to \{0, \ldots, m-1\}$ *and a constant* $b \in \{1, \ldots, m-1\}$*, we define the parameterized hash function* h *for linear probing as:*

$$h(k, i) = (hash(k) + b \times i) \bmod m.$$

For a key k, the linear probing sequence becomes $h(k, 0), h(k, 1), h(k, 2), \ldots$ When $b = 1$, this sequence simplifies to $\text{hash}(k), \text{hash}(k) + 1, \ldots$

However, linear probing has a significant drawback known as **clustering**. This phenomenon occurs when adjacent occupied locations form continuous blocks, causing subsequent insertions into these areas to become increasingly inefficient. For instance, if $b = 1$ and three keys with the same hash value 10 are inserted, they will occupy positions 10, 11, and 12. Consequently, any new key with hash value 11 will require checking two already occupied positions before finding an empty slot.

The collision resolution for hash value 10 effectively creates collisions for hash values 11 and 12, increasing operation time for any access to these positions. This effect compounds because each collision with any position in a cluster extends the cluster by one element.

11.3.2 Quadratic Probing

Linear probing's primary weakness is that hash values occurring within a cluster follow identical search patterns as those at the cluster's beginning. This results in increasingly more keys being incorporated into elongated search patterns as clusters expand. Quadratic probing offers a solution by causing the gap between consecutive indices in the probe sequence to widen as the sequence progresses.

Definition 26 (Quadratic Probing)**.** *For a hash function* $hash : U \to \{0, \ldots, m-1\}$ *and values* $b, c \in \{0, \ldots, m-1\}$*, the parameterized hash function* h *for quadratic probing can be defined as:*

$h(k, i) = hash(k) + bi + ci^2 \pmod{m}$

When a key's hash value falls within a cluster, quadratic probing allows it to "break free" from the cluster more efficiently than linear probing. Nevertheless, a different form of clustering persists: multiple items sharing the same initial hash value will still follow identical probe sequences. For instance, if four keys hash to the same index, the fourth key must examine (and bypass) the three positions already occupied by the first three keys.

11.3.3 Double Hashing

A fundamental challenge with standard collision resolution methods is that keys that initially hash to the same location will follow identical probe sequences. Double hashing addresses this issue by calculating the step size based on the key itself, creating a more sophisticated approach.

Double hashing utilizes two separate hash functions. Given hash functions hash_1 and hash_2, both mapping from the universe U to the set $\{0, 1, \ldots, m-1\}$, we define the parameterized hash function h for double hashing as:

Definition 27 (Double Hashing)**.** *Given two hash functions* $hash_1, hash_2 : U \to \{0, \ldots, m-1\}$*, we can define the parameterized hash function for double hashing* h *as follows:*

$$h(k, i) = hash_1(k) + i \cdot hash_2(k) \pmod{m}.$$

This offers a significant advantage over linear and quadratic probing. While those methods produce at most m distinct probe sequences (determined by the initial hash value), double hashing can generate up to m^2 different sequences—one for each possible pair of values $(\text{hash}_1(k), \text{hash}_2(k))$. This dramatically reduces the likelihood of cluster formation, as colliding keys would need to share both the same initial hash value and the same offset value.

11.3.4 Runtime Analysis of Open Addressing

When analyzing open addressing performance, it's worth examining the runtime characteristics. There exist $m!$ possible probe sequences (the order in which table indices are examined). A common simplification assumes each sequence occurs with equal probability ($\frac{1}{m!}$).

When we talk about probe sequences in open addressing hash tables:

1. A probe sequence is the specific order in which we check positions in the hash table when looking for an item or an empty slot.
2. In a hash table with size m, there are m possible positions to examine.
3. The $m!$ (m factorial) refers to all possible permutations of these m positions. For example, in a hash table of size 3, you could check positions in these orders:
   - $[0, 1, 2]$
   - $[0, 2, 1]$
   - $[1, 0, 2]$
   - $[1, 2, 0]$
   - $[2, 0, 1]$
   - $[2, 1, 0]$
4. That's $3! = 6$ different possible sequences.

However, the actual hashing methods like linear probing, quadratic probing, or double hashing don't generate all possible permutations. They follow specific patterns:

- Linear probing always checks consecutive positions
- Double hashing only generates about m^2 different sequences

The simplified analysis assumes each probe sequence is equally likely (probability $\frac{1}{m!}$), which is an approximation that makes the mathematical analysis more tractable.

Under this assumption, all three dictionary operations require searching an average of at most $\frac{1}{1-\alpha}$ indices (search until find an empty slot). The factor α (alpha) represents the load factor of the hash table - the proportion of slots that are filled:

- $\alpha = \frac{n}{m}$ where n is the number of elements and m is the table size

- When $\alpha = 0$, the table is empty
- When α approaches 1, the table is nearly full

The expected number of probes before finding an empty slot follows a geometric distribution which is $\frac{1}{1-\alpha}$, applies to all three dictionary operations (search, insert, and delete) under the uniformly random probe sequence assumption.

For instance, in a table with 90% occupancy ($\alpha = 0.9$), you'd expect to search only 10 indices on average, which represents excellent performance.

To understand why it is a geometric distribution, think if the chance of "success" (here: hitting an empty slot) is p, the expected number of trials until success is:

$$\mathbb{E}[X] = \frac{1}{p}$$

In our case:

- Success = finding an empty slot
- Probability of success = $1 - \alpha$

So:

$$\text{Expected probes} = \frac{1}{1-\alpha}$$

11.4 Linked List Implementation

`key` $\rightarrow$ Hash Function $\rightarrow$ Array Index

Internally:

```
HashTable
|-- bucket array
|   |-- linked list
|   |-- linked list
|   \-- linked list
```

Collisions occur when multiple keys map to the same index.

One common solution is called:

Separate Chaining

Every bucket is a `LinkedList`, and each node in that linked list stores one tuple of the form `(key, value)`. This design makes the hash table compatible with our existing linked-list implementation. Conceptually, the hash table still works the same way as before: it uses hashing to decide which bucket to use, and collisions are handled by separate chaining.

This first part imports the linked-list implementation and defines the constructor of the hash table. The constructor keeps track of the table capacity, the number of stored key-value pairs, the resize threshold, and the bucket array. Since our `LinkedList` constructor expects an iterable, each bucket is initialized as `LinkedList([])` rather than `None`. This gives us an empty linked list in every bucket from the beginning.

```python
from linkedlist import LinkedList


class HashTable:
    def __init__(self, capacity=8):
        self.capacity = capacity
        self.size = 0
        self.load_factor_threshold = 0.75

        # Each bucket is one LinkedList object
        self.buckets = [LinkedList([]) for _ in range(capacity)]

    def _hash(self, key):
        return hash(key) % self.capacity
```

The helper method `_find_node()` is the core of the implementation. After the hash function computes the bucket index, this method walks through the linked list in that bucket and searches for the target key. It returns four values: the index of the bucket, the bucket itself, the previous node, and the current node. Returning both `prev` and `curr` is especially useful when we later want to delete a key, because deletion in a singly linked list needs access to the previous node.

```
    def _find_node(self, key):
        """
        Return (index, bucket, prev, curr)
        where curr is the node containing key, or None if not found.
        """
        index = self._hash(key)
        bucket = self.buckets[index]

        prev = None
        curr = bucket.head

        while curr is not None:
            stored_key, stored_value = curr.item
            if stored_key == key:
                return index, bucket, prev, curr
            prev = curr
            curr = curr.next

        return index, bucket, prev, None
```

The method `__setitem__()` supports syntax such as `table[key] = value`. It first calls `_find_node()` to see whether the key already exists. If the key is already present, we simply replace the stored tuple with a new tuple containing the updated value. If the key is not present, we append a new `(key, value)` pair to the linked list in that bucket. After every successful insertion of a new key, we increment the size and check whether resizing is needed.

```
    def __setitem__(self, key, value):
        index, bucket, prev, curr = self._find_node(key)

        # Update existing key
        if curr is not None:
            curr.item = (key, value)
            return

        # Insert new key-value pair
        bucket.append((key, value))
        self.size += 1

        if self.size / self.capacity > self.load_factor_threshold:
            self._resize()
```

The method `__getitem__()` supports syntax such as `value = table[key]`. It again uses `_find_node()` so that the search logic is centralized in one place. If the key is found, the method returns the second component of the stored tuple, which is the value. If the key does not exist, it raises `KeyError`, which matches normal Python dictionary behavior.

```
    def __getitem__(self, key):
        index, bucket, prev, curr = self._find_node(key)

        if curr is None:
            raise KeyError(key)

        return curr.item[1]
```

Deletion is slightly more delicate because we must reconnect the linked list correctly. If the node to be deleted is the first node in the bucket, then the bucket head should move to `curr.next`. Otherwise, we make the previous node skip

the current node. We also need to update the tail reference when the removed node is the last node. Finally, if the list becomes empty after deletion, both `head` and `tail` must be set to `None`.

```
    def __delitem__(self, key):
        index, bucket, prev, curr = self._find_node(key)

        if curr is None:
            raise KeyError(key)

        # Remove curr from the linked list
        if prev is None:
            bucket.head = curr.next
        else:
            prev.next = curr.next

        # Fix tail if needed
        if curr == bucket.tail:
            bucket.tail = prev

        if bucket.head is None:
            bucket.tail = None

        self.size -= 1
```

These methods make the hash table feel more like a built-in Python container. The method `__contains__()` supports expressions such as `key in table`. The method `__len__()` returns the number of key-value pairs. The iterator method `__iter__()` yields all keys in the table by traversing every bucket and every linked-list node inside each bucket. The methods `keys()`, `values()`, and `items()` then build on this behavior.

```
    def __contains__(self, key):
        index, bucket, prev, curr = self._find_node(key)
        return curr is not None

    def __len__(self):
        return self.size

    def __iter__(self):
        for bucket in self.buckets:
            curr = bucket.head
            while curr is not None:
                yield curr.item[0]
                curr = curr.next

    def keys(self):
        return iter(self)

    def values(self):
        for bucket in self.buckets:
            curr = bucket.head
            while curr is not None:
                yield curr.item[1]
                curr = curr.next

    def items(self):
        for bucket in self.buckets:
            curr = bucket.head
            while curr is not None:
                yield curr.item
                curr = curr.next
```

As more elements are inserted, too many keys may accumulate in too few buckets, which makes the chains longer and reduces efficiency. To prevent this, we resize the table whenever the load factor exceeds the threshold. Resizing

doubles the number of buckets, creates a new list of empty linked lists, resets the size, and reinserts every old key-value pair. Reinsertion is necessary because changing the capacity changes the result of the hash function modulo operation.

```
    def _resize(self):
        old_buckets = self.buckets

        self.capacity *= 2
        self.buckets = [LinkedList([]) for _ in range(self.capacity)]
        self.size = 0

        for bucket in old_buckets:
            curr = bucket.head
            while curr is not None:
                key, value = curr.item
                self[key] = value
                curr = curr.next
```

These methods are not strictly required for the hash table to work, but they make it easier to use and debug. The `__repr__()` method creates a readable string similar to a Python dictionary. The `get()` method returns a default value instead of raising an error. The `put()` and `remove()` methods are optional convenience wrappers that call the special methods internally.

```
    def __repr__(self):
        pairs = [f"{key!r}: {value!r}" for key, value in self.items()]
        return "{" + ", ".join(pairs) + "}"

    def get(self, key, default=None):
        try:
            return self[key]
        except KeyError:
            return default

    def put(self, key, value):
        self[key] = value

    def remove(self, key):
        del self[key]
```

The following test program checks the major behaviors of the hash table. First, it inserts several elements. Then it tests lookup, update, membership, deletion, iteration, and resizing. I intentionally start with a small capacity so that resizing happens quickly. If all tests pass, the printed output should confirm that the implementation is working correctly.

```
def run_tests():
    print("Creating hash table...")
    table = HashTable(capacity=2)

    print("\nInserting values...")
    table["apple"] = 3
    table["banana"] = 5
    table["orange"] = 7
    table["grape"] = 9

    print("Current table:", table)
    print("Current size:", len(table))
    print("Current capacity:", table.capacity)

    print("\nTesting lookup...")
    print("table['apple'] =", table["apple"])
    print("table['banana'] =", table["banana"])
    print("table.get('orange') =", table.get("orange"))
```

```
    print("table.get('missing', 100) =", table.get("missing", 100))

    print("\nTesting update...")
    table["apple"] = 100
    print("Updated table['apple'] =", table["apple"])

    print("\nTesting membership...")
    print("'banana' in table:", "banana" in table)
    print("'melon' in table:", "melon" in table)

    print("\nTesting iteration over keys...")
    for key in table:
        print(key)

    print("\nTesting items()...")
    for key, value in table.items():
        print(key, value)

    print("\nTesting deletion...")
    del table["banana"]
    print("After deleting 'banana':", table)
    print("Current size:", len(table))

    print("\nTesting remove()...")
    table.remove("orange")
    print("After removing 'orange':", table)
    print("Current size:", len(table))

if __name__ == "__main__":
    run_tests()
```

Operation	Average Case	Worst Case
Insert	$O(1)$	$O(n)$
Search	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

Worst case happens when all keys collide into one bucket.

—

11.5 Open Addressing Implementation

Modern programming languages usually do **not** use chaining.

Python dictionaries instead use:

Open Addressing

All elements remain inside a single array.

If collision occurs, probing searches for another slot.

—

$$index = (hash + i) \bmod capacity$$

Example probing sequence:

$$h,\ h+1,\ h+2,\ h+3$$

—

All key-value pairs are stored directly inside arrays, and collisions are handled by searching for another available slot. In this code, the collision strategy is **linear probing**, which means that if one slot is occupied, the algorithm checks the next slot, then the next one, and so on, wrapping around when necessary. This design avoids linked lists, but it requires careful handling of insertion, lookup, deletion, and resizing.

The first part defines the class and two sentinel objects: `EMPTY` and `DELETED`. A slot marked `EMPTY` has never been used before. A slot marked `DELETED` used to hold a key, but that key was removed later. These two markers are very important in open addressing, because lookup and insertion behave differently when they see a slot that was never used versus a slot that was deleted.

```
class OpenAddressHashTable:
    # Sentinel object marking a slot that has never been used
    EMPTY = object()

    # Sentinel object marking a slot that previously held a key
    # but was later deleted
    DELETED = object()
```

The constructor initializes the table with a fixed capacity, a size counter, and two parallel arrays: one for keys and one for values. At the beginning, every slot in the key array is marked as `EMPTY`, and the corresponding value slots are set to `None`. The code also defines a load-factor threshold. Once the table becomes too full, it will need to grow larger so that operations remain efficient.

```
    def __init__(self, capacity=8):
        # Total number of slots in the table
        self.capacity = capacity

        # Number of active key-value pairs currently stored
        self.size = 0

        # Parallel arrays:
        # - keys[i] stores the key in slot i
        # - values[i] stores the corresponding value
        self.keys = [self.EMPTY] * capacity
        self.values = [None] * capacity

        # Resize threshold for open addressing
        self.load_factor_threshold = 0.7

    def _hash(self, key):
        # Compute the starting slot for a given key
        return hash(key) % self.capacity
```

The method `_find_slot()` is the core of the entire implementation. It searches for a slot using linear probing. If the method is being used for lookup, it tries to find the exact key and returns `None` if the key does not exist. If it is being used for insertion, it searches for the best location to place a new key. During insertion, the algorithm remembers the first deleted slot it sees, because a deleted slot may be reused later. This is why the method has the parameter `for_insert`.

```
    def _find_slot(self, key, for_insert=False):
        """
        Find the slot index for a key.

        If for_insert is False:
            - return the index where the key is found
            - return None if the key does not exist

        If for_insert is True:
            - return the best slot for insertion
            - if the key already exists, return its current slot
            - prefer the first DELETED slot encountered if no exact key match appears

        Linear probing is used:
            j = (start + i) % capacity
        """
        start = self._hash(key)
        first_deleted = None
```

```
        for i in range(self.capacity):
            j = (start + i) % self.capacity
            current_key = self.keys[j]

            # If we reach an EMPTY slot:
            # - lookup can stop because the key was never inserted beyond this point
            # - insertion can use this slot, unless we saw a DELETED slot earlier
            if current_key is self.EMPTY:
                if for_insert:
                    return first_deleted if first_deleted is not None else j
                return None

            # Remember the first deleted slot for possible insertion,
            # but continue probing in case the key already exists later.
            if current_key is self.DELETED:
                if for_insert and first_deleted is None:
                    first_deleted = j
                continue

            # If the key is found, return its slot
            if current_key == key:
                return j

        # If inserting and the table contains no EMPTY slots,
        # we may still reuse a deleted slot.
        if for_insert and first_deleted is not None:
            return first_deleted

        # No valid slot found
        return None
```

The method `__setitem__()` allows syntax such as `table[key] = value`. It first calls `_find_slot()` in insertion mode. If the key already exists, the returned slot will be that existing key, so the code simply overwrites the value. If the slot is new or previously deleted, the size increases. After insertion, the code checks the load factor and resizes the table if necessary.

```
    def __setitem__(self, key, value):
        """
        Insert a new key-value pair or update an existing key.

        Enables syntax like:
            table[key] = value
        """
        index = self._find_slot(key, for_insert=True)

        # If no slot is available, resize and try again
        if index is None:
            self._resize()
            index = self._find_slot(key, for_insert=True)

        # If inserting into a new or deleted slot, increase size
        if self.keys[index] in (self.EMPTY, self.DELETED):
            self.size += 1

        self.keys[index] = key
        self.values[index] = value

        # Resize if the table becomes too full
        if self.size / self.capacity > self.load_factor_threshold:
            self._resize()
```

The lookup method `__getitem__()` allows syntax such as `value = table[key]`. It calls `_find_slot()` in normal

lookup mode. If the key is found, the method returns the corresponding value from the value array. If the key is not present, it raises `KeyError`, which is the standard behavior expected from Python mappings.

```
    def __getitem__(self, key):
        """
        Retrieve the value associated with a key.

        Enables syntax like:
            value = table[key]

        Raises:
            KeyError: if the key does not exist
        """
        index = self._find_slot(key, for_insert=False)

        if index is None:
            raise KeyError(key)

        return self.values[index]
```

Deletion is more subtle in open addressing than in separate chaining. We cannot simply mark a slot as empty when removing a key, because that might break the probing chain for other keys that were inserted later. Instead, we mark the slot as `DELETED`. This tells future lookups to continue probing past that slot, while also allowing future insertions to reuse that position if appropriate.

```
    def __delitem__(self, key):
        """
        Remove a key-value pair by key.

        Enables syntax like:
            del table[key]

        Raises:
            KeyError: if the key does not exist
        """
        index = self._find_slot(key, for_insert=False)

        if index is None:
            raise KeyError(key)

        # Mark the slot as deleted so probing chains remain valid
        self.keys[index] = self.DELETED
        self.values[index] = None
        self.size -= 1
```

These methods make the table feel like a normal Python container. The method `__contains__()` supports expressions such as `key in table`. The method `__len__()` returns the number of active key-value pairs currently stored. Notice that deleted slots do not count toward the size, because those key-value pairs are no longer active.

```
    def __contains__(self, key):
        """
        Check whether a key exists in the table.

        Enables syntax like:
            key in table
        """
        return self._find_slot(key, for_insert=False) is not None

    def __len__(self):
        """
        Return the number of active key-value pairs in the table.

```

```
        Enables syntax like:
            len(table)
        """
        return self.size
```

The table also supports iteration over keys, as well as separate access to keys, values, and items. Since the arrays may contain `EMPTY` and `DELETED` markers, each method must skip those slots. These methods are very useful for printing, debugging, and looping through the contents of the table.

```
    def __iter__(self):
        """
        Iterate over the keys in the table.

        By convention, iterating over a mapping yields keys.
        """
        for key in self.keys:
            if key not in (self.EMPTY, self.DELETED):
                yield key

    def keys(self):
        """
        Return an iterator over the keys in the table.
        """
        return iter(self)

    def values(self):
        """
        Iterate over the values in the table.
        """
        for key, value in zip(self.keys, self.values):
            if key not in (self.EMPTY, self.DELETED):
                yield value

    def items(self):
        """
        Iterate over (key, value) pairs in the table.
        """
        for key, value in zip(self.keys, self.values):
            if key not in (self.EMPTY, self.DELETED):
                yield (key, value)
```

This part provides a readable string representation and a few convenience methods. The `__repr__()` method builds a string that looks similar to a Python dictionary. The `get()` method returns a default value instead of raising an error. The `put()` and `delete()` methods are simply optional wrappers for users who prefer named methods instead of bracket syntax.

```
    def __repr__(self):
        """
        Return a readable string representation of the table.

        Useful for debugging and printing.
        """
        pairs = [f"{key!r}: {value!r}" for key, value in self.items()]
        return "{" + ", ".join(pairs) + "}"

    def get(self, key, default=None):
        """
        Return the value for key if it exists;
        otherwise return default.

        Similar to dict.get().
        """
```

```
        try:
            return self[key]
        except KeyError:
            return default

    def put(self, key, value):
        """
        Optional compatibility method.

        Allows:
            table.put(key, value)
        """
        self[key] = value

    def delete(self, key):
        """
        Optional compatibility method.

        Allows:
            table.delete(key)
        """
        del self[key]
```

As the table fills up, performance would get worse if we kept probing through too many occupied slots. To avoid that, the table doubles its capacity once the load factor becomes too high. The `_resize()` method creates larger arrays, resets the size, and reinserts all active key-value pairs. Deleted slots are not copied into the new table, so resizing also cleans up old deleted entries.

```
    def _resize(self):
        """
        Double the capacity of the table and reinsert all active entries.

        Reinsertion is required because slot positions depend on capacity.
        Deleted slots are discarded during resizing.
        """
        old_keys = self.keys
        old_values = self.values

        self.capacity *= 2
        self.keys = [self.EMPTY] * self.capacity
        self.values = [None] * self.capacity
        self.size = 0

        for key, value in zip(old_keys, old_values):
            if key not in (self.EMPTY, self.DELETED):
                self[key] = value
```

```
table = OpenAddressHashTable()

# Insert values
table["name"] = "Alice"
table["age"] = 25
table["city"] = "Paris"

# Access a value
print(table["name"])          # Alice

# Safe lookup
print(table.get("country"))   # None
print(table.get("country", "Unknown"))  # Unknown
```

```
14
15  # Membership test
16  print("age" in table)          # True
17  print("country" in table)      # False
18
19  # Number of stored items
20  print(len(table))              # 3
21
22  # Iterate over keys directly
23  for key in table:
24      print("key:", key)
25
26  # Iterate over keys
27  for key in table.keys():
28      print("keys():", key)
29
30  # Iterate over values
31  for value in table.values():
32      print("value:", value)
33
34  # Iterate over key-value pairs
35  for key, value in table.items():
36      print(key, "->", value)
37
38  # Delete a key
39  del table["age"]
40  print("age" in table)          # False
41
42  # Print the table
43  print(table)
```

Why Python Uses Open Addressing

- Better cache locality
- Fewer memory allocations
- Faster in practice
- Lower pointer overhead

This design is one reason Python dictionaries are extremely fast.

Method	Memory	Cache Performance
Chaining	Higher	Moderate
Open Addressing	Lower	Excellent

12 Trie

A Trie (also called a **prefix tree**) is a tree-based data structure used to store strings character by character along paths in a tree. It is especially useful when the application needs to support:

- exact word lookup,
- prefix queries,
- autocomplete,
- lexicographic traversal.

Unlike a hash table, which usually treats a string as one whole key, a Trie exposes the internal prefix structure of strings.

12.1 Basic Idea

Suppose we insert the following words:

```
to
tea
ted
ten
in
inn
```

A Trie stores them like this:

```
(root)
|-- t
|   |-- o*
|   \-- e
|       |-- a*
|       |-- d*
|       \-- n*
\-- i
    \-- n*
        \-- n*
```

The symbol `*` means that the node marks the end of a complete word.

The key observations are:

- each edge corresponds to one character,
- each node represents a prefix,
- a root-to-node path represents a string prefix,
- a marked node represents a full stored word.

For example:

- `root` → `t` → `o` stores `"to"`,
- `root` → `t` → `e` → `a` stores `"tea"`,
- `root` → `i` → `n` → `n` stores `"inn"`.

12.2 Node Design

Since Trie nodes are only used internally by the Trie, it is natural to define the node class inside `Trie`. To indicate that it is an internal helper class, we name it `_TrieNode`.

Each node stores:

- `children`: a dictionary mapping characters to child nodes,
- `is_end_of_word`: whether the current node marks the end of a complete word.

Conceptually:

```
_TrieNode
|-- children
|   |-- 'a' -> _TrieNode
|   |-- 'b' -> _TrieNode
|   \-- ...
\-- is_end_of_word
```

12.2.1 Python Implementation of Trie and Internal Node

```python
class Trie:
    class _TrieNode:
        def __init__(self):
            self.children = {}
            self.is_end_of_word = False

    def __init__(self):
        self.root = self._TrieNode()
```

This means:

- `_TrieNode` is nested inside `Trie`,
- the leading underscore indicates that it is meant for internal use,
- the root is created as an instance of `self._TrieNode()`.

12.3 Insert

To insert a word into a Trie:

1. Start at the root.
2. For each character in the word:
   - if a child node for that character does not exist, create one,
   - move to that child.
3. After the final character, mark the node as an end-of-word node.

Example: inserting `"tea"` creates the path

```
(root) -> t -> e -> a*
```

If `"ted"` is inserted afterward, the prefix `"te"` is reused:

```
(root)
\-- t
    \-- e
        |-- a*
        \-- d*
```

```python
class Trie:
    class _TrieNode:
        def __init__(self):
            self.children = {}
            self.is_end_of_word = False

    def __init__(self):
        self.root = self._TrieNode()

    def insert(self, word):
        current = self.root

        for ch in word:
            if ch not in current.children:
                current.children[ch] = self._TrieNode()
            current = current.children[ch]
```

```
        current.is_end_of_word = True
```

- `current = self.root`: start from the root.
- For each character:
  - if the character path does not exist, create a new node,
  - move downward one level.
- After the loop, mark the final node as a complete word.

```
trie = Trie()

trie.insert("tea")
trie.insert("ted")
trie.insert("ten")
trie.insert("to")
```

After these operations, the Trie stores all four words, sharing prefixes where possible.

12.4 Exact Search

To search for a full word:

1. Start at the root.
2. For each character in the word:
   - if the child does not exist, return False,
   - otherwise move to that child.
3. After the last character, return whether the final node is marked as an end-of-word node.

This final check is very important.

If `"tea"` is stored, then the path for `"te"` also exists. But `"te"` should not be reported as a word unless its node is explicitly marked as a complete word.

So:

- path existence alone is not enough,
- the final node must also satisfy `is_end_of_word = True`.

```
class Trie:
    class _TrieNode:
        def __init__(self):
            self.children = {}
            self.is_end_of_word = False

    def __init__(self):
        self.root = self._TrieNode()

    def search(self, word):
        current = self.root

        for ch in word:
            if ch not in current.children:
                return False
            current = current.children[ch]

        return current.is_end_of_word
```

- Start at the root.

- Follow the path of the given word character by character.
- If a required child is missing, the word cannot exist.
- If the full path exists, return whether the final node marks a complete word.

```
trie = Trie()

trie.insert("tea")
trie.insert("ted")

print(trie.search("tea"))   # True
print(trie.search("ted"))   # True
print(trie.search("te"))    # False
print(trie.search("toast")) # False
```

Expected behavior:

- `"tea"` exists,
- `"ted"` exists,
- `"te"` is only a prefix, not a complete word,
- `"toast"` has no valid path.

12.5 Prefix Search

A Trie is especially powerful for prefix queries.

To check whether any stored word begins with a given prefix:

1. Start at the root.
2. Follow the prefix character by character.
3. If any required child is missing, return False.
4. If the full prefix path exists, return True.

Notice the key difference from exact search:

- prefix search only asks whether the path exists,
- it does **not** require the final node to be marked as a complete word.

Example:

- if `"tea"`, `"ted"`, and `"ten"` are stored,
- then prefix `"te"` should return True,
- even if `"te"` itself is not a stored word.

12.5.1 Python Implementation of Prefix Search

```
class Trie:
    class _TrieNode:
        def __init__(self):
            self.children = {}
            self.is_end_of_word = False

    def __init__(self):
        self.root = self._TrieNode()

    def starts_with(self, prefix):
        current = self.root

        for ch in prefix:
            if ch not in current.children:
                return False
```

```
            current = current.children[ch]

        return True
```

- Start at the root.
- Follow the given prefix one character at a time.
- If a character path is missing, then no stored word can begin with that prefix.
- If the full prefix is matched, return True immediately.

There is no final `is_end_of_word` check because prefix existence is enough.

```
trie = Trie()

trie.insert("tea")
trie.insert("ted")
trie.insert("ten")
trie.insert("to")

print(trie.starts_with("te"))   # True
print(trie.starts_with("to"))   # True
print(trie.starts_with("toa"))  # False
print(trie.starts_with("i"))    # False
```

Expected behavior:

- `"te"` is a valid prefix of several words,
- `"to"` is a valid prefix and also a complete word,
- `"toa"` is not present as a prefix,
- `"i"` is not present because no inserted word begins with `i`.

12.6 Delete

Deletion is more subtle than insertion and search.

If we want to delete a word:

1. Follow its path.
2. If the word does not exist, stop.
3. If it exists, unmark the final node as an end-of-word node.
4. Then, while returning upward, delete nodes only if:
    - they have no children, and
    - they are not the end of another word.

Why is this necessary? Because many words may share prefixes.

Example:

- stored words: `"in"`, `"inn"`
- deleting `"inn"` must not destroy `"in"`

Before deletion:

```
(root)
\-- i
    \-- n*
        \-- n*
```

After deleting `"inn"`:

```
(root)
\-- i
    \-- n*
```

So deletion must be careful not to remove nodes still needed by other words.

```python
class Trie:
    class _TrieNode:
        def __init__(self):
            self.children = {}
            self.is_end_of_word = False

    def __init__(self):
        self.root = self._TrieNode()

    def delete(self, word):
        self._delete(self.root, word, 0)

    def _delete(self, node, word, depth):
        if depth == len(word):
            if not node.is_end_of_word:
                return False
            node.is_end_of_word = False
            return len(node.children) == 0

        ch = word[depth]
        if ch not in node.children:
            return False

        should_delete_child = self._delete(node.children[ch], word, depth + 1)

        if should_delete_child:
            del node.children[ch]
            return len(node.children) == 0 and not node.is_end_of_word

        return False
```

The helper function returns a Boolean meaning:

Should this node be removed from its parent?

The logic is:

- if we reach the end of the word:
  - if this is not a real stored word, return False,
  - otherwise unmark it,
  - if it has no children, it may be deleted.
- if we are still inside the word:
  - recursively process the child,
  - if the child should be deleted, remove that child link,
  - then decide whether the current node has also become removable.

A node can be deleted only if:

- it has no children,
- it is not the end of another word.

```python
trie = Trie()

trie.insert("in")
trie.insert("inn")

```

```
print(trie.search("in"))    # True
print(trie.search("inn"))   # True

trie.delete("inn")

print(trie.search("in"))    # True
print(trie.search("inn"))   # False
```

Expected behavior:

- `"inn"` is deleted,
- `"in"` remains valid.

12.7 Time Complexity

Let L be the length of the input string.

- Insert: $O(L)$
- Exact search: $O(L)$
- Prefix search: $O(L)$
- Delete: $O(L)$

12.8 Trie vs Hash Table

A Trie and a hash table both support lookup, but they are optimized for different tasks.

Feature	Trie	Hash Table
Exact word lookup	Good	Good
Prefix lookup	Excellent	Poor
Ordered traversal	Natural	Not natural
Autocomplete	Excellent	Poor
Memory usage	Often high	Usually lower

A Trie is preferred when prefix-based queries are important. A hash table is often simpler when only exact lookup is needed.

13 Red-Black Trees

A Red-Black tree is a self-balancing binary search tree that maintains its balance during insertions and deletions. Red-Black trees are widely used in practice and are the fundamental data structure used in many programming language libraries like Java's TreeMap and C++'s std::map.

Red-Black trees offer several advantages:

- Guaranteed $O(\log n)$ time complexity for search, insertion, and deletion operations
- Self-balancing with predictable performance
- More efficient rebalancing than AVL trees in many practical scenarios

A Red-Black tree must satisfy the following properties:

1. Every node is either red or black.
2. The root is black.
3. All leaves (NULL nodes) are black. In RBT, leaf nodes are NULL/NIL nodes.
4. If a node is red, then both its children are black (no two adjacent red nodes).
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes (black-height property).

Every leaf node has invisible NULL children. These NULL nodes are always colored black and are important during deletion operations. When a node is deleted, it becomes a NULL node and may be marked as a double black (DB) node.

These properties ensure that the tree remains approximately balanced. The longest path from the root to any leaf is at most twice as long as the shortest path (The shortest possible path contains only black nodes (h nodes), the longest possible path alternates between red and black nodes (at most 2h nodes)), resulting in $O(\log n)$ height and thus $O(\log n)$ time complexity for basic operations.

```
class RedBlackTree:
    """
    Red-Black Tree class.

    # - root:
    #     The item stored at the root of this tree, or None if this tree is empty.
    # - left:
    #     The left subtree, or None if this tree is empty.
    # - right:
    #     The right subtree, or None if this tree is empty.
    # - color:
    #     The color of this node ("RED" or "BLACK"), or None if this tree is empty.
    # - parent:
    #     The parent of this node, or None if this is the root or an empty tree.
    """

    def __init__(self, root, color="RED", parent=None):
        """
        Initialize a new RBT containing only the given root value.

        If <root> is None, initialize an empty RBT.
        """
        if root is None:
            self.root = None
            self.left = None
            self.right = None
            self.color = None
            self.parent = None
        else:
            self.root = root
            self.left = RedBlackTree(None)  # self.left is an empty RBT
            self.right = RedBlackTree(None)  # self.right is an empty RBT
            self.color = color
            self.parent = parent

            # Connect children to parent
            self.left.parent = self
            self.right.parent = self
```

Red-Black trees are built on top of standard binary search trees, so we'll first define basic operations:

```
class RedBlackTree:
    def is_empty(self):
        """
        Return whether this RBT is empty.
        """
        return self.root is None

    def __contains__(self, item):
        """
        Return whether <item> is in this RBT.
        """
        if self.is_empty():
            return False
        elif item == self.root:
            return True
        elif item < self.root:
            return self.left.__contains__(item)
        else:
```

```
            return self.right.__contains__(item)

    def inorder_traversal(self):
        """
        Return a list of all items in this RBT in sorted (inorder) order.
        """
        if self.is_empty():
            return []
        else:
            # Get all values from left subtree
            left_values = self.left.inorder_traversal()

            # Add the root value with its color
            root_value = [(self.root, self.color)]

            # Get all values from right subtree
            right_values = self.right.inorder_traversal()

            # Combine all values in the correct order
            return left_values + root_value + right_values
```

13.1 Tree Rotations

Rotations are fundamental operations in maintaining the balanced structure of a Red-Black Tree (RBT). They help to preserve the properties of the tree, ensuring that the longest path from the root to any leaf is no more than twice the length of the shortest path. Rotations come in two types: **left rotations** and **right rotations**.

A left rotation at node x moves x down to the left and its right child y up to take x's place.

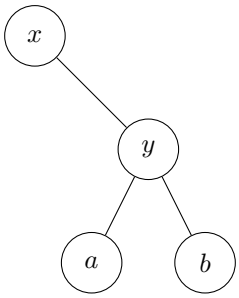


Figure 2: Before Left Rotation

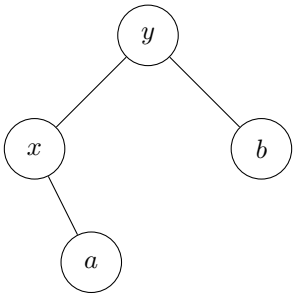


Figure 3: After Left Rotation

```
    def left_rotate(self):
        """
        Perform a left rotation on this node.

        Precondition: not self.is_empty() and not self.right.is_empty()
        """
        if self.is_empty() or self.right.is_empty():
```

```
            return

        # Store the right child
        y = self.right

        # Turn y's left subtree into self's right subtree
        self.right = y.left
        if not y.left.is_empty():
            y.left.parent = self

        # Link self's parent to y
        y.parent = self.parent

        if self.parent is None:
            # This was the root node
            pass  # The tree handling will be done by the insert/delete methods
        elif self.parent.left == self:
            self.parent.left = y
        else:
            self.parent.right = y

        # Put self on y's left
        y.left = self
        self.parent = y

        return y  # Return the new root of this subtree
```

A right rotation at node y moves y down to the right and its left child x up to take y's place.

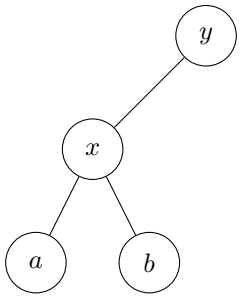


Figure 4: Before Right Rotation

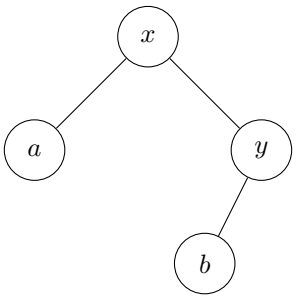


Figure 5: After Right Rotation

```
    def right_rotate(self):
        """
        Perform a right rotation on this node.

        Precondition: not self.is_empty() and not self.left.is_empty()
        """
        if self.is_empty() or self.left.is_empty():
            return

```

```
        # Store the left child
        x = self.left

        # Turn x's right subtree into self's left subtree
        self.left = x.right
        if not x.right.is_empty():
            x.right.parent = self

        # Link self's parent to x
        x.parent = self.parent

        if self.parent is None:
            # This was the root node
            pass  # The tree handling will be done by the insert/delete methods
        elif self.parent.left == self:
            self.parent.left = x
        else:
            self.parent.right = x

        # Put self on x's right
        x.right = self
        self.parent = x

        return x  # Return the new root of this subtree
```

13.2 Insertion

Insertion in a Red-Black tree starts with a standard BST insertion, but then requires additional operations to restore the Red-Black properties:

```
    def insert(self, item):
        """
        Insert <item> into this RBT and maintain the RBT properties.
        """
        # Step 1: Perform standard BST insert
        self._bst_insert(item)

        # Step 2: Fix the Red-Black Tree violations
        self._fix_insert(self._find_node(item))

    def _bst_insert(self, item):
        """
        Insert <item> into this tree using standard BST insertion.
        """
        if self.is_empty():
            # Make this node a leaf containing the item
            self.root = item
            self.left = RedBlackTree(None)
            self.right = RedBlackTree(None)
            self.color = "RED"  # New nodes are always RED

            # Connect children to parent
            self.left.parent = self
            self.right.parent = self
        elif item <= self.root:
            # Insert item in the left subtree
            self.left._bst_insert(item)
        else:
            # Insert item in the right subtree
            self.right._bst_insert(item)

    def _find_node(self, item):
        """
        Find and return the node with <item> as its root.
```

```
        If <item> is not in the tree, return None.
        """
        if self.is_empty():
            return None
        elif item == self.root:
            return self
        elif item < self.root:
            return self.left._find_node(item)
        else:
            return self.right._find_node(item)

```

The representation we will be working with is shown in Figure below:

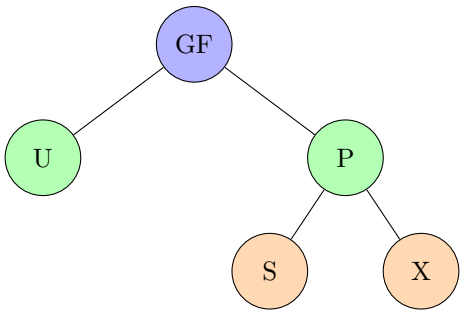


Figure 6: Node representation where GF = X's Grandfather, U = X's Uncle, P = X's Parent, S = X's Sibling, X = Node being inserted

The logic for insertion in a Red-Black Tree follows these steps:

1. Insert the node using standard binary tree insertion and assign a red color to it.
2. If the node is a root node, change its color to black.
3. If it's not a root node, check the color of the parent node:
   - If the parent's color is black, no changes are needed
   - If the parent's color is red, check the color of the node's uncle:
     - If the uncle has a red color, change the color of the node's parent and uncle to black, and that of the grandfather to red. Then, recursively check the grandfather. If the grandfather is the root, don't change its color to red.
     - If the uncle has a black color, perform rotations based on the specific case (discussed below).

Red Uncle Case

When the uncle of the newly inserted node is red, we perform recoloring as shown in Figure 7:

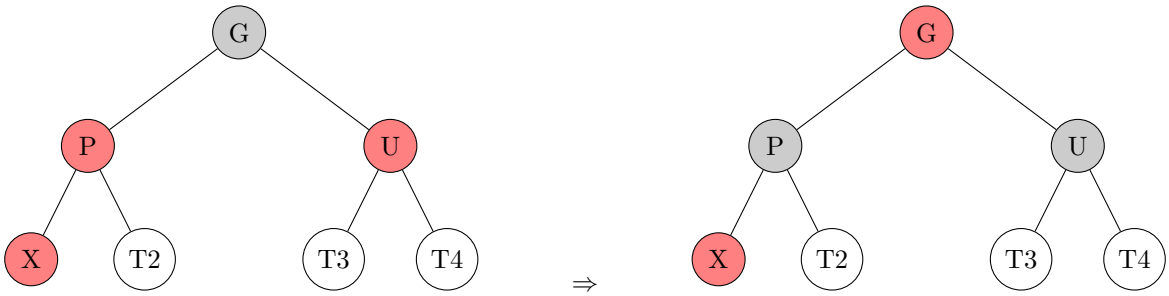


Figure 7: Red Uncle Case: 1) Change color of X's parent P and uncle U to black. 2) Change color of grandfather G to red.

After this transformation, we need to recursively apply the recoloring steps for the grandfather (G) if necessary, treating it as the new node X in our algorithm.

Black Uncle Case

When the uncle is black, there are four possible cases to consider:

1. **Left Left Case (LL rotation):**

   In this case, the parent P is the left child of grandparent G, and the newly inserted node X is the left child of P.

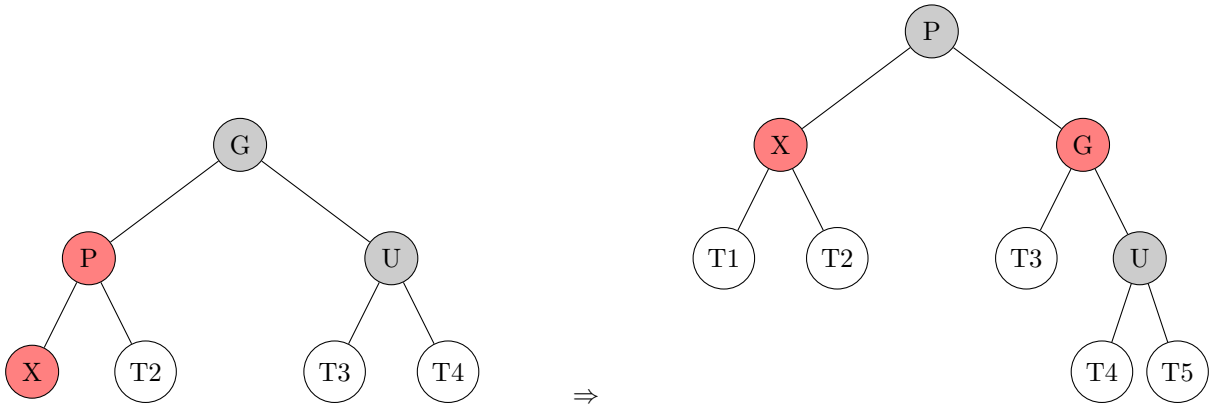


Figure 8: Left Left Case: 1) Right rotation of grandfather G. 2) Swap colors of grandfather G and parent P.

2. **Left Right Case (LR rotation):**

   In this case, the parent P is the left child of grandparent G, and the newly inserted node X is the right child of P.

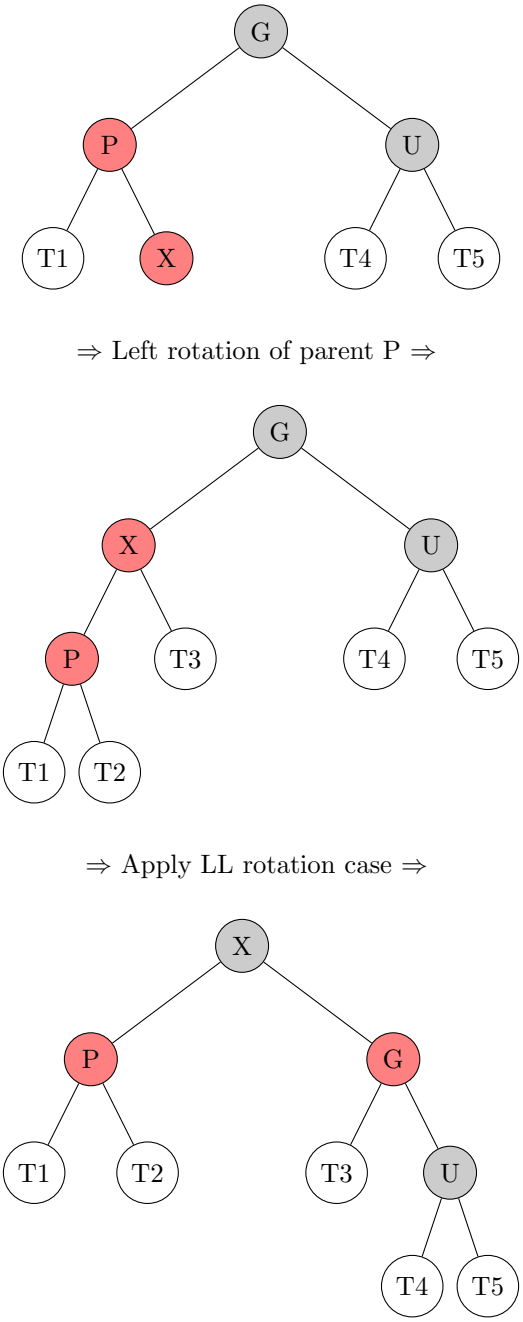


Figure 9: Left Right Case: 1) Left rotation of parent P. 2) Then apply LL rotation case.

3. **Right Right Case (RR rotation):**

   This is the mirror of the Left Left case. The parent P is the right child of grandparent G, and the newly inserted node X is the right child of P.

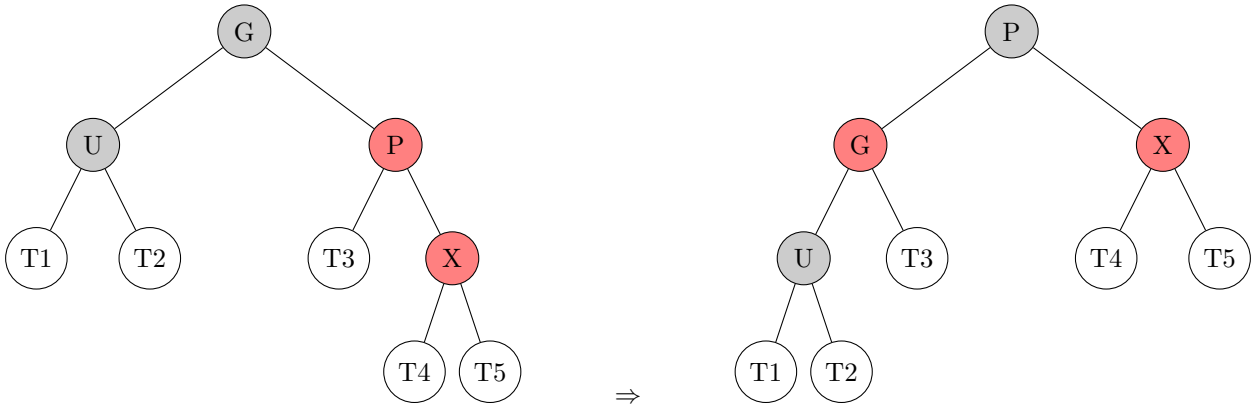


Figure 10: Right Right Case: 1) Left rotation of grandfather G. 2) Swap colors of grandfather G and parent P.

4. **Right Left Case (RL rotation):**

   This is the mirror of the Left Right case. The parent P is the right child of grandparent G, and the newly inserted node X is the left child of P.

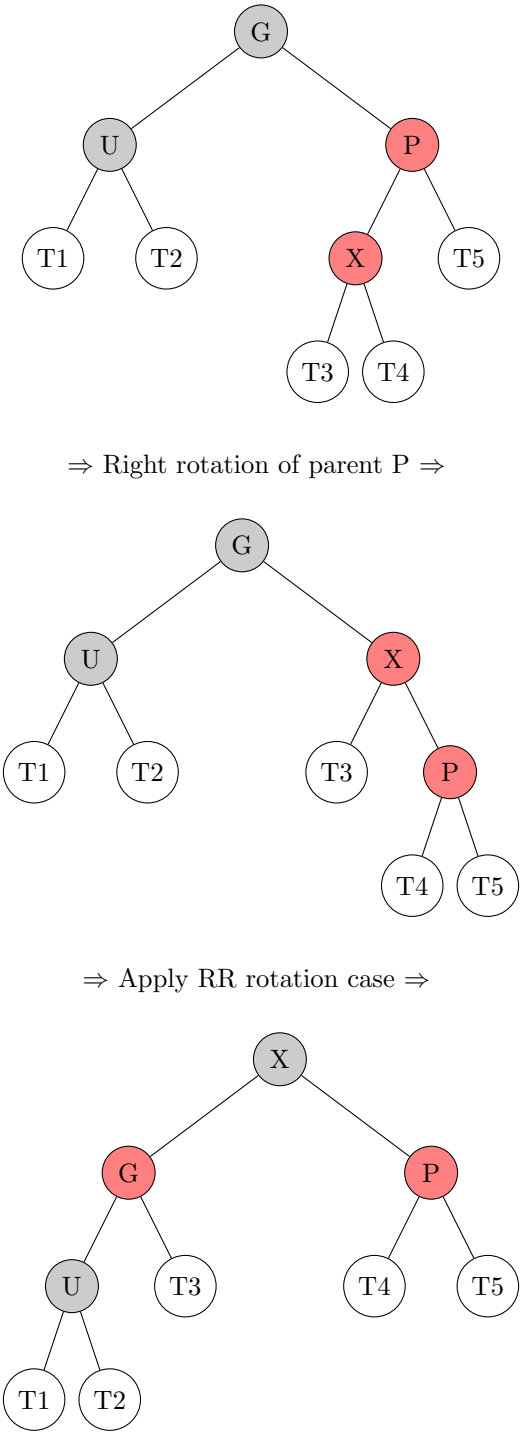


Figure 11: Right Left Case: 1) Right rotation of parent P. 2) Then apply RR rotation case.

5. **Recoloring After Rotations:**

   After performing rotations, we need to recolor nodes according to the rotation case:

   - For Left Left Case and Right Right case: Swap colors of grandparent and parent after rotations
   - For Left Right Case and Right Left Case: Swap colors of grandparent and inserted node (X) after rotations

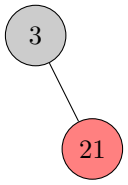
Example

Let's illustrate the insertion algorithm by creating a Red-Black tree with elements 3, 21, 32, and 15:

1. **Insert element 3**:

When the first element is inserted, it becomes the root node and acquires the black color.

2. **Insert element 21:**



The new element is always inserted with a red color. Since $21 > 3$, it becomes the right child of the root node.

3. **Insert element 32**:

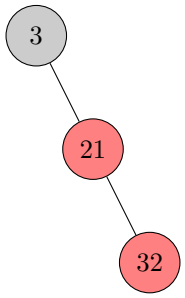


Figure 12: Tree after inserting 32, with a red-red violation

When we insert 32, we see that there is a red parent-child pair (21 and 32), which violates the Red-Black tree property. Since this is a Right-Right (RR) case, we need to perform a left rotation at the grandparent (3) and then recolor.

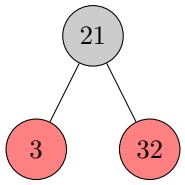


Figure 13: Tree after left rotation and recoloring

4. **Insert element 15:**

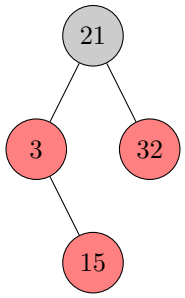


Figure 14: Tree after inserting 15, with a red-red violation

After inserting 15, we again have a red-red violation. Since the uncle (32) is also red, we perform recoloring:

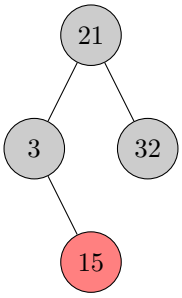


Figure 15: Final Red-Black Tree after all insertions and rebalancing

5. **Time Complexity:** The time complexity for insertion in a Red-Black Tree is $O(\log N)$, where N is the total number of nodes in the tree. This includes:
   - $O(\log N)$ for finding the position to insert the new node
   - $O(1)$ for recoloring
   - $O(1)$ for rotations
   - At most $O(\log N)$ for fixing violations all the way up to the root
6. **Space Complexity:**

   The space complexity is $O(N)$, where N is the total number of nodes in the Red-Black tree.

```
    def _get_root_tree(self):
        """
        Return the root tree of the entire RBT.
        """
        if self.parent is None:
            return self
        else:
            return self.parent._get_root_tree()

    def _get_uncle(self):
        """
        Return the uncle node of this node.
        """
        if self.parent is None or self.parent.parent is None:
            return None

        grandparent = self.parent.parent

        if grandparent.left == self.parent:
            return grandparent.right
        elif grandparent.right == self.parent:
            return grandparent.left
        else:
            # This should not happen in a properly maintained tree,
            # but add as a safety check
            return None

    def _fix_insert(self, node):
        """
        Fix the Red-Black Tree violations after insertion.

        Precondition: node is not None
        """
        if node is None:
            return

        # Case 1: Node is the root
        if node.parent is None:
            node.color = "BLACK"
            return
```

```
        # If parent is black, no violation
        if node.parent.color == "BLACK":
            return

        # At this point, we know parent is RED
        # Check if grandparent exists
        if node.parent.parent is None:
            # If no grandparent, parent must be root
            node.parent.color = "BLACK"
            return

        uncle = node._get_uncle()
        grandparent = node.parent.parent

        # Case 2: Uncle is RED
        if uncle is not None and not uncle.is_empty() and uncle.color == "RED":
            node.parent.color = "BLACK"
            uncle.color = "BLACK"
            grandparent.color = "RED"
            # Recursively fix the grandparent
            self._fix_insert(grandparent)
            return

        # Case 3: Uncle is BLACK and node is "inner grandchild"
        if grandparent.left == node.parent and node == node.parent.right:
            # Left-Right case
            new_root = node.parent.left_rotate()
            # Now node.parent is the new subtree root
            # Continue with Case 4, but with the former parent
            self._fix_insert(new_root.left)
            return
        elif grandparent.right == node.parent and node == node.parent.left:
            # Right-Left case
            new_root = node.parent.right_rotate()
            # Now node.parent is the new subtree root
            # Continue with Case 4, but with the former parent
            self._fix_insert(new_root.right)
            return

        # Case 4: Uncle is BLACK and node is "outer grandchild"
        node.parent.color = "BLACK"
        grandparent.color = "RED"

        if node == node.parent.left:
            # Left-Left case
            grandparent.right_rotate()
        else:
            # Right-Right case
            grandparent.left_rotate()
```

13.3 Deletion

Deletion in red-black trees follows these general steps:

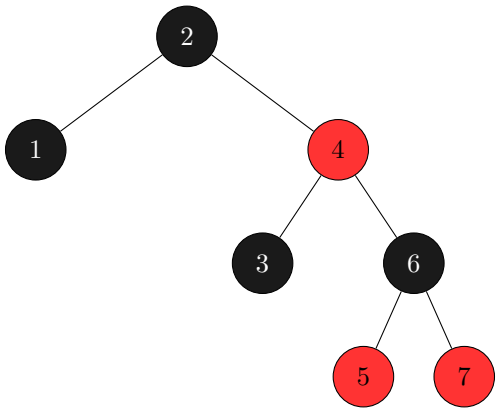
1. Perform standard BST deletion
2. If the deleted node was red, no properties are violated
3. If the deleted node was black, we create imbalance in the black height property
4. Use special cases to restore the red-black properties

The following table outlines the various cases for handling deletion in a red-black tree:

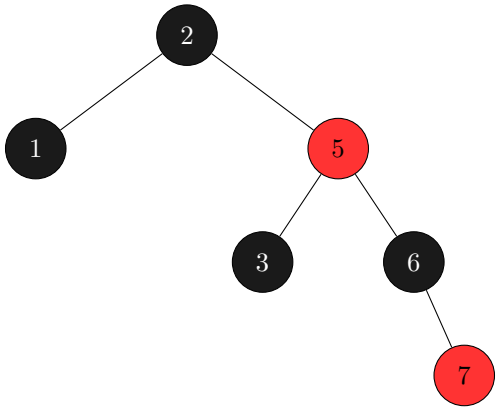
Case	Condition	Action
1	Node to be deleted is a red leaf node	Simply remove it from the tree, but still need to adjust color
2	Double Black (DB) node is root	Remove the DB; root becomes black
3	(a) DB's sibling is black, and (b) DB's children are black	(a) Remove the DB sign (b) Make DB's sibling red (c) If DB's parent is black, make it DB and continue recursion (d) If DB's parent is red, make it black and stop
4	DB's sibling is red	(a) Swap color of DB's parent with DB's sibling (b) Rotate at parent node in the direction of DB (c) Apply appropriate case (case 3, 5, 6) to the new structure
5	(a) DB's sibling is black (b) DB's sibling's far child is black (c) DB's sibling's near child is red	(a) Turn the sibling red, the near child black and: (b) If sibling is the right child, perform a right rotation around the sibling. (c) If sibling is the left child, perform a left rotation around the sibling. (d) continue recursion
6	(a) DB's sibling is black, and (b) DB's sibling's far child is red	(a) Copy the color of the parent to the sibling and turn the parent and the far child black. (b) If the sibling is the right child, perform a left rotation around the parent. (c) If sibling is the left child, perform a right rotation around the parent. (d) Done, you may stop the recursion here.

Table 2: Red-Black Tree Deletion Cases

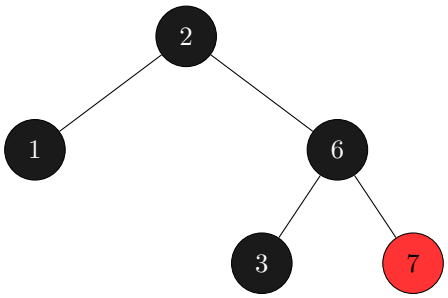
Let's delete 4,5,3 from the tree below:



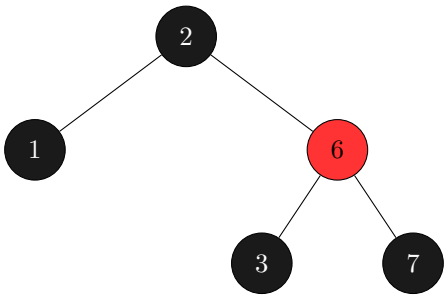
To delete the element 4, let us perform the binary search deletion first(minimum of right).



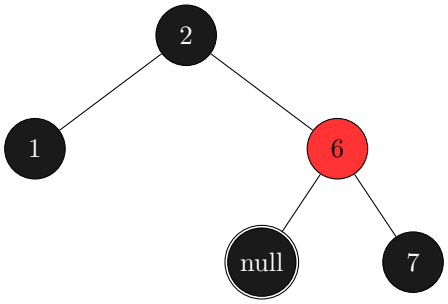
After performing the binary search deletion, the RB Tree property is not disturbed, therefore the tree is left as it is. Then, we delete the element 5 using the binary search deletion.



But the RB property is violated after performing the binary search deletion, i.e., all the paths in the tree do not hold same number of black nodes; so we swap the colors to balance the tree.



Then, we delete the node 3 from the tree obtained:



We apply case 3 deletion as double black's sibling node is black and its child nodes are also black. Here, we remove the double black, recolor the double black's parent and sibling.