ITSC 2214 - Data Structures and Algorithms - Summer 2023

2023-05-22

Table of contents

Preface

This website contains a set of readings for ITSC 2214 - Data Structures and Algorithms.

Part I

On Computers and Computing

1 Background on Computer Science and Data Structures

Let's break down the term "Computer Science" into its two parts: **computer** and **science**. What do these words mean?

The word **science** means "knowledge" in Latin. Science is a way of finding out things about the world by asking questions, doing experiments, and looking for evidence.

The word **computer** means "to calculate" in Latin. A computer is a machine that can do math problems and store information very fast. The word **computing** means using computers or other machines to solve problems or do tasks.

But did you know that computers were not always machines? Before electronic computers were invented, there were people who did math problems by hand for scientists and engineers. They were called **human computers**. They followed fixed rules and had no authority to deviate from them in any detail. They worked in teams and checked each other's results for accuracy.

1.1 Computing Capabilities

Computers, in the broadest sense, are devices that can perform calculations or manipulate information. Throughout history, humans have invented and used various types of computers, each with increasing capabilities and complexity. Here are some examples of how computing capabilities have evolved over time:

- **Tally stick** - One of the earliest forms of computers, dating back to prehistoric times. A tally stick is a piece of wood or bone with notches carved into it to record numbers or events. The computations it could do were incrementing and retrieving one piece of information. For example, a shepherd could use a tally stick to keep track of his sheep by making a notch for each one.
- **Abacus** - A manual device used for calculations by sliding counters along rods or in grooves. The abacus was invented in ancient times and is still used today in some parts of the world. It could store one set of numbers, add, subtract, multiply, divide, and perform other arithmetic operations to the stored information which can later be retrieved. For example, a merchant could use an abacus to keep track of his transactions and profits.

NASA Dryden Flight Research Center Photo Collection
http://www.dfrc.nasa.gov/gallery/photo/index.html
NASA Photo: E49-54 Date: 1949

NACA High Speed Flight Station "Computer Room"

Figure 1.1: For example, in 1959, NASA had a team of women who worked as human computers to help launch rockets into space. They used pencils, paper, and calculators to do complex calculations that machines could not do at that time.

- **Astrolabe** - A sophisticated instrument used for astronomy and navigation by measuring the positions and movements of celestial bodies. The astrolabe was developed in ancient Greece and reached its peak in the Islamic Golden Age. It could perform complex calculations such as determining the time, latitude, longitude, and direction based on the observation of stars and planets. For example, a sailor could use an astrolabe to find his way across the sea by aligning it with the sun or the pole star.
- **Antikythera mechanism** - A mechanical device that simulated the motions of the sun, moon, and planets according to a geocentric model. The Antikythera mechanism was discovered in a shipwreck near the Greek island of Antikythera in 1901. It is estimated to date back to the 2nd century BC and is considered one of the first analog computers. It could predict astronomical phenomena such as eclipses, phases of the moon, and positions of the zodiac signs. For example, a priest could use the Antikythera mechanism to plan religious ceremonies and festivals based on the celestial calendar.
- **Difference engine** - A mechanical calculator that could compute polynomial functions using the method of finite differences. The difference engine was designed by Charles Babbage in the early 19th century but was never fully completed due to technical and financial difficulties. It could generate accurate tables of values for various mathematical functions such as logarithms, trigonometry, and navigation. For example, a mathematician could use the difference engine to check his calculations and avoid errors.
- **Analytical engine** - A proposed mechanical computer that could perform any calculation given a set of instructions or a program. The analytical engine was also designed by Charles Babbage in the mid-19th century but was never built due to his death and lack of funding. It is considered the first general-purpose computer and the precursor of modern computers. It could store data in memory, process data using arithmetic and logical operations, control the flow of execution using conditional branching and looping, and output data using a printer or a punch card. For example, Ada Lovelace, who wrote the first algorithm for the analytical engine, envisioned that it could compose music based on mathematical rules.

Here is a summary of the computing capabilities of some of these devices -

- **Tally stick**: Store and retrieve one piece of data
- **Abacus**: Store and retrieve one piece of data, and perform basic arithmetical operations on them with another operand.
- **System of Gears**: Store and retrieve one piece of data. Each gear can use stored data and scale it up or down (multiply or divide) by a fixed constant determined by the gear ratio.
- **Difference engine**: Can perform complex, but non-programmable computations. Produced tables of input-output pairs.
- **Analytical engine**: Can perform complex, programmable computations. Can store data in memory, process data using arithmetic and logical operations, control the flow of execution using conditional branching and looping, and output data using a printer or a punch card.

1.2 What is Computer Science then?

The science of computers, or Computer Science, seeks to answer fundamental questions like: What are the essential parts of a computer? What can be computed, and what cannot? What determines the ease and speed of computation? This field provides the foundation for understanding the principles that drive computational systems.

1.3 Data Structures and Algorithms

The field of Data Structures and Algorithms expands upon the principles of Computer Science. It determines the efficiency of computation by answering the question: How easily or quickly can something be computed, and what factors influence these metrics? It outlines the necessary building blocks or tools required for programming, and explores the common patterns and problems in programs, offering known ways to enhance their speed. It is this branch of Computer Science that gives us the skills to design, write and analyze the efficiency of our programs. This understanding is crucial to becoming proficient in the broader field of Computer Science.

In essence, the intertwined journey of computers and computing science is a testament to human ingenuity and the relentless pursuit of understanding and harnessing the principles that underlie our world. As we delve deeper into the concepts of data structures and algorithms, we continue to contribute to this exciting journey.

1.4 Review

We talked a little about what computers are, and what capabilities a device needs to have to be able to compute certain types of problems. The next chapter will be about "general purpose, programmable computers", and how they work.

Here's a fun exercise for you -

- Are analog wristwatches computers? What do they compute?
- Can you design a system of gears that can convert Fahrenheit to Celcius?

2 Modern General-Purpose, Programmable Computers

In this chapter, we will explore the functioning of modern, general-purpose computers. We will approach this topic by drawing parallels with a relatable example from an alternate universe, where two boys, Cory and Jamal, develop a unique system of communication.

In this parallel universe, Cory and Jamal are two teenage neighbors that enjoy playing Scrabble, but there's a hitch. Cory's parents are quite religious and in this universe, there exists a prophecy from their scriptures that a game played by youngsters will one day end the world; and so they disapprove of Cory playing Scrabble.

To get around this, Cory and Jamal come up with a clever plan. Cory happens to have two desk lamps that are visible from Jamal's window. They decide that if Cory's mother isn't home, Cory will switch on the right lamp, and if his father isn't home, he will turn on the left one. If both lamps are on, it essentially signals to Jamal that he can come over to play Scrabble.

Cory also has a sister who is usually at the University, but also at home on some weekends. When she is home, she tends to wake up for late-night snacks around 1 am. And so when Cory's sister is home, they can only play until 1 am. To work around this, they add another signal. If Cory switches on his ceiling fan, it will signal that he can come to play, but only until 1 am.

As we see, each additional piece of information that we need to share requires an additional indicator.

2.1 The Working of a Processor

The way Cory and Jamal communicate parallels how a computer's processor operates. A processor has a set of operations it can perform - like copying, moving, adding, subtracting data, jumping to another instruction, conditional branching, and more.

To tell a computer to perform a certain operation, we need several indicators, just like Cory and Jamal's system. Computers use tiny wires that may or may not have electricity running through them. The presence or absence of electricity indicates what the processor should do.

We refer to the absence as a `0` and the presence of electricity as a `1`. Each wire conveying either a `0` or a `1` is said to convey one `bit` of information.

This is a fixed group of wires going into the processor to convey such information on what needs to be done. This grouping is known as "Instruction." The number of bits in an instruction tells us the 'size" of the instruction.

A processor usually has a fixed instruction size (64 for a 64-bit processor, 32 for a 32-bit processor, etc.).

Sometimes even more information is needed. For example, for operations like jumping to a different instruction, the processor needs to know where to jump to. For operations like addition, two numbers are needed. Sometimes, along with the operation, the processor also needs to be instructed about where to store the result. We call these pieces of data on which we perform instructions as `operands`. We often refer to instructions as operations.

We cannot have an extra set of wires for each time we need to share another operand with the processor, so we share this data in sequence.

Say `0010 1010` is the code for the `add` operation, `0000 0000`` is the number 0, and`0000 0001' is the number 1. If we want an 8-bit processor to add these together, we might share the following data through the wires -

```
...
0000 0001 // third cycle - second operand
0000 0000 // second cycle - first operand
0010 1010 // first cycle - add operation
```

There is a "clock" mechanism within the processor to signal the start and end of "cycles". Generally the processor "pipelines" its operation such that it can complete one instruction in each cycle. However, many complex instructions like division or square roots take >20 cycles to complete.

A computer processor can be compared to a complex Rube Goldberg machine, with wires that take in the presence or absence of electricity similar to how many Rube Goldberg machines' mechanism is kicked off by a rolling ball or marbles. The "marbles" of electricity roll through the processor, along each wire, triggering complex chain reactions or side effects that are designed to elicit the intended operation. Consider that a processor, as small as it is, frequently contains billions of transistors - use this information to imagine how complex these "Rube Goldberg machines" are.

2.2 Memory

Let's compare the computer's memory to a vast grid filled with tiny cells. These cells can each store some electricity - and again the presence or absence of electricity conveys a `bit` of information. If we continue the previous analogy, it can be compared to a large grid storing balls (data).

The processor has instructions that allow it to select which cells to "read" and transfer those bits to the processor - to be interpreted either as data or instructions. The processor also has instructions that allow it to write to memory.

It is this interaction of a processor being able to "write" instructions and data to memory, and then conditionally "read" and execute them, and then write new instructions or data that makes a processor "programmable".

What we describe in this section is Active memory or RAM. It is the data in RAM that is readily available for the processor to read and write.

2.3 Programs

Programs are essentially detailed sets of instructions that command a computer to execute specific operations. They can be likened to a recipe that a computer follows to achieve a particular task. Programs dictate what steps the computer must take and in what sequence to reach the desired outcome.

The program is stored on disk in a format that is standard for that operating system, and where the first instruction is stored in this format is always known - say, the first instruction in a "main" section. When a program is executed, it is loaded from the computer's storage into RAM, and then the processor reads and executes the first instruction; and so the execution begins.

The complexity of programs can vary greatly, from a simple one that performs basic arithmetic to an intricate operating system like Windows or Linux that manages every aspect of a computer. However, the core characteristic of all programs is the same: they are sequences of instructions that the computer follows.

2.4 The Java Compiler

As instructions are hard to write directly, we make use of programs called compilers that take in "code" in human-readable text format and output a list of instructions. The program is designed in such a way that the produced list of valid instructions always carries out the task described in "code" faithfully.

The Java compiler is such a program for Java code. When a Java program is compiled, the compiler reviews the code for syntax errors and then translates it into bytecode, a type of intermediate language closer to machine language. The Java Virtual Machine (JVM) then interprets this bytecode into machine code that your computer's processor can execute.

Syntax errors or other kinds of errors essentially refer to situations where the compiler doesn't know how to, or cannot produce a valid set of instructions that can carry out what the code is describing.

Compilers like the Java compiler don't just translate code. They also optimize it, making it more efficient so that the resulting program runs faster and consumes less memory. For example, they may look at your `for` loop that is adding up the first `N` natural numbers and decide to replace the loop with the formula for this computation instead.

2.5 Operating Systems

With a sound understanding of the fundamental components of modern computing, it's important to highlight the role of the operating system.

An operating system (OS) is a type of system software that manages computer hardware and software resources and provides various services for computer programs. It acts as a mediator between users and the computer hardware. Users interact with the operating system through user interfaces such as a command-line interface (CLI) or a graphical user interface (GUI).

Operating systems bear the responsibility of managing the computer's resources, including the processor, memory, disk space, and input/output devices. They coordinate tasks, ensuring that the processor's time is used judiciously, and manage the memory, keeping track of which parts are in use and which are available.

In essence, the operating system provides the platform on which all other software runs. It is the environment in which programs, written in languages like Java and then compiled, operate.

This fundamental understanding of modern computing components helps elucidate the intricate operations that are continuously happening within our laptops, desktops, and even our smartphones. These fundamental aspects form the backbone of the digital age we live in.

Part II

Algorithmic Analysis

3 Algorithmic Analysis

In this chapter, we will delve into the exciting world of algorithmic analysis. The objectives of this chapter are to learn to communicate the speed of an algorithm effectively and to appraise the performance of an algorithm from pseudocode or Java code.

3.1 Problems, Algorithms, and Programs

Before understanding algorithmic analysis, it's essential to differentiate between problems, algorithms, and computer programs. These are three distinct concepts that are interrelated.

3.1.1 Problems

A problem in computer science refers to a specific task that needs to be solved. It can be thought of in terms of inputs and matching outputs. For instance, to solve the problem of finding the youngest student in our class, the input would be the names and ages of all students in the class. The output would be the name of the youngest student.

It's helpful to perceive problems as functions in a mathematical sense. In mathematics, a function is a relationship or correspondence between two sets — the input set (domain) and the output set (range).

3.1.2 Algorithms

An algorithm, on the other hand, is a method or a process followed to solve a problem. If we perceive the problem as a function, then an algorithm can be seen as an implementation of this function that transforms an input into the corresponding output.

Since there are typically numerous ways to solve a problem, there could be many different algorithms for the same problem. Having multiple solutions is advantageous because a specific solution might be more efficient than others for certain variations of the problem or specific types of inputs.

For instance, one sorting algorithm might be best suited for sorting a small collection of integers. Another might excel in sorting a large collection of integers, while a third might be ideal for sorting a collection of variable-length strings.

By definition, a sequence of steps can only be called an algorithm if it fulfills the following properties:

- It must be correct.
- It consists of a series of concrete steps.
- There is no ambiguity about the step to be performed next.
- It must comprise a finite number of steps.
- It must terminate.

3.1.3 Programs and Their Building Blocks

Before discussing programs in detail, let's briefly review two essential components that make a program run: the CPU (Central Processing Unit) and memory.

A CPU is the electronic circuitry within a computer that has the ability to execute certain instructions. Its primary function is to fetch, decode, and execute instructions. It has slots to store data, referred to as registers. Communicating with a CPU involves telling it what operation you want to perform and on which data. For instance, you might instruct the CPU to add two numbers stored in registers `A` and `B` and store the result in register `C`.

CPU instructions are binary codes that specify which operation the CPU should perform. Here's an example of what they look like:

```
10010011001100111110000111011111
```

Some bits in the instruction form the `opcode`, the operation code. The opcode is a unique identifier for an operation, like adding integers. Other bits form the `operand`(s), the data on which to operate. The operand can be where the data is stored (the name of a register or an address in memory), or where to store the result of the operation (again, the name of a register or an address in memory).

For human readability, there are notations to represent these binary instructions. Here is an example of a set of instructions in a human-readable form:

```
.global main
main:
addi   sp, sp, -16
sd     t0, 0(sp)
sd     t1, 8(sp)
call   some_function
ld     t0, 0(sp)
ld     t1, 8(sp)
# Use t0 and t1 here as if nothing happened.
```

```
addi   sp, sp, 16
```

Programs are structured into sections. They include code sections, which contain a list of instructions, and data sections, which hold binary data such as text, images, or numbers that the program needs to use. Typically, there is a designated "main" section that contains the instructions to be executed first.

When you initiate an executable (with the exception of Mac "applications"), the binary data ("bits") are read from the hard disk and transferred to the main memory (RAM). The execution of the program begins when the first instruction from the "main" section is transferred to the CPU.

In this context, a computer program's code can be seen as an instance, or concrete representation, of an algorithm. Although the terms "algorithm" and "program" are distinct, they are often used interchangeably for simplification.

3.2 Comparing the Performance of Programs

When you compile or build a program, its code is converted into a series of instructions and data in memory. However, the execution time of the same program can vary across different machines due to differences in the processor's capabilities.

As each machine can potentially have a different processor, comparing the speed of programs can be a complex task, and is only meaningful when the processor is the same or standardized. Even when the processor is standardized, many factors affect performance -

- Background tasks on one machine can interfere with performance measurements.
- Even if you have the exact same processor, differences in manufacturing mean each can run at a different clock frequency.
- Small differences in ambient temperature affect how high a processor can clock.
- The mounting pressure of a cooler can affect heat transfer and in turn how high a processor can clock.
- Many cooling solutions involve vapor chambers. The orientation of vapor chambers can affect heat transfer and in turn how high a processor can clock.
- Even cosmic radiation can affect processors and memory.

Therefore, we usually prefer to compare algorithms instead. The methods of comparing algorithms will be discussed in the following sections of this chapter.

3.3 Analyzing Algorithms

One of the key components of this course is to provide a framework for predicting the performance of algorithms just by inspecting their structure. Let's dive into the process of analyzing an algorithm's time complexity.

3.3.1 Predicting Execution Time

Consider a simple method that adds two numbers:

```
public int add(int lhs, int rhs)
```

Suppose calling `add(2, 4)` takes 1 second. How long would `add(40, 50)` or `add(343245634, 32432423)` take? As you might expect, all these operations, despite the difference in magnitude of the numbers involved, take approximately the same time. That's because the time complexity of an addition operation does not depend on the values of the numbers but on the number of operations involved, which, in this case, is a single addition.

3.3.2 Impact of Input Size

Now, let's examine a slightly more complex method that sums up a list of numbers:

```
public int sumOfList(List<int> l)
```

Assuming that adding two numbers takes one second, how long would summing a list of 10 numbers take? We can infer that the time taken by `sumOfList` depends on the size of the list we provide. For instance, summing up a list of 10 numbers would take about half the time needed to sum up a list of 20 numbers.

Here's an implementation of `sumOfList`:

```
import java.util.ArrayList;

class Square {
  static int sumOfList(ArrayList<Integer> l) {
    int sum = 0;
    for (int i : l) {
      sum += i;
    }
    return sum;
  }
```

```
}
```

The key insight is that the execution time of this method depends on the number of elements in the list - which is the size of the input. The time complexity is directly proportional to the number of times the addition statement is executed, which is equal to the size of the list.

3.3.3 Iterations and Input Size

Let's take it a step further. If you're summing a list of N items, each of which is another list of M items, the operation would take N * M addition statements. Here, the time complexity is a function of both N and M.

3.3.4 Gauging Relative Execution Time

The crux of analyzing an algorithm's performance lies in understanding how many times statements in the program run as a function of the size of the input. This approach enables us to estimate how long two invocations of the same method will take relative to each other, given the size of the input for each.

Understanding this concept will allow you to better predict the performance of algorithms, which is a crucial skill in efficient programming and system design.

3.4 Algorithm Complexities

In order to evaluate an algorithm's efficiency, we analyze its time complexity and space complexity, both of which describe how the algorithm's performance scales with the size of the input.

3.4.1 Time Complexity

Time complexity measures how the execution time of an algorithm increases with the size of the input. For instance, counting how many times the number 5 appears in a list requires checking each number in the list. Hence, the time complexity is directly proportional to the size of the list.

3.4.2 Space Complexity

Space complexity quantifies the amount of memory an algorithm requires relative to the size of the input. Using the previous example, we would need enough space to store the list and an additional space to store the counter. Thus, the space complexity is proportional to the size of the list, or more precisely, $N + 1$.

3.4.3 Conditional Statements and Complexity

In cases where conditional statements are present, the number of executed statements depends on which branch the program takes. One branch may contain more statements than the other. To handle such scenarios, we introduce the concept of Big-O, Big-Ω, and Big-Θ notations.

3.4.3.1 Big-O Notation

The Big-O notation describes the worst-case time complexity of an algorithm, essentially providing an upper bound on the time taken. This notation considers the scenario where the program consistently takes the path with the most statements. While Big-O is commonly used for time complexity, it can also describe space complexity.

The notation comprises two parts: the function itself and the variable representing the input size. Generally, 'n' is used to represent the input size, and constants and coefficients are typically ignored. For instance, the following functions are all $O(n)$:

- $n + 1$
- $2n$
- $103n + 124$

  Only the highest degree of 'n' is considered when determining Big-O notation. Therefore, functions such as n^2, $n/2$, or $\sqrt{n}$ are not considered O(n).

3.4.3.2 Big-Ω Notation

The Big-Ω notation represents the best-case complexity of an algorithm. It follows the same format as Big-O notation but focuses on the scenario where the program consistently takes the path with the fewest statements.

3.4.3.3 Big-Θ Notation

The Big-Θ notation is used to denote the average-case complexity of an algorithm. It again follows the same structure as Big-O, but it considers both the best and worst-case scenarios to provide an average estimate of the algorithm's performance.

Remember, these notations and complexities are pivotal in estimating the performance of an algorithm based on the size or other properties of the input, correlating to the number of steps taken by the algorithm. This understanding is crucial when designing efficient and effective solutions in computer science.

3.5 Common Big-O Complexities

Big-O notation is a way of expressing the worst-case time complexity of an algorithm. It describes how the running time of an algorithm changes as the size of its input grows. The most common Big-O complexities are:

3.5.1 Constant Complexity and Growth

Constant complexity, often represented as $O(1)$, occurs when the running time of an algorithm or the amount of work needed does not change with the size of the input (N). This means the algorithm takes a fixed amount of time, regardless of how many elements it is processing.

For example, accessing an element in an array by its index is an operation of constant complexity. This is because it takes roughly the same amount of time, regardless of the size of the array:

- $f(n) = 1$ for all n

In this case, no matter how large or small our input size is, the amount of work we have to do remains the same. This is the most efficient complexity an algorithm can have.

3.5.2 Linear Complexity and Growth

Linear complexity, often represented as $O(n)$, occurs when the running time of an algorithm or the amount of work needed scales proportionally with the size of the input (N). For every additional element in the input, a fixed amount of work is added.

For example, finding an element in an unsorted list is an operation of linear complexity, as the algorithm might need to look at every element once:

- $f(n) = n$

In this case, if we add one more element to our input size, we add one more unit of work. This is because every additional element requires the same amount of work.

3.5.3 Quadratic Complexity and Growth

Quadratic complexity, often represented as $O(n^2)$, occurs when the running time of an algorithm or the amount of work needed scales with the square of the size of the input (N). For every additional element in the input, the work increases by a factor of n.

For example, a simple nested loop for comparing pairs of elements in a list has quadratic complexity. This is because each element is compared to every other element:

- $f(n) = n^2$

In this case, if we add one more element to our input size, we add n units of work, as we have to compare this new element with every other element already in the list.

3.5.4 Exponential Functions and Growth

An exponential function is one that includes a variable in the exponent. To illustrate, the function 2^n is an exponential function. Here, with each unit increase in the input, the output is multiplied (or scaled up) by a factor of 2. For instance:

- $f(1) = 2^1 = 2$
- $f(2) = 2^2 = 4$
- $f(n + 1) = f(n) * 2$

Exponential growth is characterized by a constant factor scaling up the running time of the algorithm, or the amount of work needed, for each unit increase in the size of the input (often denoted as N). This constant factor is the base of the exponent. This means that for an algorithm with a time complexity of 2^N, adding one more element to the input could potentially double the amount of work required. Examples of algorithms exhibiting exponential growth include certain solutions to problems like the Towers of Hanoi and calculations of the Fibonacci sequence.

3.5.5 Logarithmic Functions and Growth

Logarithmic functions serve as the inverse of exponential functions. For example, $log_2(n)$ is a logarithmic function. Here, the output is decremented by 1 each time the input is divided (or scaled down) by a factor of 2. For instance:

- $f(16) = log_2(16) = 4$

- $f(8) = log_2(8) = 3$
- $f(n/2) = f(n) - 1$

Logarithmic growth is characterized by a decrement of 1 in the running time of the algorithm or the amount of work needed, each time the size of the input (denoted as N) is divided by a constant factor. This constant factor is the base of the logarithm. For an algorithm with a time complexity of $log_2(n)$, if we have an input size of 16, doubling the input size will only increase the amount of work by 1 unit. Similarly, halving the input size will decrease the work by 1 unit.

Algorithms often display complexities such as $N * log_2(N)$, which represents a combination of linear and logarithmic growth. Examples of such algorithms that include logarithmic growth are binary search and certain sorting algorithms.

3.5.6 Examples

Let's look at some examples of algorithms and their Big-O complexities.

```
static int findMin(x, y) {
  if (x < y) {
    return x;
  } else {
    return y;
  }
}
```

This algorithm finds the minimum of two numbers x and y. It does not depend on the input size, since it only performs one comparison and one return statement. Therefore, its worst-case complexity is $O(1)$. Its best case and average case are also $O(1)$ since they are the same as the worst case.

```
static int linearSearch(numbers[], target)
  for (int i = 0; i < numbers.length; i++) {
    if (numbers[i] == target) {
      return i;
    }
  }
  return -1;
}
```

This algorithm performs a linear search on an array of numbers to find a target value. It iterates through each element of the array until it finds the target or reaches the end of the array. In the worst case, it has to check every element of the array, which means its worst-case

complexity is $O(n)$, where n is the length of the array. In the best case, it finds the target in the first element, which means its best-case complexity is $O(1)$. In the average case, it finds the target somewhere in the middle of the array, which means its average-case complexity is also $O(n)$.

We will talk about more examples of other common worst-case complexities throughout this course.

3.6 Growth Rate

The growth rate for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows. The cost can be measured in terms of time, space, or other resources. The worst-case complexity notation essentially denotes the growth rate of the time complexity with respect to the size of a worst-case input.

The table below summarizes how different Big-O complexities compare in terms of their growth rates.

Complexity	Growth Rate
O(1)	Constant
O($log(n)$)	Logarithmic
O(n)	Linear
O(n^2)	Quadratic
O(2^n)	Exponential

As we can see, constant and logarithmic complexities have very low growth rates, meaning that they are very efficient and scalable algorithms. Linear complexity has a moderate growth rate, meaning that it can handle reasonably large inputs but may become slow for very large inputs. Quadratic and exponential complexities have very high growth rates, meaning that they are very inefficient and unscalable algorithms that can only handle small inputs.

3.7 Summary / Review

In this section, we learned how to analyze the performance of algorithms using Big-O notation. We saw that measuring the actual running time of an algorithm on real hardware is difficult and impractical because it depends on many factors such as the hardware specifications, the programming language, the compiler, the input size, and the input distribution. Therefore, we need a way to simplify and standardize the performance analysis across different hardware and software platforms.

We learned that one way to simplify the performance analysis is to count the number of steps or instructions that an algorithm needs to execute before finishing. We saw that these steps are analogous to the instructions generated by a compiler when it translates our code into machine code. We also learned that different steps may have different costs depending on their complexity, but we can ignore these differences for simplicity and focus on the overall number of steps.

We learned that Big-O notation is a mathematical tool that allows us to express the worst-case complexity of an algorithm. It describes how the number of steps grows as a function of the input size in the worst possible scenario. It gives us an upper bound on the performance of an algorithm, meaning that it tells us how slow an algorithm can get in the worst case. We also learned that Big-O notation ignores constant factors and lower-order terms because they become insignificant as the input size grows.

We learned about some common Big-O complexities and their growth rates, such as constant, logarithmic, linear, quadratic, and exponential. We saw some examples of algorithms and their Big-O complexities, and how to analyze them using simple rules such as adding complexities for sequential steps, multiplying complexities for nested steps, and taking the maximum complexity for conditional steps.

We learned that Big-O notation helps us compare different algorithms and choose the most efficient one for a given problem. It also helps us estimate how well an algorithm can scale to larger inputs and how it can affect the performance of our applications.

Part III

Unit Testing

4 Unit Testing and Test-Driven Development

4.1 Background

Imagine you are an architect and you've just designed a large, complex building - let's say a skyscraper. This skyscraper is not merely a single entity; it's composed of thousands of individual components - the plumbing, the electrical wiring, the elevators, the heating system, and the building's structural elements, among many others. Now, how would you ensure that the skyscraper works as intended? Would you wait until the entire building is constructed and then start testing every possible scenario? Obviously, this approach is time-consuming, and it exposes you to significant risk.

A similar challenge exists in the world of software development. Take, for example, a web browser. This is a complex piece of software with hundreds of classes interacting in intricate ways. These classes and methods perform various tasks such as rendering HTML and CSS, processing JavaScript, managing cookies, implementing security features, and many others. Ensuring the correct functionality of this software is a daunting task, given the vast range of potential inputs. After all, there are billions of web pages on the internet, each with its unique combination of technologies, designs, and user interactions. How can you guarantee that your browser works flawlessly with all of them? A naive approach would be to load each web page and observe the output, but this process is not only time-consuming but also practically impossible.

This conundrum begs the question: How can we validate the correct functionality of a software product efficiently? The direction points towards automation - the ability to conduct tests without manual intervention. But how can we achieve this, especially given the enormous application surface area?

This is where unit testing and Test-Driven Development (TDD) come in.

4.2 The Power of Unit Testing

While the surface area of an entire application is vast, the surface area of individual classes and units of code within the software project is significantly smaller. If we can write tests to verify that each method within each class functions correctly for all possible inputs, we reduce the complexity of the problem.

At first glance, it might seem like an overwhelming task. Even a moderately complex software project can have thousands of methods spread across hundreds of classes. Writing tests for all of them could result in thousands of test cases. But this is precisely where automation proves its worth. By automating these tests, we can execute them each time we modify our code, ensuring the functionality remains intact. This method gives us confidence that our changes have not inadvertently introduced bugs into existing functionality.

The key principle here is that by ensuring each individual unit of our software behaves correctly, we can be reasonably confident that the application as a whole operates as expected, provided the software architecture is sound. In this manner, unit testing allows us to break down the monumental task of verifying a complex software system's functionality into manageable, automated tasks.

In the following sections, we will delve deeper into the concept of unit testing, its implementation in Java, and the practice of Test-Driven Development, where tests actually guide and shape the development of the software. Buckle up, for we're about to embark on an exciting journey that will fundamentally change how you approach software development!

4.3 A Basic Approach to Unit Testing

In order to illustrate the process of unit testing in Java, let's consider a simple utility class named `MathUtil`. This class defines basic arithmetic operations such as `add`, `subtract`, etc.

```
public class MathUtil {
    public int add(int a, int b) {
        return a + b;
    }

    // More methods for subtract, multiply, etc.
}
```

As we discussed in the previous section, to ensure our `MathUtil` functions correctly, we associate it with a `MathUtilTest` class. This class contains multiple test methods, each designed to verify a different scenario of the operations provided by `MathUtil`.

```
public class MathUtilTest {

    public boolean testAdd1() {
        MathUtil m = new MathUtil();
        int lhs = 5;
        int rhs = 7;
```

```
        if (m.add(lhs, rhs) == lhs + rhs) {
            return true;
        } else {
            return false;
        }
    }

    // More test methods for other cases...
}
```

In the above example, the `testAdd1` method tests the addition of two positive numbers. We could also add methods like `testAdd2` to test adding a positive and a negative number, `testAdd3` to test adding two negative numbers, and so forth. Each of these methods tests a specific scenario and validates that the result is as expected.

4.4 Recognizing the Inefficiencies and Redundancies

While the above approach accomplishes our objective of validating the methods in our `MathUtil` class, you might have already noticed that it's far from optimal. There are several glaring issues that can make this method tedious and inefficient:

4.4.1 Redundancy

Every test method follows a similar pattern - we perform an operation and then verify if the result matches the expected outcome. This redundancy suggests we could abstract out the verification part into a separate method.

In the following expanded `MathUtilTest` class, you can observe that `testAdd2` and `testAdd3` follow the exact same pattern as `testAdd1`. They create an instance of `MathUtil`, perform an operation, and then compare the result with the expected outcome. This repetitive pattern across multiple tests highlights the redundancy and inefficiency of this approach.

```
public class MathUtilTest {

    public boolean testAdd1() {
        MathUtil m = new MathUtil();
        int lhs = 5;
        int rhs = 7;

        if (m.add(lhs, rhs) == lhs + rhs) {
```

```
            return true;
        } else {
            return false;
        }
    }

    public boolean testAdd2() {
        MathUtil m = new MathUtil();
        int lhs = -5;
        int rhs = 7;

        if (m.add(lhs, rhs) == lhs + rhs) {
            return true;
        } else {
            return false;
        }
    }

    public boolean testAdd3() {
        MathUtil m = new MathUtil();
        int lhs = -5;
        int rhs = -7;

        if (m.add(lhs, rhs) == lhs + rhs) {
            return true;
        } else {
            return false;
        }
    }

    // More test methods for other cases...
}
```

4.4.2 Lack of Automation

Let's see how we need to currently run the tests we've written.

```
public class MathUtilTest {

    // testAdd1, testAdd2, testAdd3, etc. test methods...
```

```
    public static void main(String[] args) {
        MathUtilTest test = new MathUtilTest();

        System.out.println("testAdd1 result: " + (test.testAdd1() ? "PASS" : "FAIL"));
        System.out.println("testAdd2 result: " + (test.testAdd2() ? "PASS" : "FAIL"));
        System.out.println("testAdd3 result: " + (test.testAdd3() ? "PASS" : "FAIL"));

        // add more prints for other test cases
    }
}
```

With this `main` method, you can now run the `MathUtilTest` class, and it will execute each of the `testAdd` methods and print whether each test passed or failed. This method is a basic way to manually execute the tests and check their results.

Currently, we need to call each test method manually to run our tests. An automated system that could execute all tests for us would save time and reduce the chances of human error. However, as we will see later, there are better approaches to automation than the one we've used here.

4.5 A slightly more sophisticated approach to Unit Testing

To alleviate the redundancy, we can create a method that compares the expected and actual results and raises an error if they do not match. This method, which we can call `assertEquals`, would look something like this:

```
public static void assertEquals(String testCaseName, int expected, int actual) {
    if (expected != actual) {
        System.out.println(testCaseName + " result: FAIL");
    } else {
        System.out.println(testCaseName + " result: PASS");
    }
}
```

Then we can simplify our test methods by using `assertEquals`:

```
public void testAdd() {
    MathUtil m = new MathUtil();

    assertEquals("testAddTwoPositive", m.add(5, 7), 13);
    assertEquals("testAddTwoNegative", m.add(-5, -7), -13);
```

```
    assertEquals("testAddNegPos", m.add(-5, 7), 2);
}
```

Now, our test case looks cleaner and easier to understand. The `assertEquals` method abstracts away the comparison details, leaving only the test logic in the test case. We can apply this simplification to all our test methods.

This approach significantly reduces the redundancy in our test code, making it easier to write and maintain our tests. However, we are still manually running each test method from the `main` method. What if we could also automate the execution of all test methods?

4.6 Automating Test Execution

What if we could just call a single method that runs all our test methods? Let's define a simple `runTests` method that does exactly that.

```
public void runTests() {
    testAdd();
    // Call all other test methods here...
}
```

And then you can simply call the `runTests` method to execute all your tests:

```
public static void main(String[] args) {
    MathUtilTest test = new MathUtilTest();
    test.runTests();
}
```

This approach is an improvement over manually running each test. However, it still has some drawbacks. For instance, when you add a new test method, you need to remember to add a call to this method in the `runTests` method. It would be better if our test framework could automatically detect and run all test methods without requiring any modifications to the `runTests` method. As we'll see later, this is precisely what test frameworks like JUnit offer.

4.7 Summary

So far, we have seen how unit testing can be a powerful tool in ensuring that individual units of code within a larger software application function as expected. We have also discussed and

implemented a basic system for automating unit tests in Java, gradually refining this system to make it more efficient and less redundant.

In the following sections, we will discuss JUnit, a popular unit testing framework in Java that takes automation and convenience to the next level. We will also explore the practice of Test-Driven Development, where we let our tests guide the development of our software, helping us to write cleaner, more robust code. Stay tuned!

5 Writing JUnit Tests and Test-Driven Development

Unit testing is crucial to ensure the accuracy and performance of your code. But as we've seen, managing and writing tests can be a bit cumbersome. This is where testing frameworks, like JUnit, come in. They automate the tedious parts of testing and provide us with a plethora of tools to write effective tests.

5.1 Introduction to JUnit

JUnit is a widely used testing framework in the Java world. It automates the process of running tests and provides us with a wide range of assertion methods to validate our code. JUnit helps to simplify our test code, making it easier to read and maintain.

So, why is JUnit so popular?

1. **Simplicity:** JUnit simplifies the process of writing and running tests. The framework handles the boilerplate code, allowing us to focus solely on writing the test cases.
2. **Assertion Library:** JUnit provides a comprehensive set of assertion methods that help us validate our code against a wide range of conditions.
3. **Annotations:** JUnit uses annotations to define test methods and setup methods, making our test code easier to read and understand.
4. **Automatic Test Discovery:** JUnit automatically finds and runs all test methods, so we don't have to manually list them in our code.
5. **IDE Integration:** Most modern IDEs provide first-class support for JUnit, including features such as generating test cases and displaying test results in a friendly format.

Now that you understand what JUnit is and why it's beneficial let's see how to use it in our `MathUtil` class.

5.2 Useful JUnit Assertions

JUnit provides a set of methods called assertions that are used to test the expected output of your code. These assertions help verify that your code behaves as expected under different conditions.

Let's take a look at some commonly used assertions:

5.2.1 `assertEquals`

This assertion checks if two values are equal:

```
assertEquals(expected, actual);
```

If `actual` is not equal to `expected`, the assertion fails, and the test method will terminate immediately.

Let's rewrite our `addTest1` method using JUnit's `assertEquals`:

```
@Test
public void addTest1() {
    MathUtil m = new MathUtil();
    int lhs = 5;
    int rhs = 7;

    assertEquals(lhs + rhs, m.add(lhs, rhs));
}
```

5.2.2 `assertTrue` and `assertFalse`

These assertions verify if a condition is `true` or `false`, respectively:

```
assertTrue(condition);
assertFalse(condition);
```

If the `condition` does not meet the expectation (i.e., `true` for `assertTrue` and `false` for `assertFalse`), the assertion fails, and the test method will terminate immediately.

5.2.3 `assertNotNull` and `assertNull`

These assertions check if an object is `null` or not:

```
assertNotNull(object);
assertNull(object);
```

If the `object` does not meet the expectation (i.e., not `null` for `assertNotNull` and `null` for `assertNull`), the assertion fails, and the test method will terminate immediately.

These are just a few examples. JUnit provides a comprehensive set of assertions to cover almost any condition you might want to verify.

5.3 Setting Up JUnit

Before you can use JUnit, you need to make sure the library is on your classpath. This process can vary depending on the IDE and build system you're using.

For our labs, we will ensure the `JUnit` library is on our classpath by pre-configuring the project and IDE for you. However, if you're working on your own project, you'll need to add the `JUnit` library to your project's classpath.

When working on your own projects, you might be interested in using a build system like Maven or Gradle to manage your dependencies. These build systems make it easy to add and manage dependencies in your project. For example, if you're using Maven, you can add the following dependency to your `pom.xml` file:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>


 <version>5.7.0</version>
    <scope>test</scope>
</dependency>
```

This will automatically download the JUnit library and add it to your project's classpath.

Regardless of how you added the `JUnit` library to your project, next, we need to import the necessary classes and annotations from JUnit. At the top of our `MathUtilTest` class, we add the following import statements:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
```

The first import statement statically imports all assertion methods from `Assertions`, allowing us to use them directly in our code. The second import statement imports the `Test` annotation, which we use to denote our test methods. The third import statement imports the `BeforeEach` annotation, which we'll discuss in a moment.

5.4 Utilizing `@Test` and `@BeforeEach` Annotations

In JUnit, we use the `@Test` annotation to indicate that a method is a test method. This allows JUnit to automatically discover and run this method as a test.

```
@Test
public void addTest1() {
    // test code...
}
```

However, what if we have some setup code that we want to run before each test? This is where the `@BeforeEach` annotation comes in. Any method annotated with `@BeforeEach` will be run before each `@Test` method.

Let's say we want to create a new `MathUtil` instance before each test:

```
MathUtil m;

@BeforeEach
public void setup() {
    m = new MathUtil();
}
```

Now, before each test method is run, JUnit will first execute the `setup` method, ensuring that we have a fresh `MathUtil` instance for each test.

5.5 Interpreting JUnit Test Runner Output

Understanding the output of the JUnit test runner is crucial for interpreting the results of your tests. This helps you diagnose issues in your code and identify exactly what went wrong.

Let's analyze the output of the `junit-platform-console-standalone` test runner to get a feel for how this works.

```
   JUnit Jupiter
   JUnit Vintage
      BinarySearchTreeHiddenTest
         testInsertAndSearch
         testDeleteSingleNode
         testTreeTraversals  expected:<[2, 3, 4, 5, [6, 7], 8]> but was:<[2, 3, 4, 5, [7, 6
         testContainsElementNotInTree
         testContainsEmptyTree
         testDeleteEmptyTree
         testContainsElementInTree
         testSearchElementInTree
         testInsertMultipleElements
         testDeleteDuplicateElements
         testDeleteElementNotInTree
         testInsertNegativeNumbers
         testInsertAndSize
         testSearchEmptyTree
         testDeleteNodeWithMultipleElements
         testInsertDuplicatesAndRemove
         testInsertSingleElement
         testSearchElementNotInTree
   JUnit Platform Suite

Failures (1):
   JUnit Vintage:BinarySearchTreeHiddenTest:testTreeTraversals
     => org.junit.ComparisonFailure: expected:<[2, 3, 4, 5, [6, 7], 8]> but was:<[2, 3, 4,
        DataStructures.BinarySearchTreeHiddenTest.testTreeTraversals(BinarySearchTreeHidden
        [...]
```

The JUnit console output provides a tree structure representing the test execution. The topmost nodes represent the test engines used, in this case, `JUnit Jupiter` and `JUnit Vintage`. Underneath each engine are the individual test classes, such as `BinarySearchTreeHiddenTest`.

Within each test class node, there are child nodes representing each test method, such as `testInsertAndSearch` or `testDeleteSingleNode`. These methods are marked with a symbol if they passed, and with a symbol if they failed. In this case, we see that `testTreeTraversals` has failed.

Accompanying the failure symbol is a brief description of the failure, which is the assertion

message from the test method. In this example, the test expected the array `[2, 3, 4, 5, 6, 7, 8]`, but received the array `[2, 3, 4, 5, 7, 6, 8]`. This discrepancy caused the test to fail.

After the tree structure, there is a section titled `Failures` which provides more detailed information about each failure. For each failure, it lists:

1. The test class and method that failed.
2. The type of assertion failure that occurred, which is `org.junit.ComparisonFailure` in this case.
3. The detailed assertion failure message, which is the same as what's shown in the tree structure.
4. The location in the code where the failure occurred, which can be very useful (and is often the first thing you should look at when debugging a test failure). In this case, the failure occurred on line 220 of `BinarySearchTreeHiddenTest.java` (see the `DataStructures.BinarySearchTreeHiddenTest.testTreeTraversals(BinarySearchTreeHiddenTest.` line).

5.6 Conclusion

In conclusion, JUnit simplifies the process of writing and managing tests. It provides a comprehensive set of assertion methods to verify our code and uses annotations to define and organize our tests, making them easier to read and understand. By taking advantage of these features, we can write more effective and maintainable tests. In the next section, we'll dive deeper into Test-Driven Development, a methodology that leverages the power of testing to guide and improve the development process.

5.7 Test-Driven Development

Test-Driven Development (TDD) is a software development methodology that is centered around the idea of writing tests before writing the actual code. It is a highly disciplined process that follows a strict order of operations: red, green, refactor. This method has profound implications on the design, quality, and reliability of the software.

Let's dive into what these steps entail.

1. **Red**: Write a test that covers a specific functionality you want to implement. This test should fail initially because you haven't written the actual code yet. This stage helps you think about the functionality in detail, ponder on the inputs and expected outputs, and outline the structure of your code.

2. **Green**: Write the minimal amount of code needed to pass the test. At this stage, don't worry about the elegance of your code. Your primary focus is on functionality. Run your test, and it should pass this time.

3. **Refactor**: Refactor the code you just wrote in the green stage to eliminate duplication, improve readability, and ensure the code adheres to the best practices. After refactoring, all tests should still pass. If a test fails, it means the refactoring broke the functionality, and you need to revise your changes.

This cycle repeats for every small chunk of functionality you add to your software. With this approach, you are incrementally building your software with the assurance that at each step, the implemented functionality is working as expected.

5.7.1 The Motivation Behind Test-Driven Development

You might be wondering, why would you want to put in the extra effort to write tests before writing the actual code? Here are a few motivating factors:

1. **Confidence**: With TDD, you can be confident that your code works because you have tests that prove it. This confidence is especially important when you need to modify your code later. Changes can break existing functionality, but with a robust set of tests, you can quickly catch and fix these regressions.

2. **Better Design**: Writing tests first forces you to think about your code from a user's perspective. This shift in viewpoint often results in better code organization and modularity because you design your code to be easy to test, which typically means it is also easy to use and modify.

3. **Documentation**: Tests act as a form of documentation that shows how the code is supposed to work. New team members can look at the tests to understand what each function is supposed to do and what edge cases it handles.

4. **Development Speed**: While TDD might seem to slow you down at the beginning, it typically results in faster development in the long run. With TDD, you spend less time debugging and fixing bugs because you catch them early in the development process, before they become entangled with other parts of the code.

In conclusion, TDD is a powerful methodology that can significantly improve the quality of your code and your efficiency as a developer. While it might seem difficult at first, with practice, it becomes a natural part of the development process.

6 Introduction to Mutation Testing

6.1 Unit Testing, a Recap

Writing *correct* code is challenging. Owning your code, maintaining it, and ensuring it always performs as expected can be even more difficult. After all, we are human, and mistakes are a part of our nature. Therefore, we can't always rely solely on our ability to catch these mistakes. Instead, automated processes that can help us identify and correct these errors are truly invaluable.

Unit testing is one such automated process that tests your code. These tests target individual units of your code and are designed to verify their behavior. In programming, a method can be thought of as the smallest unit of code, and Classes, Packages, and Modules are larger units of code. Every unit test is designed to validate a single behavior of a single unit of code.

Unit testing is so crucial that they are a mandatory requirement for many software development projects. For instance, if you are contributing a Java source class to most projects, you are also required to provide unit tests for that class. In this course, where we deal with Data Structures and Algorithms, learning how to implement them in Java, etc., submitting your implementations without Unit tests would be like submitting a paper without a bibliography. It might make us question, "Sure, you wrote this and it looks convincing, but where's your evidence? Why should I believe you?"

And that's why you will be required to write unit tests for your Data Structures and Algorithms implementations. But then, how do you know that your unit tests are correct?

Consider this example:

```
@Test
public void testAdd() {
   assertTrue(1 == 1);
}
```

The test passes, but does it mean it's correct? Let's examine the test.

6.2 Writing *Correct* Unit Tests

Taking a closer look at the unit test above, two problems become evident:

- The `add` method isn't called in the test.
- The test can't fail.

To address the first issue, we can leverage tools that detect when a unit test doesn't cover a particular line of code. Here "cover" means "execute".

So what about a test like this?

```
@Test
public void testAdd() {
   assertTrue(add(1, 1) == add(1, 1));
}
```

The `add` method is called, and test coverage is at 100%. However, this test can't fail, meaning it is not *correct*.

6.3 Mutation Testing: A Solution to Test the Correctness of Your Unit Tests

Mutation testing provides a way to test the correctness of your unit tests. It can help you find tests that can't fail, tests that need more test cases, and even logic errors in your code.

The basic idea of mutation testing is as follows: if you claim your source code is correct, and that the tests prove it, mutation testing will challenge that claim. Mutation testing introduces small changes, called *mutations*, to your source code. If your tests still pass after these changes, it suggests that your test or source code may not be correct.

These mutations are created by algorithms called *mutators* or *mutation operators*. Each time a mutator runs, it receives a fresh copy of your source code and makes only one change. The result of a mutator is a *mutant* - a mutated version of your source code.

6.4 Mutation Testing in Practice

Here's how mutation testing works in practice:

- Apply a set of mutators to your source code to produce a collection of mutants.
- For each mutant, run your unit tests.

- If the unit tests pass, the mutant has *survived*.
- If the unit tests fail, the mutant was *killed*

.

The aim is to kill as many mutants as possible. If too many mutants survive, it indicates that your unit tests are not sufficient to prove that your source code is correct. If you manage to kill all mutants, you can use your unit tests to argue that your source code is indeed correct.

In our course, only unit tests that leave no surviving mutants will be accepted as a valid submission.

6.4.1 What are these mutators?

Mutators introduce specific types of changes to your code. For instance, a primitive returns mutator (PRIMITIVE_RETURNS) replaces int, short, long, char, float, and double return values with 0. So, for example, this method:

```
public int add(int lhs, int rhs) {
    return lhs + rhs;
}
```

becomes:

```
public int add(int lhs, int rhs) {
    return 0;
}
```

This mutant would survive in two cases: if the `add` method is never tested or if the only test cases for `add` are ones where the result is 0. To fix this, you can add more test cases that test `add` with non-zero results.

Another example of a mutator is the Remove Conditionals Mutator (REMOVE_CONDITIONALS), which removes all conditional statements such that the guarded statements always execute.

For example, this code:

```
if (a == b) {
  // do something
}
```

becomes:

```
if (true) {
  // do something
}
```

This mutant would survive if the source method is never tested, if all test cases for the source method only ever test the true case, or if both branches have the exact same code or equivalent code, to begin with. To fix this, you can add more test cases that test the false case, and ensure that the true and false branches are not equivalent.

Through mutation testing, you will get valuable feedback to improve your code and tests, making them more robust and reliable.

Remember, a well-tested code base is not just about coverage—it's about the quality of tests, their ability to catch mistakes, and their resilience against possible errors. Mutation testing is an invaluable tool in achieving this.

Certainly, let's decipher the feedback from the Coding Rooms.

6.5 Understanding CodingRooms Feedback

In the feedback provided by the autograder in Coding Rooms, you can find two main parts. The first part lists the mutation tests that have been run, and the second part provides a grading overview.

6.5.1 Understanding Mutation Test Feedback

Each mutation test feedback starts with "Running Mutation tests", followed by details about each mutation test performed.

```
Running Mutation tests -
Ran mutation tests for Calculator.CalculatorTest -
-[ RECORD 0 ]---------+-----------------------------------
Mutation type         | RemoveConditionalMutator_EQUAL_ELSE
Source method mutated | divide
Line no. of mutation  | 56
Test examined         | None
Result                | SURVIVED
```

In the example above, the autograder has run a mutation test on the "divide" method of your "Calculator" class. Here is what each line in the record means:

- **Mutation type:** The type of mutation that was made to your code. In this case, a RemoveConditionalMutator_EQUAL_ELSE mutation was made. This mutation type removes a conditional (==) operator and always takes the else branch.
- **Source method mutated:** The method in your code that was mutated for the test. In this case, it's the "divide" method.
- **Line no. of mutation:** The line number in your code where the mutation was applied.
- **Test examined:** This line indicates which test case was run against the mutant. "None" means no specific test case was chosen, and all available tests were run.
- **Result:** The outcome of the mutation test. If your tests fail against the mutated code (which is a good thing!), this line will read "KILLED". If your tests pass (which implies your tests didn't catch the error introduced by the mutation), this line will read "SURVIVED". In this case, the mutant has survived, indicating your tests didn't catch the error.

If any mutants survive, the autograder lists those under the line, "Problematic mutation test failures printed about."

6.5.2 Understanding the Grading Overview

The grading overview gives you a quick summary of how your code has performed in the tests. It breaks down the grading by requirements.

```
                  Grading Overview

 Requirement      Grade              Reason

      1        10.00/10.00     - 4/4 tests passing.

      2        32.00/40.00   -8 Penalty due to surviv
                                   ing muations

                Total: 42.00/50.00
```

In the example above, we have:

- **Requirement 1:** This section corresponds to the first requirement of the assignment. The student received a full score (10.00/10.00) because all 4 test cases associated with this requirement passed.

- **Requirement 2:** This section corresponds to the second requirement of the assignment. The student lost 8 points because of surviving mutations, resulting in a score of 32.00/40.00 for this requirement.

The total grade of the assignment is

42.00/50.00, indicating the need for further improvement.

Remember, each surviving mutation indicates a flaw in your test cases—they didn't catch an erroneous mutation. To improve your grade, aim to update your test cases to ensure that they effectively detect the mutations.

7 List of Mutators

7.1 Conditionals Boundary Mutator (CONDITIONALS_BOUNDARY)

The conditionals boundary mutator replaces the relational operators `<, <=, >, >=` with their boundary counterpart as per the table below.

Original conditional	Mutated conditional
<	<=
<=	<
>	>=
>=	>

{:.table }

For example

```
if (a < b) {
  // do something
}
```

will be mutated to

```
if (a <= b) {
  // do something
}
```

7.2 Increments Mutator (INCREMENTS)

The increments mutator will mutate increments, decrements and assignment increments and decrements of local variables (stack variables). It will replace increments with decrements and vice versa.

For example

```
public int method(int i) {
  i++;
  return i;
}
```

will be mutated to

```
public int method(int i) {
  i--;
  return i;
}
```

Please note that the increments mutator will be applied to increments of **local variables only**. Increments and decrements of member variables will be covered by the Math Mutator.

7.3 Invert Negatives Mutator (INVERT_NEGS)

The invert negatives mutator inverts negation of integer and floating point numbers. For example

```
public float negate(final float i) {
  return -i;
}
```

will be mutated to

```
public float negate(final float i) {
  return i;
}
```

7.4 Math Mutator (MATH)

The math mutator replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation. The replacements will be selected according to the table below.

Original conditional	Mutated conditional
+	-
-	+
*	/
/	*
%	*
&	|
|	&
^	&
«	»
»	«
»>	«

{:.table}

For example

```
int a = b + c;
```

will be mutated to

```
int a = b - c;
```

Keep in mind that the `+` operator on `String`s as in

```
String a = "foo" + "bar";
```

is **not a mathematical operator** but a string concatenation and will be replaced by the compiler with something like

```
String a = new StringBuilder("foo").append("bar").toString();
```

Please note that the compiler will also use binary arithmetic operations for increments, decrements and assignment increments and decrements of non-local variables (member variables) although a special `iinc` opcode for increments exists. This special opcode is restricted to local variables (also called stack variables) and cannot be used for member variables. That means the math mutator will also mutate

```
public class A {
  private int i;
```

```
  public void foo() {
    this.i++;
  }
}
```

to

```
public class A {
  private int i;

  public void foo() {
    this.i = this.i - 1;
  }
}
```

See the Increments Mutator for details.

7.5 Negate Conditionals Mutator (NEGATE_CONDITIONALS)

The negate conditionals mutator will mutate all conditionals found according to the replacement table below.

Original conditional	Mutated conditional
==	!=
!=	==
<=	>
>=	<
<	>=
>	<=

{:.table }

For example

```
if (a == b) {
  // do something
}
```

will be mutated to

```
if (a != b) {
  // do something
}
```

This mutator overlaps to a degree with the conditionals boundary mutator, but is less **stable** i.e these mutations are generally easier for a test suite to detect.

7.6 Return Values Mutator (RETURN_VALS)

This mutator has been superseded by the new returns mutator set. See Empty returns, False returns, True returns, Null returns and Primitive returns.

The return values mutator mutates the return values of method calls. Depending on the return type of the method another mutation is used.4

Return Type	Mutation
`boolean`	replace the unmutated return value `true` with `false` and replace the unmutated return value `false` with `true`
`int byte short`	if the unmutated return value is `0` return `1`, otherwise mutate to return value `0`

Return Type	Mutation
`long`	replace the unmutated return value `x` with the result of `x+1`
`float double`	replace the unmutated return value `x` with the result of `-(x+1.0)` if `x` is not `NAN` and replace `NAN` with `0`
`Object`	replace non-`null` return values with `null` and throw a `java.lang.RuntimeExcepti` if the unmutated method would return `null`

{:.table}

For example

```
public Object foo() {
  return new Object();
}
```

will be mutated to

```
public Object foo() {
  new Object();
  return null;
}
```

Please note that constructor calls are **not considered void method calls**. See the Constructor Call Mutator for mutations of constructors or the Non Void Method Call Mutator for mutations of non void methods.

7.7 Void Method Call Mutator (VOID_METHOD_CALLS)

The void method call mutator removes method calls to void methods. For example

```
public void someVoidMethod(int i) {
  // does something
}

public int foo() {
  int i = 5;
  someVoidMethod(i);
  return i;
}
```

will be mutated to

```
public void someVoidMethod(int i) {
  // does something
}

public int foo() {
  int i = 5;
  return i;
}
```

7.8 Empty returns Mutator (EMPTY_RETURNS)

Replaces return values with an 'empty' value for that type as follows

- java.lang.String -> “ ”
- java.util.Optional -> Optional.empty()
- java.util.List -> Collections.emptyList()
- java.util.Collection -> Collections.emptyList()
- java.util.Set -> Collections.emptySet()
- java.lang.Integer -> 0
- java.lang.Short -> 0
- java.lang.Long -> 0
- java.lang.Character -> 0
- java.lang.Float -> 0
- java.lang.Double -> 0

Pitest will filter out equivalent mutations to methods that are already hard coded to return the empty value.

7.9 False returns Mutator (FALSE_RETURNS)

Replaces primitive and boxed boolean return values with false.

Pitest will filter out equivalent mutations to methods that are already hard coded to return false.

7.10 True returns Mutator (TRUE_RETURNS)

Replaces primitive and boxed boolean return values with true.

Pitest will filter out equivalent mutations to methods that are already hard coded to return true.

7.11 Null returns Mutator (NULL_RETURNS)

Replaces return values with null. Methods that can be mutated by the EMPTY_RETURNS mutator or that are directly annotated with NotNull will not be mutated.

Pitest will filter out equivalent mutations to methods that are already hard coded to return null.

7.12 Primitive returns Mutator (PRIMITIVE_RETURNS)

Replaces int, short, long, char, float and double return values with 0.

Pitest will filter out equivalent mutations to methods that are already hard coded to return 0.

7.13 Constructor Call Mutator (CONSTRUCTOR_CALLS)

Optional mutator that replaces constructor calls with `null` values. For example

```
public Object foo() {
  Object o = new Object();
  return o;
}
```

will be mutated to

```
public Object foo() {
  Object o = null;
  return o;
}
```

Please note that this mutation is fairly unstable and likely to cause **`NullPointerExceptions`** even with weak test suites.

This mutator does not affect non constructor method calls. See Void Method Call Mutator for mutations of void methods and Non Void Method Call Mutator for mutations of non void methods.

Part IV

Iterative Sorting and Searching

8 Introduction to Sorting

8.1 Understanding the Sorting Problem

Sorting is an essential problem in computer science and software engineering. Simply put, sorting is the task of rearranging a set of items, such as an array, in a specific order. For instance, an array of integers might be sorted in ascending or descending order based on their numeric value.

One might wonder why sorting is so ubiquitous in the field of software engineering. The answer lies in the multitude of applications that it has. Sorting allows for more efficient searching, data compression, and organization. It is a crucial step in numerous algorithms and computational tasks. The efficiency of a sorting algorithm can significantly affect the performance of these applications.

8.2 Developing a Sorting Algorithm from Scratch

One might find the task of developing a sorting algorithm from scratch quite daunting. Sorting algorithms, like many computational problems, have an abstract and often complex nature that can make it difficult to understand, let alone create. If shown the solution, one might understand how it works and even memorize it, but the thought process behind creating such a solution can be elusive.

A common approach to overcoming such challenges is to simplify the problem. By breaking down a complex problem into simpler subproblems, we can gain insights into the problem's structure and ultimately develop a solution. This approach will be demonstrated as we develop a sorting algorithm.

8.3 Starting Simple: Sorting an Array of Size 1

The simplest possible sorting problem involves an array of size one. But, of course, this array is already sorted as it only contains a single element. Thus, we need to do nothing to sort this array. Here's how this can be implemented in Java:

```java
public static void sort1(int[] arr) {
  // do nothing
}
```

8.4 Sorting an Array of Size 2

What about an array of size two? In this case, we simply need to compare the two elements and swap them if they are out of order:

```java
public static void sort2(int[] arr) {
    if (arr[0] > arr[1]) {
      int temp = arr[0];
      arr[0] = arr[1];
      arr[1] = temp;
    }
}
```

8.5 Expanding the Problem: Sorting an Array of Size 3

When sorting an array of size three, we perform similar steps as we did for the array of size two. First, we compare and potentially swap the first two numbers. Then, we do the same for the second and third numbers. Finally, we compare the first two numbers once again:

```java
public static void sort3(int[] arr) {
    if (arr[0] > arr[1]) {
      int temp = arr[0];
      arr[0] = arr[1];
      arr[1] = temp;
    }
    if (arr[1] > arr[2]) {
      int temp = arr[1];
      arr[1] = arr[2];
      arr[2] = temp;
    }
    if (arr[0] > arr[1]) {
      int temp = arr[0];
      arr[0] = arr[1];
      arr[1] = temp;
```

```
    }
}
```

By following this process, we ensure that the array is sorted, regardless of the initial order of the elements.

8.6 Scaling Up: Sorting an Array of Size 4

We can extend the same approach to sort an array of size four. However, notice that once we have the largest number at the last position, the problem

reduces to sorting an array of size three. This realization will be crucial as we move forward.

```
public static void sort4(int[] arr) {
    for(int i = 0; i < 3; i++) {
      if (arr[i] > arr[i+1]) {
        int temp = arr[i];
        arr[i] = arr[i+1];
        arr[i+1] = temp;
      }
    }
    sort3(arr);
}
```

8.7 Recognizing the Pattern

As we incrementally solve larger versions of our sorting problem, we start to notice a pattern. For every array size, we first ensure the largest number ends up at the end of the array. Then, we apply the same sorting logic to the remaining elements in the array.

For instance, when sorting an array of size 5, we first find and move the largest number to the end of the array, then apply the sorting process for an array of size 4 to the remaining numbers:

```
public static void sort5(int[] arr) {
  for(int i = 0; i < 4; i++) {
    if (arr[i] > arr[i+1]) {
      int temp = arr[i];
      arr[i] = arr[i+1];
      arr[i+1] = temp;
```

```
    }
  }
  sort4(arr);
}
```

8.8 Generalizing the Algorithm

This pattern suggests a way to generalize our algorithm to sort an array of any size. For an array of size `n`, we first ensure the largest number ends up at the end, then recursively apply the same process to the remaining `n-1` elements. By repeatedly applying this process, we ensure that the entire array ends up sorted.

```
public static void sort(int[] arr) {
  for(int i = arr.length - 1; i > 0; i--) {
    for(int j = 0; j < i; j++) {
      if (arr[j] > arr[j+1]) {
        int temp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = temp;
      }
    }
  }
}
```

This algorithm, known as Bubble Sort, represents a simple yet effective solution to the sorting problem. It demonstrates how complex problems can often be broken down into simpler subproblems, providing a valuable lesson for problem-solving in computer science and beyond.

9 Comparable and Comparator

9.1 Sorting Complex Objects

In our previous lessons, we have primarily dealt with arrays of basic data types, such as integers, which are inherently easy to compare and sort. However, software development often involves sorting collections of complex objects, such as `Employee` or `Student` objects. These user-defined types lack a natural order for comparison, necessitating a custom solution.

9.2 Defining a Comparison Method

A rudimentary solution could involve appending a comparison method to our `Employee` or `Student` classes, like `compareSalary` or `compareGPA`. However, this design lacks versatility - each new attribute requires a fresh comparison method.

A superior solution employs the `Comparable` interface, comprising a single method, `compareTo`. This method, accepting an object of the same type, returns an integer indicating if the current object is lesser than, equal to, or greater than the input object.

The `Comparable` interface in Java is defined as follows:

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

9.3 Implementing Comparable

Implementing the `Comparable` interface in any class endows it with sortable attributes. For example, sorting `Student` objects by GPA could be achieved by implementing `Comparable<Student>` in the `Student` class, defining `compareTo` to compare GPAs:

```
public class Student implements Comparable<Student> {
    private double gpa;
    // other fields and methods...
```

```
    @Override
    public int compareTo(Student other) {
        return Double.compare(this.gpa, other.gpa);
    }
}
```

The array of `Student` objects can be sorted with our new comparison logic, `arr[j].compareTo(arr[j+1]) > 0`:

```
public static <T extends Comparable<T>> void sort(T[] arr) {
    for(int i = arr.length - 1; i > 0; i--) {
        for(int j = 0; j < i; j++) {
            if (arr[j].compareTo(arr[j+1]) > 0) {
                T temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

9.4 `Comparable` in the Standard Library

The `Comparable` interface facilitates the creation of versatile sorting functions, capable of sorting arrays of any type implementing `Comparable`. This applies to user-defined types, as well as many built-in types in the Java standard library.

Implementing the `Comparable` interface communicates that a class is able to be compared with other instances of its type. This standard behavior empowers methods to interact with objects more abstractly and flexibly.

The Java standard library provides a `Comparable` interface similar to the one defined earlier. Many built-in classes such as `Integer`, `Double`, and `String` already implement this interface, enabling their comparison and sorting with no additional code. Our `Student` class can be modified to use `java.lang.Comparable` in place of our custom interface, maintaining the same `compareTo` method for use in our generic sort function.

9.5 Sorting Flexibility with `Comparator`

The `Comparable` interface is limited to a single `compareTo` method per class. When multiple sorting methods are required, Java's `Comparator` interface comes into play. `Comparator` represents different orderings for a specific type, allowing multiple comparators per class.

For instance, a `Comparator` for `Student` objects that orders by GPA could look like this:

```
import java

.util.Comparator;

public class StudentGPAComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return Double.compare(s1.getGPA(), s2.getGPA());
    }
}
```

A modified version of our generic sort function, now accepting a `Comparator`, can compare elements of the array:

```
public static <T> void sort(T[] arr, Comparator<? super T> comparator) {
    for(int i = arr.length - 1; i > 0; i--) {
        for(int j = 0; j < i; j++) {
            if (comparator.compare(arr[j], arr[j+1]) > 0) {
                T temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

9.6 Understanding Natural and Total Orderings

Implementing `Comparable` endows a class with a *natural ordering*, a default ordering used in operations like sorting. For example, `String` objects follow lexicographic order and `Integer` objects adhere to numerical order. When `Comparable<Student>` is implemented to compare GPAs, we establish GPA as the natural ordering for `Student` objects.

Java's `Comparator` interface allows the definition of additional orderings, known as *total orderings*. While the natural ordering is default, total orderings, which also follow rules of completeness, transitivity, and antisymmetry, provide flexibility to define any suitable ordering.

The `Comparable` and `Comparator` interfaces epitomize the power of abstraction in computer science. By concentrating on the core concept of order and essential operations involving order, we can create general, flexible, and reusable code to work with a broad spectrum of data types and orderings.

In the next chapter, we will delve into how these concepts are employed in data structures like trees and heaps to manage data for efficient searching and sorting. For now, contemplate the elegance and versatility of the `Comparable` and `Comparator` interfaces, and the compelling concept of order they embody.

10 Bubble, Selection, and Insertion Sort

10.1 Bubble Sort

Bubble sort is a simple sorting algorithm that works by repeatedly comparing and swapping adjacent elements in an array until the array is sorted. Bubble sort is easy to implement and understand, but it is not very efficient for large arrays. The name bubble sort comes from the idea that the larger elements "bubble up" to the end of the array after each iteration.

10.1.1 Pseudocode

The pseudocode for bubble sort is as follows:

```
procedure bubbleSort(A : list of sortable items)
    n = length(A)
    repeat
        swapped = false
        for i = 1 to n-1 inclusive do
            if A[i-1] > A[i] then
                swap(A[i-1], A[i])
                swapped = true
            end if
        end for
        n = n - 1
    until not swapped
end procedure
```

The procedure takes an array `A` of sortable items and sorts it in ascending order. The procedure uses a variable `n` to keep track of the unsorted part of the array, and a variable swapped to indicate whether any swaps occurred in the current iteration. The procedure repeats until no swaps are made, which means the array is sorted.

10.1.2 An Example

1. Let us see an example of bubble sort on the following array:

   ```
   [5, 1, 4, 2, 8]
   ```

2. We start with `n = 5` and `swapped = false`. We compare the first and second elements, 5 and 1, and swap them since $5 > 1$. We set swapped to true to indicate that a swap occurred. We compare the second and third elements, 5 and 4, and swap them as well. We continue to compare and swap the adjacent elements until we reach the end of the array. The array after the first iteration is:

   ```
   [1, 4, 2, 5, 8]
   ```

3. We decrement `n` by 1, since the last element 8 is now in its correct position. We repeat the process with `n = 4` and `swapped = false`. We compare and swap the first and second elements, 1 and 4, since $1 < 4$. We compare and swap the second and third elements, 4 and 2, since $4 > 2$. We compare the third and fourth elements, 5 and 8, but do not swap them since $5 < 8$. The array after the second iteration is:

   ```
   [1, 2, 4, 5, 8]
   ```

4. We decrement `n` by 1, since the last element 5 is now in its correct position. We repeat the process with `n = 3` and `swapped = false`. We compare and swap the first and second elements, 1 and 2, since $1 < 2$. We compare and swap the second and third elements, 2 and 4, since $2 < 4$. The array after the third iteration is:

   ```
   [1, 2, 4, 5, 8]
   ```

5. We decrement `n` by 1, since the last element 4 is now in its correct position. We repeat the process with `n = 2` and `swapped = false`. We compare and swap the first and second elements, 1 and 2, since $1 < 2$. The array after the fourth iteration is:

   ```
   [1, 2, 4, 5, 8]
   ```

6. We decrement `n` by 1, since the last element 2 is now in its correct position. We repeat the process with `n = 1` and `swapped = false`. We do not compare any elements, since there is only one element left. The array after the fifth iteration is:

   ```
   [1, 2, 4, 5, 8]
   ```

7. We exit the loop, since `swapped` is false, which means the array is sorted.

10.1.3 Lab Instructions

1. The template uses Java generics to create a generic class `Main` that can sort arrays of any type that implements the `Comparable` interface. Generics are a way of implementing generic programming in Java, which allows you to write code that can work with different types of objects without casting or risking `ClassCastException`.
2. The constructor of the `Main` class takes an array of type `T` as a parameter and assigns it to the data field. The data field is also of type `T`, which means it can store any type of object that implements `Comparable`.
3. The `BubbleSort` method returns an array of type `T` that is sorted in ascending order using the bubble sort algorithm. The method uses a local variable `sorted` to store a copy of the data array and then modifies it using the pseudocode provided in the assignment instructions.
4. To compare and swap the elements of the `sorted` array, you need to use the `compareTo` method of the `Comparable` interface. The `compareTo` method returns a negative integer, zero, or a positive integer if the current object is less than, equal to, or greater than the specified object. For example, if you want to compare the elements at index `i-1` and `i`, you can write:

```
if (sorted[i-1].compareTo(sorted[i]) > 0) {
    // swap the elements
}
```

5. To swap the elements of the `sorted` array, you can use a temporary variable of type `T` to store one of the elements, and then assign the other element to its place. For example, if you want to swap the elements at index `i-1` and `i`, you can write:

```
T temp = sorted[i-1];
sorted[i-1] = sorted[i];
sorted[i] = temp;
```

6. After performing the bubble sort algorithm, the method returns the sorted array.
7. To test your code, you can create an object of the `Main` class with different types of arrays, such as `Integer`, `String`, or `Double`, and call the `BubbleSort` method on them. You can print the original and sorted arrays to check the output. For example, you can write:

```
Integer[] intArray = {5, 1, 4, 2, 8};
Main<Integer> intMain = new Main<>(intArray);
Integer[] intSorted = intMain.BubbleSort();
```

```
System.out.println("Original array: " + Arrays.toString(intArray));
System.out.println("Sorted array: " + Arrays.toString(intSorted));
```

The output should be:

```
Original array: [5, 1, 4, 2, 8]
Sorted array: [1, 2, 4, 5, 8]
```

10.2 Selection Sort

10.2.1 Introduction

Selection sort is a sorting algorithm that repeatedly finds the smallest element in the unsorted part of the array and swaps it with the first element of the unsorted part. This way, the array is divided into two subarrays: one that contains the sorted elements and one that contains the unsorted elements. The process is repeated until the entire array is sorted. Selection sort is a simple and efficient algorithm that works well for small arrays, but it has a high time complexity of O(n^2) for large arrays.

10.2.2 Pseudocode

The pseudocode for selection sort is as follows:

```
Procedure selection_sort(array, N)
  array - array of items to be sorted
  N - size of array
Begin
  For i = 1 to N-1
    Begin
      Set min = i
      For j = i+1 to N
        Begin
          If array[j] < array[min] Then
            min = j
          End If
        End For
      // Swap the minimum element with the current element
      If min != i Then
        Swap array[min] and array[i]
```

```
      End If
    End For
End Procedure
```

10.2.3 An Example

1. Let us see an example of how selection sort works on the following array:

   ```
   [64, 25, 12, 22, 11]
   ```

2. In the first iteration, the algorithm finds the smallest element (11) in the unsorted part of the array and swaps it with the first element of the unsorted part (64). The array becomes:

   ```
   [11, 25, 12, 22, 64]
   ```

3. In the second iteration, the algorithm finds the smallest element (12) in the remaining unsorted part of the array and swaps it with the second element of the unsorted part (25). The array becomes:

   ```
   [11, 12, 25, 22, 64]
   ```

4. In the third iteration, the algorithm finds the smallest element (22) in the remaining unsorted part of the array and swaps it with the third element of the unsorted part (25). The array becomes:

   ```
   [11, 12, 22, 25, 64]
   ```

5. In the fourth iteration, the algorithm finds the smallest element (25) in the remaining unsorted part of the array and swaps it with the fourth element of the unsorted part (25). The array becomes:

   ```
   [11, 12, 22, 25, 64]
   ```

6. In the fifth iteration, the algorithm finds the smallest element (64) in the remaining unsorted part of the array and swaps it with the fifth element of the unsorted part (64). The array becomes:

   ```
   [11, 12, 22, 25, 64]
   ```

7. The array is now sorted and the algorithm terminates.

10.2.4 Instructions

1. The template is a generic class that can sort any type of data that implements the `Comparable` interface. This means that the data type `T` must have a method `compareTo(T other)` that returns a negative, zero, or positive integer depending on whether the current object is less than, equal to, or greater than the other object.
2. The constructor of the class takes an array of type `T` as a parameter and assigns it to the data field. The data field is the array that needs to be sorted.
3. The `SelectionSort()` method is where the students need

to write the code for the selection sort algorithm. The method should return a sorted array of type `T`.

4. The students can use the pseudocode provided in the previous section as a guide for writing the code. They need to use a for loop to iterate over the unsorted part of the array, find the minimum element, and swap it with the first element of the unsorted part. They can use the `compareTo()` method to compare the elements of the array. They can use a temporary variable to store the value of the element to be swapped.
5. The students can test their code by creating an object of the `Main` class with different types of data, such as integers, strings, or custom objects, and calling the `SelectionSort()` method on it. They can print the original and sorted arrays to check the output. They can also use different sizes of arrays to see how the algorithm performs.

10.3 Insertion Sort

10.3.1 Introduction

Insertion sort is a simple and adaptive sorting technique that works by inserting each element into its correct position in a sorted subarray. The sorted subarray starts with the first element and grows by one element in each iteration. The element to be inserted is compared with the previous elements in the sorted subarray and swapped if they are larger. This process continues until the element finds its correct position or reaches the beginning of the array.

10.3.2 Pseudocode

The pseudocode for insertion sort is as follows:

```
procedure insertionSort (A: list of sortable items)
    n = length (A)
    for i = 1 to n - 1 do
        j = i
        while j > 0 and A [j-1] > A [j] do
            swap (A [j], A [j-1])
            j = j - 1
        end while
    end for
end procedure
```

10.3.3 An Example

Consider the array: [3, 5, 7, 11, 13, 2, 9, 6]

Below is a detailed walkthrough of each iteration:

1. **Iteration 1**: `[3, 5, 7, 11, 13, 2, 9, 6]`
   The sorted subarray is [3] and the element to be inserted is 5. Since 5 is larger than 3, no swap is needed.
2. **Iteration 2**: `[3, 5, 7, 11, 13, 2, 9, 6]`
   The sorted subarray is [3, 5] and the element to be inserted is 7. Since 7 is larger than 5, no swap is needed.
3. **Iteration 3**: `[3, 5, 7, 11, 13, 2, 9, 6]`
   The sorted subarray is [3, 5, 7] and the element to be inserted is 11. Since 11 is larger than 7, no swap is needed.
4. **Iteration 4**: `[3, 5, 7, 11, 13, 2, 9, 6]`
   The sorted subarray is [3, 5, 7, 11] and the element to be inserted is 13. Since 13 is larger than 11, no swap is needed.
5. **Iteration 5**: `[3, 5, 7, 11, 13, 2, 9, 6]`
   The sorted subarray is [3, 5, 7, 11, 13] and the element to be inserted is 2. Since 2 is smaller than 13, 13 and 2 are swapped.

   After swapping: `[3, 5, 7, 11, 2, 13, 9, 6]`
6. **Iteration 6**: `[3, 5, 7, 11, 2, 13, 9, 6]`
   The sorted subarray is [3, 5, 7, 11, 2, 13] and the element to be inserted is 2. Since 2 is smaller than 11, 11 and 2 are swapped.

   After swapping: `[3, 5, 7, 2, 11, 13, 9, 6]`
7. **Iteration 7**: ‘[3, 5, 7,

2, 11, 13, 9, 6]‘
The sorted subarray is [3, 5, 7, 2, 11, 13] and the element to be inserted is 2. Since 2 is smaller than 7, 7 and 2 are swapped.

After swapping: `[3, 5, 2, 7, 11, 13, 9, 6]`

8. **Iteration 8**: `[3, 5, 2, 7, 11, 13, 9, 6]`
   The sorted subarray is [3, 5, 2, 7, 11, 13] and the element to be inserted is 2. Since 2 is smaller than 5, 5 and 2 are swapped.

   After swapping: `[3, 2, 5, 7, 11, 13, 9, 6]`

9. **Iteration 9**: `[3, 2, 5, 7, 11, 13, 9, 6]`
   The sorted subarray is [3, 2, 5, 7, 11, 13] and the element to be inserted is 2. Since 2 is smaller than 3, 3 and 2 are swapped.

   After swapping: `[2, 3, 5, 7, 11, 13, 9, 6]`

10. **Iteration 10**: `[2, 3, 5, 7, 11, 13, 9, 6]`
    The sorted subarray is [2, 3, 5, 7, 11, 13] and the element to be inserted is 9. Since 9 is smaller than 13, 13 and 9 are swapped.

    After swapping: `[2, 3, 5, 7, 11, 9, 13, 6]`

11. **Iteration 11**: `[2, 3, 5, 7, 11, 9, 13, 6]`
    The sorted subarray is [2, 3, 5, 7, 11, 9, 13] and the element to be inserted is 9. Since 9 is smaller than 11, 11 and 9 are swapped.

    After swapping: `[2, 3, 5, 7, 9, 11, 13, 6]`

12. **Iteration 12**: `[2, 3, 5, 7, 9, 11, 13, 6]`
    The sorted subarray is [2, 3, 5, 7, 9, 11, 13] and the element to be inserted is 9. Since 9 is larger than 7, no swap is needed.

13. **Iteration 13**: `[2, 3, 5, 7, 9, 11, 13, 6]`
    The sorted subarray is [2, 3, 5, 7, 9, 11, 13] and the element to be inserted is 6. Since 6 is smaller than 13, 13 and 6 are swapped.

    After swapping: `[2, 3, 5, 7, 9, 11, 6, 13]`

14. **Iteration

14**: `[2, 3, 5, 7, 9, 11, 6, 13]`
The sorted subarray is [2, 3, 5, 7, 9, 11, 6, 13] and the element to be inserted is 6. Since 6 is smaller than 11, 11 and 6 are swapped.

`*After swapping*: ``[2, 3, 5, 7, 9, 6, 11, 13]``

15. **Iteration 15**: `[2, 3, 5, 7, 9, 6, 11, 13]`
    The sorted subarray is [2, 3, 5, 7, 9, 6, 11, 13] and the element to be inserted is 6. Since 6 is smaller than 9, 9 and 6 are swapped.

    After swapping: `[2, 3, 5, 7, 6, 9, 11, 13]`

16. **Iteration 16**: `[2, 3, 5, 7, 6, 9, 11, 13]`
    The sorted subarray is [2, 3, 5, 7, 6, 9, 11, 13] and the element to be inserted is 6. Since 6 is smaller than 7, 7 and 6 are swapped.

    After swapping: `[2, 3, 5, 6, 7, 9, 11, 13]`

17. **Iteration 17**: `[2, 3, 5, 6, 7, 9, 11, 13]`
    The sorted subarray is [2, 3, 5, 6, 7, 9, 11, 13] and the element to be inserted is 6. Since 6 is larger than 5, no swap is needed.

Final array: `[2, 3, 5, 6, 7, 9, 11, 13]`
The sorted subarray is [2, 3, 5, 6, 7, 9, 11, 13] and the array is fully sorted.

10.3.4 Instructions

1. Start by understanding how insertion sort works and what it does to sort an array.
2. Study the given template for the Main class and understand how it uses Java generics to sort an array of any type that implements the Comparable interface.
3. Implement the `InsertionSort` method by following the pseudocode provided above. Students should create a copy of the input array and modify the copy in-place to produce the sorted array.
4. Test their implementation using a variety of input arrays and verify that the output is correct.
5. Encourage students to optimize their implementation by considering edge cases and identifying areas for improvement.
6. Remind students to comment their code and explain their thought process as they work through the assignment. This will help them develop their coding skills and become better problem solvers.

11 Linear Search

Linear search is one of the simplest searching algorithms. The linear search algorithm scans one item at a time, without jumping to any item. It starts at the beginning of a list (or array) and sequentially checks each element until it finds a match.

Here's a general breakdown of how the linear search algorithm works:

1. It starts at the first item in the list.
2. It moves from item to item, checking if the current item is equal to the item being searched.
3. If a match is found, the algorithm stops and returns the index of the item.
4. If the algorithm reaches the end of the list without finding a match, it returns a signal that no match was found (often this is -1 or null).

Linear search can be applied to any type of data. However, since it searches element-by-element, it can be inefficient for large lists or arrays.

11.1 Pseudocode

The pseudocode for linear search is straightforward:

```
procedure LinearSearch(A, key)
    for i from 1 to length(A)
        if A[i] equals key
            return i
    return -1
```

This pseudocode represents a function called `LinearSearch` that takes two parameters: a list (or array) `A` and a `key` to search for in `A`. The function returns the index of `key` if found, otherwise it returns `-1`.

11.2 An Example

Let's consider the following example: We have an array `[10, 15, 20, 25, 30, 35]` and we want to find the number 20.

1. **Iteration 1**: The first element is 10. 10 is not equal to 20, so we move on to the next element.
2. **Iteration 2**: The second element is 15. 15 is not equal to 20, so we move on to the next element.
3. **Iteration 3**: The third element is 20. 20 is equal to 20, so we stop searching. The index of the number 20 is 2 (considering the first index as 0).

Therefore, the linear search algorithm would return `2` as the index of the number 20 in the given array.

11.3 Instructions

Here are some instructions to help you start working on the assignment:

1. The template uses Java generics to create a generic class Main that can search for a key element in an array of any type that implements the Comparable interface. Generics are a way of implementing generic programming in Java, which allows you to write code that can work with different types of objects without casting or risking ClassCastException.
2. The constructor of the Main class takes an array of type T as a parameter and assigns it to the array field. The array field is also of type T, which means it can store any type of object that implements Comparable.
3. The `LinearSearch` method returns an int that is the index of the key element in the array, or -1 if the key element is not found. The method uses a for loop to iterate over the array and compare each element with the key using the `compareTo` method of the Comparable interface. The `compareTo` method returns a negative integer, zero, or a positive integer if the current object is less than, equal to, or greater than the specified object. For example, if you want to compare the element at index i with the key, you can write:

   ```
   if (array[i].compareTo(key) == 0) {
       // return the index
   }
   ```

   The `LinearSearch` method has a time complexity of O(n), where

n is the size of the array, because it may have to scan the entire array to find the key element.
4. To test your code, you can create an object of the Main class with different types of arrays, such as Integer, String, or Double, and call the `LinearSearch` method on them. You can print the key element and the index returned by the methods to check the output. For example, you can write:

```java
Integer[] intArray = {1, 2, 4, 5, 8};
Main<Integer> intMain = new Main<>(intArray);
Integer key = 4;
int linearIndex = intMain.LinearSearch(key);
System.out.println("Key element: " + key);
System.out.println("Linear search index: " + linearIndex);
```

The output of the above code should be:

```
Key element: 4
Linear search index: 2
```

Part V

Sequential Collections of Data

12 Collections of Data

12.1 (Active) Memory in Computers

To fully grasp the concept of data collections in Java, we must first lay a foundational understanding of memory in computing systems. It is helpful to visualize the computer's memory as a vast grid, with each cell in the grid being capable of storing a certain number of bits. This simple grid becomes the underpinning infrastructure for data storage, holding variables that serve a multitude of functions in our programming ventures.

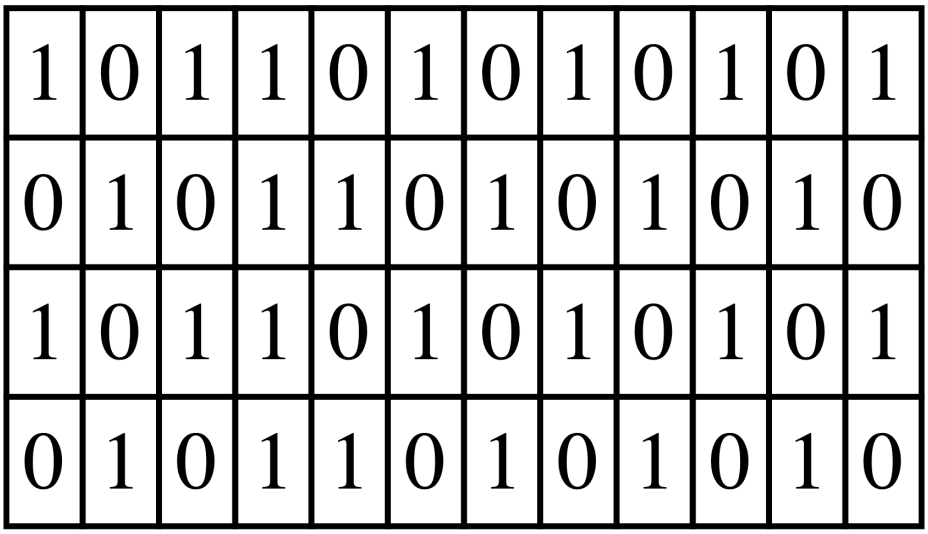


Figure 12.1: A diagram showing a grid of memory cells, each capable of storing a single bit.

12.2 The Concept of Data Collections

In programming, the need often arises to handle groups of similar variables. These groups are what we refer to as data collections. A data collection is essentially an assembly of bits in the

memory, representing a group of variables. For instance, you could have a collection of four 2-bit variables, which together would take up eight bits of memory. But the behavior of these collections can vary, particularly when considering size constraints.

12.3 Types of Collections: Fixed and Dynamic

When analyzing the size of data collections, we encounter two main types: fixed (or static) and dynamic collections.

Fixed or Static Collections: As the name suggests, in a static collection, the number of variables is constant. This implies that the collection's size is immutable, and the total memory occupancy remains consistent. An array of 10 integers is a perfect example of a static collection - regardless of the values stored in it, the array always contains ten integers.

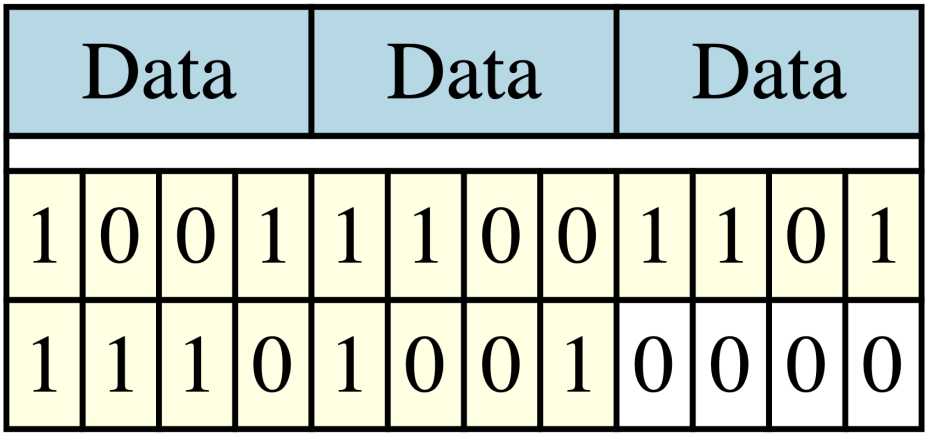


Figure 12.2: A diagram showing a static collection of data where the five 4-bit variables are placed sequentially in memory.

Dynamic Collections: Dynamic collections differ from their static counterparts in that their sizes are subject to change over time. Consider, for example, a collection storing the names of your favorite musicians. As your taste in music evolves, so does this list, reflecting an expansion or contraction based on your preferences. Thus, it is a dynamic collection.

12.4 Memory Layout of Collections

Let's take a deeper look into how these collections are stored in memory. Note that the memory allocation of these collections can be influenced by factors such as the variables' data type and size.

A common approach to store a collection of two 4-bit numbers is sequentially, positioning them back-to-back. However, this is not the only strategy. Depending on the system's memory management and data alignment methodologies, 'padding' could be introduced, which involves inserting extra bits between variables to align them properly in memory.

Regardless of these variations, a crucial consideration when working with collections is devising a mechanism to locate the next variable in the sequence. This task is straightforward for static collections due to the known size of each variable. But how about dynamic collections?

12.5 Tackling Dynamic Collections

Dynamic collections present an interesting challenge due to their mutable size. One way to navigate this issue is by earmarking a portion of the memory to store the length of the collection. For instance, the first four bits could be utilized to denote the size of the collection.

By adopting this method, we can efficiently locate subsequent data elements in our collection and discern the collection's end point. Bear in mind that the data in collections doesn't have to be stored adjacently. We can opt to pad each element with a fixed number of bits. As long as the padding size remains consistent, we can still locate the next piece of data.

12.6 Memory Layout and Program Performance

The manner in which data collections are stored in memory can have a significant impact on the performance of a program. For instance, accessing memory sequentially (in a pattern that matches the memory layout of the collection) allows the system to leverage cache lines, which are blocks of memory that are read into the CPU's cache. Due to the way modern CPUs are designed, once a part of the memory is read, an entire cache line (usually 64 or 128 bytes) is loaded into the cache. Thus, sequential access often results in fewer cache misses, which improves the performance of your program.

In contrast, random access to memory can lead to frequent cache misses, as each access may require loading a different cache line into the CPU cache, slowing down your program. Therefore, understanding the memory layout of your data collections and designing your program to access data in a manner that complements this layout can lead to significant performance improvements.

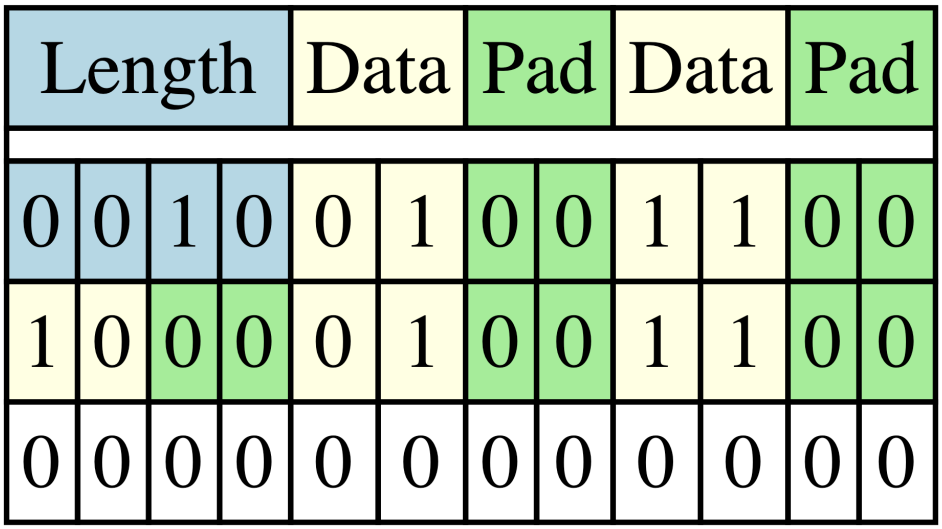


Figure 12.3: A diagram showing a dynamic collection of data where the first 4 bits represent the length of the collection, and the next few bits are the collection of variables. Each variable will be separated by 2-bit padding.

So, as you delve deeper into the world of data collections, remember that your approach to handling memory can either elevate or hinder your program's efficiency. The choice, as always, is in your hands!

12.7 Summary and Conclusion

This chapter provided a comprehensive overview of data collections in programming, focusing mainly on their storage in computer memory. It began by explaining the fundamental understanding of memory in computing systems, portraying it as a vast grid where each cell stores a certain number of bits. This concept helped set the groundwork for understanding how data collections, which are groups of similar variables, are stored in memory.

The chapter then discussed two main types of data collections: fixed (static) and dynamic collections. While fixed collections have a constant number of variables and thus a constant memory size, dynamic collections have a variable size that can change over time. An illustration was given of both types to aid understanding.

The memory layout of these collections was then delved into, noting that how these collections are stored in memory can be influenced by the variables' data type and size. The chapter also explained the strategies for accessing variables in both static and dynamic collections.

Moreover, the chapter emphasized the significant impact of the memory layout of data collections on program performance. Sequential memory access in line with the memory layout could leverage cache lines and enhance the program's performance, while random access could lead to frequent cache misses and slow down the program.

In conclusion, understanding data collections, how they are stored in memory, and how to access this data efficiently are critical aspects of programming. By considering these factors, programmers can significantly influence the performance of their programs. Thus, the mastery of data collections is not only essential for writing efficient code but is also a vital skill for optimizing the overall performance of a program.

13 Operations on Lists

In the prior lecture, we journeyed through the concept of collections of data and their manifestations in computer memory. Such collections were referred to as lists, a data structure central to understanding the organization and manipulation of data in programming. Our objective is to delve into the practical implementation of these lists in Java. However, before we embark on that, we must consider the types of operations we desire our lists to perform. These operations guide our implementation strategy and ensure that our list behaves in the ways we want it to. In this chapter, we'll be investigating operations performed on a List Abstract Data Type (ADT).

13.1 Adding Data to a List

As you may recall, a list is a collection of data elements. A key operation, therefore, is the ability to add data to this collection. Let's visualize a simple list:

1	2	3	4	5	6

There are multiple ways to insert new data into this list. For instance, you might want to add a new element, say "X", at the end:

1	2	3	4	5	6	X

Alternatively, you could place "X" at the beginning:

X	1	2	3	4	5	6

Or perhaps insert "X" somewhere in the middle:

1	2	3	X	4	5	6

There are three distinct methods to add data to our list, and each method may serve different needs in your program. Let's discuss how these operations are implemented in the List ADT provided by the Java standard library, `java.util.List`.

13.1.1 Appending an Element

Appending an element to the end of a list is perhaps the most straightforward method of adding data. The `add` method provided by `java.util.List` is used for this operation. Here is how you might use it:

```
listObject.add(itemToAdd);
```

13.1.2 Prepending an Element

Inserting an element at the beginning of a list is also a common operation, often called "prepending". Though the `java.util.List` interface doesn't provide a `prepend` method, we can use the overloaded `add` method, which accepts an index and an element. To prepend, we simply pass `0` as the index:

```
listObject.add(0, itemToAdd);
```

13.1.3 Inserting an Element at a Specific Index

As hinted in the prepend operation, the `add` method in `java.util.List` allows for adding an element at any index in the list:

```
listObject.add(index, itemToAdd);
```

13.1.4 Adding All Elements from Another List

Sometimes, you may want to merge one list into another. The `addAll` method allows you to add all the elements from another list to the end of the current list:

```
listObject.addAll(anotherList);
```

13.2 The Role of Abstract Data Types (ADTs)

We've mentioned the term "Abstract Data Type" (ADT) several times so far, but what does it really mean? An ADT is a high-level description of a collection of data and the operations that can be performed on that data. It is "abstract" in that it describes what operations are to be done but not how these operations will be implemented.

In the context of lists,

the List ADT defines a list's behavior. For instance, we can append, prepend, or insert an element at a specific index, but we don't care about how these operations are performed.

Why do we need ADTs? ADTs allow us to abstract away the details of the data structure's implementation. They encapsulate the data and provide a well-defined interface to interact with it, ensuring that data remains consistent and operations on the data are predictable. Moreover, by using ADTs, different implementations of the same type of data structure can be swapped seamlessly, without changing the code that uses the data structure.

In your journey with Java so far, you've only used one kind of list – the ArrayList. As you progress, you'll encounter other types of lists, like LinkedLists and Stacks, each with its own strengths, weaknesses, and suitable use cases. All these list types adhere to the same List ADT, allowing us to switch from one type to another depending on our requirements, while our code remains largely the same.

In the next sections of this course, we will dive deeper into different types of lists and their unique characteristics. For now, the important takeaway is that ADTs allow us to focus on *what* operations we want to perform without worrying about *how* they are implemented.

13.2.1 Summary of Addition Operations

Here is a summary of the adding operations available in the `java.util.List` ADT:

Modifier and Type	Method and Description
boolean	add(E e) Appends the specified element to the end of this list (optional operation).
void	add(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
boolean	addAll(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).

Modifier and Type	Method and Description
boolean	addAll(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).

13.3 Removing data from our List

13.4 Removing Data from a List

Having explored the addition of data to a list, let's now shift our attention to its counterpart: removing data from the list. Remember our list from the previous section?

1	2	3	4	5	6

Let's consider different ways to remove data from this list.

13.4.1 Removing an Element at a Specific Index

Sometimes, you might need to remove an item from a specific position within the list. Java's List ADT provides the `remove(int index)` method for this purpose. This method removes the element at the specified position:

```
listObject.remove(index);
```

13.4.2 Removing the First Occurrence of an Element

At times, you might not know (or care about) the index of the element you want to remove, but you know the value of the element. In such cases, you can use the `remove(Object o)` method, which removes the first occurrence of the specified element from the list:

```
listObject.remove(objectToRemove);
```

13.4.3 Removing All Elements from Another List

Consider that you have two lists, and you want to remove all elements in the second list from the first one. You can use the `removeAll(Collection<?> c)` method, which removes from the current list all of its elements that are contained in the specified collection:

```
listObject.removeAll(anotherList);
```

13.4.4 Removing All Elements from the List

In some scenarios, you might want to clear your list entirely. The `clear()` method comes in handy for this, as it removes all elements from the list:

```
listObject.clear();
```

Like the addition operations, these removal operations offer different ways to manage the data in our list, providing flexibility based on the specific needs of our program.

13.4.5 Summary of Removal Operations

Here is a summary of the removal operations available in the `java.util.List` ADT:

Modifier and Type	Method and Description
E	remove(int index) Removes the element at the specified position in this list (optional operation).
boolean	remove(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
boolean	removeAll(Collection<?> c) Removes from this list all of its elements that are contained in the specified collection (optional operation).
void	clear() Removes all of the elements from this list (optional operation).

13.5 Searching for data in our List

13.6 Searching in a List

Beyond just adding and removing data from our list, we often need to find or check for the existence of certain elements in our list. Java's List ADT provides several methods for such search operations. Let's dive into these operations using our familiar list:

1	2	3	4	5	6

13.6.1 Checking If an Element Exists in the List

Sometimes, all we need to know is whether a certain element exists in our list. The `contains(Object o)` method serves this purpose, returning `true` if this list contains the specified element, and `false` otherwise:

```
boolean contains = listObject.contains(objectToCheck);
```

13.6.2 Checking If All Elements of Another Collection Exist in the List

If you have a collection of elements and you want to check whether all these elements exist in your list, you can use the `containsAll(Collection<?> c)` method. It returns `true` if the list contains all of the elements of the specified collection:

```
boolean containsAll = listObject.containsAll(anotherCollection);
```

13.6.3 Finding the Index of an Element

If you want to find the position of a certain element in your list, Java provides the `indexOf(Object o)` method. This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element:

```
int index = listObject.indexOf(objectToFind);
```

13.6.4 Finding the Last Index of an Element

In a list with duplicate elements, you might be interested in finding the last occurrence of an element. In this case, you can use the `lastIndexOf(Object o)` method, which returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element:

```
int lastIndex = listObject.lastIndexOf(objectToFind);
```

13.6.5 Summary of Search Operations

Here is a summary of the search operations available in the `java.util.List` ADT:

Modifier and Type	Method and Description
boolean	contains(Object o) Returns true if this list contains the specified element.
boolean	containsAll(Collection<?> c) Returns true if this list contains all of the elements of the specified collection.
int	indexOf(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
int	lastIndexOf(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

Search operations are a fundamental part of manipulating lists. They allow us to locate and verify the presence of data, enhancing our ability to interact with our list and derive meaningful results.

13.7 Miscellaneous Operations on a List

In addition to adding, removing, and searching elements in our list, Java's List ADT provides a variety of other useful operations. These allow us to manipulate and inquire our list in a more sophisticated manner.

13.7.1 Accessing an Element

Sometimes, we need to retrieve the element at a specific position in our list without removing it. The `get(int index)` method provides this functionality, returning the element at the specified position:

```
E element = listObject.get(index);
```

13.7.2 Modifying an Element

What if we need to change an element at a specific position? Java provides the `set(int index, E element)` method. This method replaces the element at the specified position in this list with the specified element:

```
listObject.set(index, newElement);
```

13.7.3 Determining the Size of the List

When working with lists, it's often necessary to know the number of elements present. The `size()` method returns the number of elements in this list:

```
int size = listObject.size();
```

13.7.4 Converting the List to an Array

On occasion, we may need to convert our list into an array. The `toArray()` method fulfills this purpose, returning an array containing all the elements in this list in proper sequence:

```
Object[] array = listObject.toArray();
```

13.7.5 Summary of Miscellaneous Operations

Here is a summary of the miscellaneous operations available in the `java.util.List` ADT:

Modifier and Type	Method and Description
E	get(int index) Returns the element at the specified position in this list.

Modifier and Type	Method and Description
E	set(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	size() Returns the number of elements in this list.
Object[]	toArray() Returns an array containing all of the elements in this list in proper sequence (from first to last element).

These miscellaneous operations on a list help us to manipulate and utilize our lists effectively. In the subsequent sections, we will delve deeper into different types of lists and how these operations can be applied differently. Stay tuned!

13.8 Summary and Conclusion

This chapter aimed to offer a comprehensive understanding of operations that can be performed on lists using the Java programming language. It introduced key manipulations including adding, removing, and searching for elements in a list, as well as several other useful list operations.

We began by delving into the process of adding elements to a list, where we considered various insertion methods such as appending, prepending, and inserting at a specific index. We also explored the ability to add all elements from another list into the current list. These addition operations provide flexibility and adaptability in managing data in our list as required by the program.

Next, we moved on to the concept of removing elements from a list. We investigated removing elements at specific indices, removing the first occurrence of an element, and removing all elements from a list or all elements present in another list. These removal operations are just as vital in data management, providing diverse ways to regulate the contents of our list.

Subsequently, we analyzed search operations within a list. These operations are crucial in data retrieval, determining the existence and position of data elements in our list. Methods like `contains(Object o)`, `containsAll(Collection<?> c)`, `indexOf(Object o)`, and `lastIndexOf(Object o)` were discussed in depth.

Furthermore, we examined miscellaneous operations that provide additional functionality in our list. These include accessing and modifying elements at specific indices, determining the size of the list, and converting the list into an array.

Throughout our exploration, we highlighted the significant role of Abstract Data Types (ADTs). By standardizing the operations that can be performed on a list, the List ADT allows us to focus on what we want to accomplish without concerning ourselves with the underlying implementation details. This abstraction brings forth code consistency, reliability, and the potential for seamless integration of different list implementations.

In conclusion, understanding and utilizing list operations in Java is fundamental for effective data management in programming. The breadth of operations available affords us immense flexibility in manipulating lists to align with our program's requirements. As we proceed in this course, we will delve deeper into the different types of lists and their unique characteristics, all the while, leveraging the power and versatility of the List ADT.

14 Implementing Array-backed Lists

15 Implementing Linked Lists

In this chapter, we'll take a closer look at how objects interact in Java. To fully understand this interaction, we first need to grasp the concept of references in Java, memory allocation for Java objects, and how different objects can interact through references. Later, we'll also examine how these principles apply to implementing linked lists in Java.

15.1 Understanding References in Java

In Java programming, the statement that "all objects are references" is a cornerstone. Understanding what this statement means is a fundamental step in mastering Java.

In Java, when we say an object is a *reference*, it means that the object acts as an address pointer, referencing a specific location in memory. This location is where the actual data associated with that object is stored.

Consider your computer's memory as a large grid, as illustrated in Table 6.1. Each cell can store some data.

Table 15.1: Representation of computer's memory

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

For any object in Java, it's not the data of the object that is directly stored in the variable, but rather the *reference* to the data.

15.2 Memory Allocation for Java Objects

Consider the following example, in which we have a `Dog` class.

```
public class Dog {
    String name;
    int age;
}
```

Initially, the `Dog` class doesn't occupy any memory space. It simply serves as a blueprint for creating `Dog` objects.

Now, let's create a new instance of `Dog` and assign it to the variable `myDog`:

```
Dog myDog = new Dog();
```

This operation performs two main actions:

1. Memory is allocated for a new `Dog` object.
2. The address of that memory is stored in the `myDog` variable.

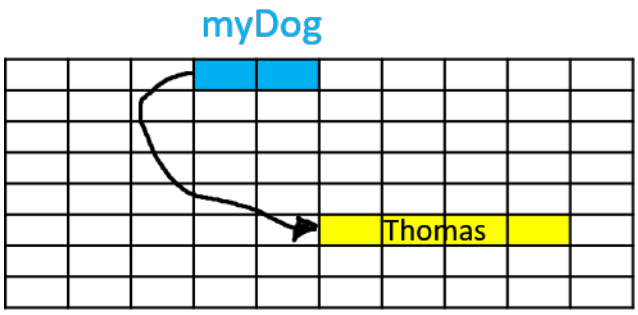


Figure 15.1: Dog Object Memory Allocation

So, `myDog` is essentially a pointer that points to the location in memory where the `Dog` object's data is stored.

15.3 6.3 Objects Interacting Through References

Now, let's consider a scenario where we have two classes, `Dog` and `HomeOwner`. The `Dog` class is the same as before, but the `HomeOwner` class is a bit more complex.

```
public class HomeOwner

 {
    String name;
    Dog pet;
}
```

Here, the `HomeOwner` class has a `pet` field of type `Dog`. This means a `HomeOwner` can own a `Dog`.

Now, suppose we have a constructor in the `Dog` class that accepts `name` and `age` as parameters, and a default constructor in the `HomeOwner` class:

```
public class Dog {
    String name;
    int age;

    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class HomeOwner {
    String name;
    Dog pet;

    public HomeOwner() {
        this.name = "Timothy";
        this.pet = new Dog("Timothy Jr.", 5);
    }
}
```

When we create an instance of `HomeOwner`:

```
HomeOwner owner = new HomeOwner();
```

A new `Dog` object is created and assigned to the `pet` field of the `HomeOwner` object. Importantly, the `pet` field stores the *reference* to the `Dog` object, not the actual `Dog` object data.

This creates a chain of references in memory, with the `owner` variable pointing to the `HomeOwner` object, which in turn has a reference to the `Dog` object.

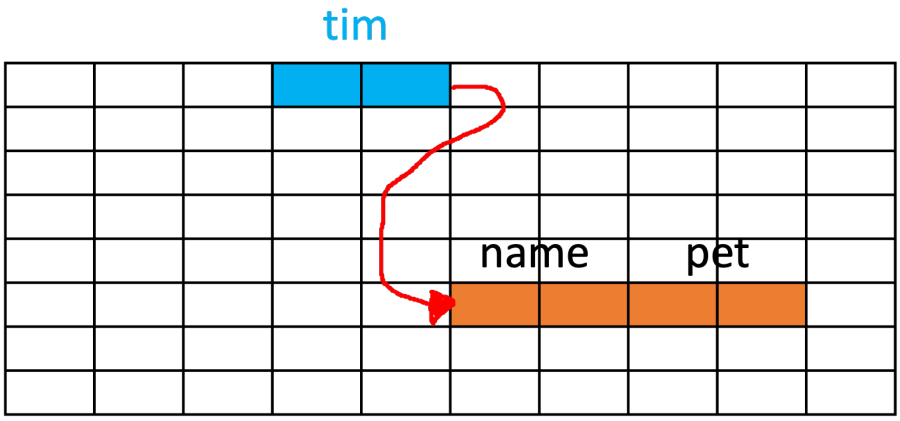


Figure 15.2: HomeOwner Object Creation

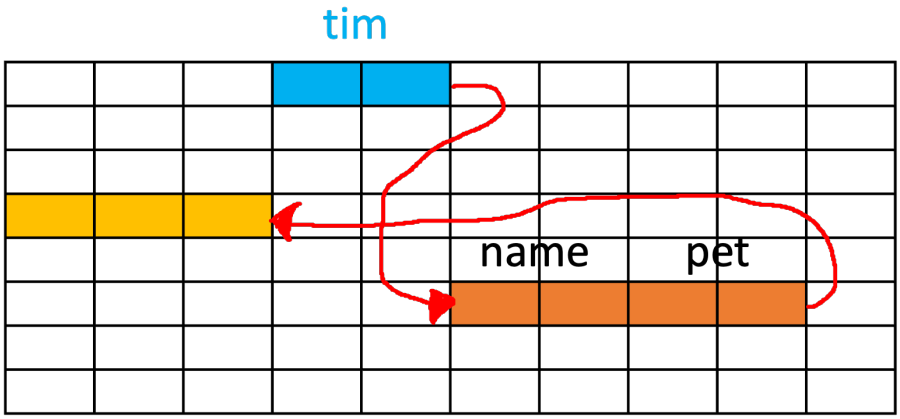


Figure 15.3: Dog Object Creation and Assignment

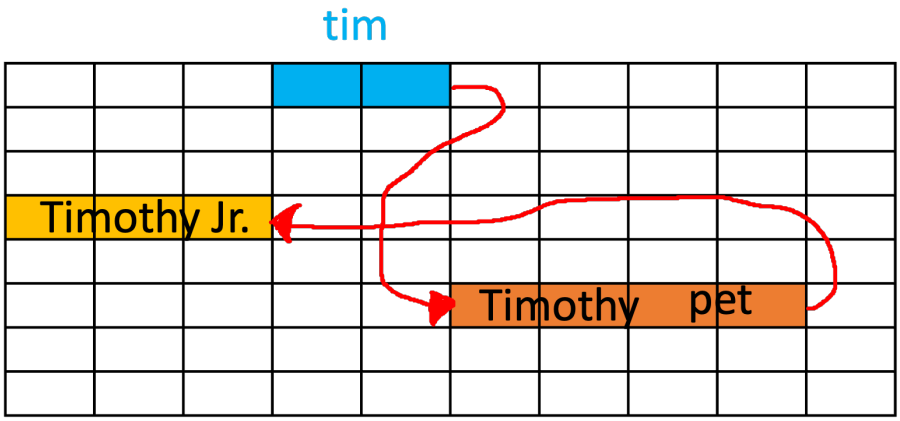


Figure 15.4: Chain of References

This understanding of references and memory allocation sets the foundation for how linked lists work in Java, which we will explore in the upcoming sections.

15.4 Implementing Linked Lists in Java

Absolutely! Last time, we used arrays to construct lists, but we can indeed build a list using the concept of references we have just discussed.

We'll start by creating a class where we'll store a reference to the first piece of data in our List. Then, our first piece of data will contain a reference to the next piece, and the next piece will point to the one following it, and so on.

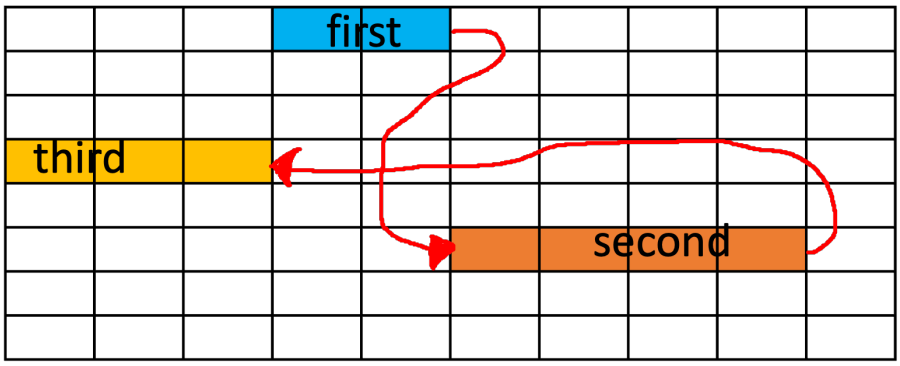


Figure 15.5: Linked List

This gives us the following structure:

```
class LinkedList {
  firstPieceOfData head;
}

class firstPieceOfData {
  nextPieceOfData next;
}
```

However, there's no actual data in this structure yet.

So, how about this?

```
class LinkedList {
  firstPieceOfData head;
}

class firstPieceOfData {
  nextPieceOfData next;
  actualData data;
}

class nextPieceOfData {
  nextPieceOfData next;
  actualData data;
}
```

To ensure our list is flexible and can store any type of data, we should make it generic:

```
class LinkedList<T> {
  firstPieceOfData<T> head;
}

class firstPieceOfData<T> {
  nextPieceOfData<T> next;
  T data;
}

class nextPieceOfData<T> {
  nextPieceOfData<T> next;
  T data;
}
```

Here, `<T>` is a type parameter that allows us to define a class with placeholders for the types

they use, which we can specify when we create an instance of the class.

We can also observe that `firstPieceOfData` and `nextPieceOfData` are structurally the same, as both classes store a reference to an object that holds data, and both hold some actual data. We can therefore simplify our structure by using a single class for both, like so:

```
class LinkedList<T> {
  Node<T> head;
}

class Node<T> {
  Node<T> next;
  T data;
}
```

The list we've created is known as a *singly linked list*. In some algorithms, we may want to access the previous node. In that case, we can store a reference to the previous node as well, creating what's known as a *doubly linked list*:

```
class LinkedList<T> {
  Node<T> head;
  Node<T> tail;
}

class Node<T> {
  Node<T> next;
  Node<T> prev;
  T data;
}
```

When we create a `Node` object, the constructor should initialize the `next` reference to `null` and `data` to the data we pass in. This is because at that point, we don't yet know what the next node will be, but we do know what data we want to store.

```
class SinglyLinkedNode<T> {
  private SinglyLinkedNode<T> next;
  private T data;

  public SinglyLinkedNode(T data) {
    this.data = data;
    this.next = null;
  }
}
```

We also renamed the `Node` class to `SinglyLinkedNode` to make it clear that this is for a singly-linked list.

The constructor for `SinglyLinkedList` will just set the head to null, initializing it as an empty list.

```
class SinglyLinkedList<T> {
  private SinglyLinkedNode<T> head;

  public SinglyLinkedList() {
    this.head = null;
  }
}
```

15.5 6.5 Adding to a Singly Linked List

The `SinglyLinkedNode`

class needs getter and setter methods for accessing and modifying its private data and next fields.

```
class SinglyLinkedNode<T> {
  private SinglyLinkedNode<T> next;
  private T data;

  public SinglyLinkedNode(T data) {
    this.data = data;
    this.next = null;
  }

  public SinglyLinkedNode<T> getNext() {
    return this.next;
  }

  public void setNext(SinglyLinkedNode<T> next) {
    this.next = next;
  }

  public T getData() {
    return this.data;
  }
```

```
  public void setData(T data) {
    this.data = data;
  }
}
```

With the getters and setters in place, we can now turn our attention to adding a new node to the list. Here's what we need to do when the list is empty:

```
public class SinglyLinkedList<T> {
  private SinglyLinkedNode<T> head;

  public SinglyLinkedList() {
    this.head = null;
  }

  public void add(T data) {
    head = new SinglyLinkedNode<T>(data);
  }
}
```

When we're adding elements or nodes to our singly linked list, we've seen that for the first addition, we simply set the head to a new node with the provided data. This is essentially the creation of the first element or node in our list.

Let's take a moment to visualize this scenario:

```
head --> [A]
```

In this ASCII diagram, the `-->` represents the link from the `head` to the first node `A`.

Now, when we're adding a second element, we must be careful. We can't simply overwrite the `head` again, as it contains our first piece of data. Instead, we want to add our new node at the next available position.

We achieve this by setting the `next` member of the head node to point to our new data, like so:

```
public void add(T data) {
  if (head == null) {
    head = new SinglyLinkedNode<T>(data);
  } else {
    head.setNext(new SinglyLinkedNode<T>(data));
  }
```

```
  }
```

The updated state of the list is now:

```
head --> [A] --> [B]
```

But what if we want to add a third node? Using the code above, we would inadvertently overwrite the link from `A` to `B`, and instead, link `A` directly to the new node.

We have to revise our strategy. What if we checked whether the `next` of the head was null and then added our new node there?

```
public void add(T data) {
  if (head == null) {
    head = new SinglyLinkedNode<T>(data);
  } else if (head.getNext() == null) {
    head.setNext(new SinglyLinkedNode<T>(data));
  } else {
    head.getNext().setNext(new SinglyLinkedNode<T>(data));
  }
}
```

This would result in:

```
head --> [A] --> [B] --> [C]
```

That seems to work, right? But this code has limitations; it only accommodates the addition of up to three nodes.

Let's observe what happens when we add a fourth node using this code. We'd end up with:

```
head --> [A] --> [B] --> [D]
```

The third node, `[C]`, disappears! That's not what we want. We need a way to add nodes to our list, regardless of its current length. Can you see a pattern forming in how we're adding nodes? How about a loop to iterate over the list until we find a node with a `null` `next` field?

```
public void add(T data) {
  if (head == null) {
    head = new SinglyLinkedNode<T>(data);
  } else {
```

```
    SinglyLinkedNode<T> current = head;

    while (current.getNext() != null) {
      current = current.getNext();
    }

    current.setNext(new SinglyLinkedNode<T>(data));
  }
}
```

Now, each time we call `add()`, we start at the head and follow the `next` references until we find a node where `next` is `null`. We then set that node's `next` to the new node, effectively adding it to the end of the list.

And there you have it! We've cracked the code for adding nodes to a singly linked list, regardless of its size.

15.6 Searching a Singly Linked List

Having explored the intricacies of adding elements to our singly linked list, it's only logical to now ask, "How can we find an element in our list?" After all, what good is storing data if we can't retrieve it?

Let's delve into the process of developing a `search` method that behaves similarly to the `indexOf` method in the Java standard library. Our `search` method should return the index of the first occurrence of a given element in the list and `-1` if the element is not found.

Before we get started, there's an important update we need to make to our generics. We'll be comparing objects in our list to the search target, which means we need to ensure these objects are comparable. To do this, we'll update `T` to `T extends Comparable<T>`. This ensures that the type `T` implements the `Comparable` interface, providing us with the ability to compare objects of this type.

```
class SinglyLinkedNode<T extends Comparable<T>> {
  private SinglyLinkedNode<T> next;
  private T data;

  //...
}

class SinglyLinkedList<T extends Comparable<T>> {
  private SinglyLinkedNode<T> head;
```

```
  //...
}
```

Alright, with this adjustment in place, let's get to developing our `search` method!

If you reflect on how we traverse our linked list, it may become clear that the same process can be utilized for searching. We start at the `head`, and we progress through the list via the `next` pointers until we either find our target or reach the end of the list.

Here's a potential implementation for the `search` method:

```
public int search(T target) {
  SinglyLinkedNode<T> current = head;
  int index = 0;

  while (current != null) {
    if (current.getData().compareTo(target) == 0) {
      return index;
    }
    index++;
    current = current.getNext();
  }

  return -1;  // Target not found
}
```

Let's examine this piece by piece. We start by creating a reference to the `head` of our list and initializing an `index` variable at `0`. We then enter a `while` loop that will continue as long as `current` is not `null`, effectively iterating over the entire list.

Inside the loop, we compare the data of the `current` node to our `target` using the `compareTo` method, which is available to us thanks to our `Comparable` constraint. If the data matches our target (`compareTo` returns `0`), we've found our target and we return the current index.

If the data does not match our target, we increment our `index` and move to the next node in the list. If we reach the end of the list without finding our target, the method returns `-1`, indicating the target is not in the list.

And that's how we develop a search method for our singly linked list, providing us with the means to locate and retrieve data efficiently!

15.7 Removing from a Singly Linked List

Just as we learned how to add elements to a singly linked list, let's now turn our attention to the process of removing elements. As with adding elements, we'll start with a simpler case and gradually address more complex scenarios. We'll design a `removeLast` method, which removes the last node from the list.

There are three scenarios we need to consider:

1. The list is empty.
2. The list contains only one element.
3. The list contains more than one element.

15.7.1 Case 1: Empty Linked List

The simplest scenario to handle is an empty list. If the list is empty, we have nothing to remove. Here's a simple starting point for our `removeLast` method:

```
public void removeLast() {
  if (head == null) {
    return; // Nothing to remove.
  }
}
```

This code handles an empty list by simply returning without doing anything.

15.7.2 Case 2: Linked List with One Element

Now, let's consider the case where our list contains only one element.

To remove the only element from the list, we would set `head` to `null`, effectively removing the link to that node.

Here's how our `removeLast` method looks now:

```
public void removeLast() {
  if (head == null) {
    return; // Nothing to remove.
  } else if (head.getNext() == null) {
    head = null; // Remove the only node in the list.
  }
}
```

15.7.3 Case 3: Linked List with More Than One Element

When the list contains more than one element, we must find the last node and the node before it. Why? Because to remove the last node, we must set the `next` field of the node before it to `null`.

Let's think about how we might achieve this. We can't simply set `head.getNext() = null` like we did with the one-element list, because this would leave us with only the first node.

Instead, we could use a loop similar to the one in the `add` method, but with a twist. Instead of stopping when `current.getNext() == null`, which would leave us at the last node, we stop when `current.getNext().getNext() == null`, which will leave us at the second-to-last node.

```
public void removeLast() {
  if (head == null) {
    return; // Nothing to remove.
  } else if (head.getNext() == null) {
    head = null; // Remove the only node in the list.
  } else {
    SinglyLinkedNode<T> current = head;
    while (current.getNext().getNext() != null) {
      current = current.getNext();
    }
    current.setNext(null); // Remove the last node from the list.
  }
}
```

Now, we have a `removeLast` method that handles any size list!

15.7.4 Moving to `remove(int i)`

Removing the last node from a list is a good starting point, but what if we want to remove an arbitrary node at index `i`?

For this, we would need to iterate over the nodes until we reach the `(i-1)`th node (just before the node we want to remove), then update its `next` field to skip over the `i`th node and link to the `(i+1)`th node.

However, this requires careful checking of edge cases, such as when `i` is `0` (requiring us to update the `head`), or when `i` is greater than the size of the list.

This will be our next challenge to tackle! Let's see how we can go about creating a `remove(int i)` method, which will remove a node at a given index from our singly linked list.

As previously mentioned, our approach will be to traverse the list until we reach the `(i-1)`th node, and then adjust its `next` field to skip the `i`th node, effectively removing it from the list.

Here's a basic version of this method:

```
public void remove(int i) {
  if (i == 0) {
    head = head.getNext(); // If the node to remove is the head, move the head to the next
  } else {
    SinglyLinkedNode<T> current = head;
    for (int j = 0; j < i - 1; j++) {
      current = current.getNext();
    }
    current.setNext(current.getNext().getNext());
  }
}
```

This is a simple implementation, but it lacks protection against edge cases. For instance, what happens if `i` is negative or if it's larger than the size of the list?

We can address these cases by adding a few checks to our method:

```
public void remove(int i) {
  if (i < 0) {
    return; // Do nothing for negative index.
  } else if (i == 0) {
    head = head.getNext(); // If the node to remove is the head, move the head to the next
  } else {
    SinglyLinkedNode<T> current = head;
    for (int j = 0; j < i - 1; j++) {
      if (current.getNext() == null) {
        return; // If the index is out of range, do nothing.
      }
      current = current.getNext();
    }
    if (current.getNext() != null) {
      current.setNext(current.getNext().getNext());
    }
  }
}
```

This updated method takes care of any negative index or index that's out of range by simply returning without making any changes. If the index is zero, we update the head of the list to

be the next node. If the index is within the range of the list, we find the node at `(i-1)` and update its `next` reference to skip over the `i`th node.

And with that, we've covered the basics of how to add and remove nodes from a singly linked list!

15.7.5 Summary and Conclusion

In this chapter, we learned how to implement linked lists in Java using references and objects. We saw how references can point to other objects in memory and how we can use them to create nodes that store data and link to other nodes. We also learned how to define a class for a singly linked list that contains a reference to the first node, called the head, and methods for adding, searching, and removing elements. We compared the advantages and disadvantages of linked lists with array-backed lists and discussed some applications of linked lists in real-world problems. We concluded that linked lists are a dynamic and flexible data structure that can grow and shrink as needed, but they also have some drawbacks such as extra memory overhead, lack of random access, and potential memory leaks.

Part VI

Stacks and Queues

16 Stack Data Structure

16.1 Introduction

A stack is a linear data structure that follows a **last-in-first-out (LIFO)** principle. That means the last element added to a stack is the first one to be removed. A stack has only one end, called the **top**, where elements can be inserted or deleted.

You can think of a stack as a pile of books. You can only add or remove books from the top of the pile. The book that you added last will be on top, and you have to remove it first before you can access any other book below it.

Stacks are useful for many applications that require reversing, backtracking, or undoing operations. For example, when you use an undo button in a text editor, you are using a stack to store your previous actions and revert them in reverse order.

16.2 Operations on a Stack

The two basic operations of a stack are **push** and **pop**. Push adds a new element to the top of the stack, while pop removes an element from the top of the stack. Both operations take constant time, that is **O(1)**, because they only involve changing one pointer.

Another operation that can be useful for stacks is **peek**. Peek returns the top element of the stack without removing it. This can be helpful for checking what is on top of the stack before performing any other operation. Peek also takes constant time, **O(1)**, because it only accesses one element.

Here is an example of how these operations work on a stack:

- Initially, the stack is empty: `[]`
- Push 10: `[10]`
- Push 20: `[10, 20]`
- Push 30: `[10, 20, 30]`
- Pop: returns 30 and removes it from the stack: `[10, 20]`
- Peek: returns 20 but does not remove it from the stack: `[10, 20]`
- Pop: returns 20 and removes it from the stack: `[10]`
- Pop: returns 10 and removes it from the stack: `[]`

- Pop: returns an error or null because there is nothing to pop

16.3 Applications of a Stack

Stacks have many applications in various domains, such as expression evaluation, backtracking, function calls, undo/redo operations, browser history, etc. In this section, we will focus on one application: expression evaluation.

Expression evaluation is the process of computing the value of an arithmetic or logical expression. For example, given an expression like `2 + 3 4 - 5 / (6 + 7)`, we want to find its value according to some rules of precedence and associativity.

So, for the problem of "Expression evaluation", the input is a **string** like `"2 + 3 * 4 - 5 / (6 + 7)"`, and the output is the **value** of the expression as an integer or a floating point number. This is much harder to do than it initially appears to be, particularly when we consider the rules of precedence and associativity.

To solve this problem, we can first convert an expression from infix notation (where operators are between operands) to postfix notation (where operators are after operands) using a stack.

It is useful to convert infix expressions to postfix expressions because postfix expressions are easier for computers to evaluate. Postfix expressions do not need parentheses or precedence rules to determine the order of operations. They can be evaluated from left to right using a stack.

- Infix: `2 + 3 * 4` (=> `2 + 12` => `14`)
- Postfix: `2 3 4 * +` (=> `2 12 +` => `14`)
- Infix: `2 + 3 * 4 - 5 / (6 + 7)`
- Postfix: `2 3 4 * + 5 6 7 + / -`

For example, to evaluate a postfix expression like `2 3 4 * +`, we can use a stack as follows:

1. Scan the expression from left to right
2. If we encounter an operand (a number), we push it onto the stack
3. If we encounter an operator (+, -, *, /), we pop two operands from the stack, apply the operator on them, and push the result back onto the stack
4. At the end of the expression, there will be only one value on the stack, which is the final result.

Here are the steps in action -

1. Scan the expression from left to right - `2 3 4 * +`
2. Encounter 2, push it onto the stack (stack: [2])
3. Encounter 3, push it onto the stack (stack: [2, 3])
4. Encounter 4, push it onto the stack (stack: [2, 3, 4])

5. Encounter , pop two operands (4 and 3) from the stack and apply on them. Push the result (12) onto the stack.(stack: [2, 12])
6. Encounter +, pop two operands (12 and 2) from the stack and apply + on them. Push the result (14) onto the stack. (stack: [14])
7. Reach the end of the expression, pop the final result (14) from the stack and output it. (stack: [])

As for how you convert an infix expression to a postfix expression, you can use stacks again! Here are the steps:

1. If we encounter an operand (a number), we append it to the output string.
2. If we encounter an operator (+, -, *, /), we push it onto the stack if it has higher precedence than the top of the stack. Otherwise, we pop all operators with equal or higher precedence than it from the stack and append them to the output string. Then we push it onto the stack.
3. If we encounter a left parenthesis '(', we push it onto the stack.
4. If we encounter a right parenthesis ')', we pop all operators from the stack until we reach a left parenthesis '(' and append them to the output string. Then we discard both parentheses.
5. If we reach the end of the input, pop all operators from the stack and append them to the output string.
6. The final output string is the postfix expression.

For example, to convert the infix expression `2 + 3 * 4 - 5 / (6 + 7)` to postfix, we can use a stack as follows:

1. Scan the expression from left to right - 2 + 3 * 4 - 5 / (6 + 7)
2. Encounter 2, append it to the output string `(output: 2)`
3. Encounter +, push it onto the stack `(stack: [+])`
4. Encounter 3, append it to the output string `(output: 2 3)`
5. Encounter *, push it onto the stack `(stack: [+, *])`
6. Encounter 4, append it to the output string `(output: 2 3 4)`
7. Encounter -, push it onto the stack `(stack: [+, *, -])`
8. Encounter 5, append it to the output string `(output: 2 3 4 5)`
9. Encounter /, pop all operators with higher or equal precedence from the stack and append them to the output string. Then push / onto the stack `(output: 2 3 4 5 * /, stack: [+, -])`
10. Encounter (, push it onto the stack `(stack: [+, -, (])`
11. Encounter 6, append it to the output string `(output: 2 3 4 5 * / 6)`
12. Encounter +, push it onto the stack `(stack: [+, -, (, +])`
13. Encounter 7, append it to the output string `(output: 2 3 4 5 * / 6 7)`
14. Encounter), pop all operators from the stack until we reach the matching left parenthesis and append them to the output string. Discard both parentheses `(output: 2 3 4 5 * / 6 7 + -, stack: [+])`

15. Pop the remaining operator (+) from the stack and append it to the output string `(output: 2 3 4 5 */ 6 7 + - +)`

The final output string is `2 3 4* + 5 6 7 + / -`, which is the postfix notation of the original infix expression.

17 Queue Data Structure

17.1 Introduction

A queue is a linear data structure that is open at both ends and follows a particular order for storing data. The order is **First In First Out** (FIFO). That means that the first element that is added to the queue is also the first one that is removed from it3. One can imagine a queue as a line of people waiting to receive something in sequential order which starts from the beginning of the line.

Some real-life examples of queues are:

- Waiting lines at a bank, a restaurant, or an airport
- Printer jobs that are processed in the order they are received
- Customer service calls that are answered based on who called first

Queues are useful for modeling situations where we need to process data elements one by one in the order they arrive.

17.2 Operations on a Queue

The main operations that we can perform on a queue are:

- **Enqueue**: This operation adds an element to the rear end of the queue. We can only enqueue an element if the queue is not full.
- **Dequeue**: This operation removes an element from the front end of the queue. We can only dequeue an element if the queue is not empty. The dequeued element is returned as the output of this operation.
- **Peek** or **Front**: This operation returns (but does not remove) the element at the front end of the queue without modifying it. We can only peek at an element if the queue is not empty.
- **Poll**: This operation is similar to dequeue, but it returns null instead of throwing an exception if the queue is empty.

Note that the exact operations, their names and behavior often vary depending on the programming language and the implementation of the queue. You always need to check the documentation of the programming language you are using to know the exact operations and their behavior.

17.3 Applications of Queues

A queue data structure is generally used in scenarios where we need to follow a **First In First Out (FIFO)** approach for managing data elements. That means that we process data elements one by one in the order they arrive or are added to the queue.

Some common applications of the queue data structure are:

- **Task Scheduling**: Queues can be used to schedule tasks based on priority or the order in which they were received. For example, a printer may use a queue to store print jobs that are waiting to be executed. The printer will print the documents in the order they were added to the queue, ensuring fairness and efficiency.
- **Resource Allocation**: Queues can be used to manage and allocate resources, such as printers, CPU processing time, disk space, etc. For example, an operating system may use a queue to handle requests for CPU time from different processes. The operating system will grant CPU time to each process based on its position in the queue, ensuring that no process is starved or neglected.
- **Data Buffering**: Queues can be used to buffer data between two components that operate at different speeds or have different capacities. For example, a keyboard may use a queue to store keystrokes that are entered by the user but not yet processed by the computer. The keyboard will enqueue each keystroke as it is typed, and the computer will dequeue each keystroke as it is ready to process it.

One application of queue data structure that we will focus on is **CPU Scheduling**.

CPU scheduling is the problem of deciding which process should get access to the CPU at any given time. CPU scheduling is important for optimizing CPU utilization, throughput, response time, waiting time, etc. CPU scheduling algorithms use queues to store processes that are ready or waiting for execution.

Different types of queues can be used for CPU scheduling:

- **Single Queue**: This is a simple FIFO queue that stores all processes that are ready for execution. The CPU will execute processes in the order they arrive or are added to this queue. This type of queue ensures fairness but does not consider priority or burst time (the amount of time required by a process for execution) of processes. An example of this type of queue is FCFS (First Come First Serve) scheduling algorithm.

- **Multiple Queues**: This is a set of FIFO queues that store processes based on some criteria such as priority or burst time. The CPU will execute processes from different queues according to some rules such as round robin (each queue gets an equal share of CPU time), shortest job first (the queue with the shortest burst time gets higher preference), etc. This type of queue allows more flexibility and control over CPU scheduling but may introduce complexity and overheads. An example of this type of queue is MLFQ (Multi-Level Feedback Queue) scheduling algorithm.
- **Priority Queue**: This is a special type of queue that stores processes based on their priority values. The priority value can be static (assigned at creation) or dynamic (changed during execution). The CPU will execute processes with higher priority values before lower ones. This type of queue ensures urgency but may cause starvation (low-priority processes never get executed) or inversion (high-priority process waits for the low-priority process) problems.

Part VII

Recursion

18 Recursion

Recursion is a computational technique in which a function refers to itself in its own definition to solve a problem. The essence of creating recursive functions is ensuring that the recursive call addresses a smaller version of the initial problem. This strategy enables the efficient and elegant decomposition of complex problems into more manageable sub-problems.

Let's consider some key characteristics of recursive functions:

- **Base cases** must exist in any recursive function. These are simple scenarios of the problem that can be solved directly without necessitating a recursive call. Base cases are vital to avoid infinite recursion, a situation where the function endlessly calls itself without termination.
- The **recursive case** comprises a call to the same function but on a smaller scale of the original problem. The arguments passed on during this recursive call should progressively approach the base case.
- Data flows through the recursive calls via function parameters and return values. Parameters help in propagating information forward through the recursive calls until reaching the base case. Conversely, return values relay information back from the base case.

Some Advantages of Recursion -

- Recursive solutions often embody elegance and simplicity that make them easier to understand than their iterative counterparts. The code tends to naturally mirror the structure of the problem.
- Some problems are inherently recursive, such as navigating tree data structures. Here, the tree is a common type of recursive data structure where each node links to smaller subtrees. We will learn more about trees in the next module, but this characteristic makes them ideally suited to recursive approaches.
- Recursive code is typically more concise than equivalent iterative code.

Some Disadvantages of Recursion -

- Recursive calls come with overheads, such as stacking parameters, return addresses, and local variables during each invocation. This additional resource consumption often makes iterative solutions more efficient.
- Recursive code can pose debugging and tracing challenges. Understanding the program flow often involves mentally "unraveling" each recursive call.

- Deeply recursive algorithms risk causing stack overflow exceptions if the recursion depth exceeds the stack's capacity.

Some Examples of Recursive Problems -

- Manipulation of recursive data structures like trees and linked lists. These structures contain smaller instances of themselves, making recursion a natural fit for processing them.
- Certain sorting algorithms, such as quicksort and merge sort, implement recursion.
- Mathematical problems, including the computation of factorials, Fibonacci numbers, and so on, often rely on recursive methods.
- Advanced computational paradigms like divide-and-conquer, backtracking, and dynamic programming frequently use recursion.

18.1 Writing Recursive Functions

Understanding and implementing recursive functions often necessitates thinking of the broader problem in terms of smaller, similar instances. If you can decompose the main problem into subproblems that echo the original, you're in a position to use recursion.

Follow these general steps when designing a recursive function:

1. **Identify the base case(s)**: Determine the simplest version of the problem that can be resolved directly without invoking recursion. This serves as the recursion's termination point.
2. **Reduce the problem**: Consider how the main problem can be diminished to a smaller but similar subproblem.
3. **Invoke the recursive call**: Initiate the recursive call on the smaller subproblem, passing any necessary parameters.
4. **Construct the solution**: Utilize the result of the recursive call to assemble the solution for the original problem.
5. **Return the result**: Once the solution is obtained, return the result.

Let's delve into examples of recursive functions that exhibit diverse structures:

18.1.1 Single Base Case, Single Recursive Case

This is the most straightforward recursive structure featuring a solitary base case and a single recursive call on a reduced subproblem.

```
// Calculate factorial of n
int factorial(int n) {
  // Base case
  if (n == 0) {
    return 1;
  }

  // Recursive call
  return n * factorial(n-1);
}
```

In the above code, the base case is when `n` is zero, and the recursive call is made on `n-1` (a smaller instance of the original problem).

18.1.2 Multiple Base Cases

Some problems might need multiple base cases to accommodate different elementary variants of the problem:

```
// Count ways to climb n steps
// Can climb 1 or 2 steps at a time
int countWays(int n) {

  // Base cases
  if (n == 1) {
    return 1;
  }
  if (n == 2) {
    return 2;
  }

  // Recursive calls
  return countWays(n-1) + countWays(n-2);

}
```

Here, there are two base cases (`n == 1` and `n == 2`), reflecting the two possible ways of ascending a short staircase.

18.1.3 Multiple Recursive Cases

More intricate problems might necessitate multiple recursive calls on diverse subproblems:

```
// Fibonacci number
int fib(int n) {
  // Base case
  if (n <= 1) {
    return n;
  }

  // Recursive calls
  return fib(n-1) + fib(n-2);
}
```

The Fibonacci sequence computation involves two recursive calls, each on a different subproblem (`n-1` and `n-2`).

18.2 Writing Advanced Recursive Functions

While some recursive functions feature only a single base case and a single recursive call, it's often the case that a function may have multiple base cases, multiple recursive cases, or a combination of both.

Here's the general template for such a recursive function in Java:

```
if ( base case 1 ) {
  // return some simple expression
}
else if ( base case 2 ) {
  // return some simple expression
}
...
else if ( recursive case 1 ) {
  // some work before
  // recursive call
  // some work after
 }
...
else { // last recursive case
  // some work before
```

```
  // recursive call
  // some work after
}
```

Each base case returns a simple expression, whereas each recursive case performs some work before and after the recursive call.

Let's now explore some specific examples.

18.2.1 Example 1: Prime Number Determination

Consider a function that checks whether a given integer `X` is a prime number. Here, `Y` is a helper variable acting as the divisor. When the function is initially invoked, `Y` is set to `X - 1`.

```
boolean prime(int x, int y) {
  if (y == 1) {
    return true;
  }
  else if (x % y == 0) {
    return false;
  }
  else {
    return prime(x, y-1);
  }
}
```

Let's understand how the `prime` function works. Remember, the purpose of this function is to check if a number `X` is prime.

A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. For instance, the first six prime numbers are 2, 3, 5, 7, 11, and 13.

In our `prime` function, we recursively check if `X` can be evenly divided by any number from `X - 1` down to `2`. If `X` can be evenly divided by any such number (i.e., `x % y == 0` is `true` for some `y`), then `X` is not a prime number, and the function returns `false`.

On the other hand, if `X` cannot be evenly divided by any such number (i.e., we've checked all `y` from `X - 1` down to `2` without finding an even divisor), then `X` is a prime number, and the function returns `true`.

Let's walk through the example of `prime(7, 6)`:

1. Since `y` is not `1`, and `7 % 6` does not equal `0`, we make the recursive call `prime(7, 5)`.

2. Again, `y` is not `1`, and `7 % 5` does not equal `0`, so we make the recursive call `prime(7, 4)`.
3. This pattern continues until we reach `prime(7, 2)`, which again does not evenly divide `7`, so we make the last recursive call `prime(7, 1)`.
4. Now `y` is `1`, so we return `true`, unwinding the recursion and indicating that `7` is indeed a prime number.

This function correctly identifies whether a number `X` is prime. As with all recursive functions, it is important to understand the base case(s) and how the recursive calls are working towards reaching them. The process of "winding" (making recursive calls) and "unwinding" (returning from recursive calls) is central to how recursion works.

18.2.2 Example 2: Subset Sum Problem

The following function, `isSubsetSum`, checks whether a subset of an integer array `set` can sum up to a given number `sum`. Here, `n` represents the number of array elements to consider. We don't directly use `set.length` as recursive calls need to examine only part of the array.

```
boolean isSubsetSum(int set[], int n, int sum) {
  if (sum == 0) {
    return true;
  }
  if ((n == 0) && (sum != 0)) {
    return false;
  }
  if (set[n - 1] > sum) {
    return isSubsetSum(set, n - 1, sum);
  }
  return isSubsetSum(set, n - 1, sum) || isSubsetSum(set, n - 1, sum - set[n - 1]);
}
```

To understand how this function works, we need to consider the problem it's trying to solve: checking if there is a subset of `set` that sums up to `sum`. It does this by recursively examining each element in `set` and checking two cases:

1. Is there a subset within the first `n - 1` elements of `set` that sums to `sum` (i.e., the current element is not included in the sum)?
2. Is there a subset within the first `n - 1` elements that sums to `sum - set[n - 1]` (i.e., the current element is included in the sum)?

Before checking these two cases, we have three base cases:

1. If `sum == 0`, the function returns `true`. This is because an empty set always sums to `0`.

2. If `n == 0` (i.e., there are no more elements to consider) and `sum` is not zero, the function returns `false` as there are no more numbers to add up to `sum`.
3. If the current element (i.e., `set[n - 1]`) is larger than `sum`, we cannot include this element in the sum, so we only check for a subset within the first `n - 1` elements that sum to `sum`.

To illustrate this, let's trace `isSubsetSum([3, 1, 5, 9, 12], 5, 9)`, which checks if there's a subset of `[3, 1, 5, 9, 12]` that sums to `9`.

The recursive tree will look like this:

```
isSubsetSum([3, 1, 5, 9, 12], 5, 9)
|____ isSubsetSum([3, 1, 5, 9, 12], 4, 9)
|     |____ isSubsetSum([3, 1, 5, 9, 12], 3, 9)
|     |     |____ isSubsetSum([3, 1, 5, 9, 12], 2, 9)
|     |     |     |____ isSubsetSum([3, 1, 5, 9, 12], 1, 9)  => False
|     |     |     |____ isSubsetSum([3, 1, 5, 9, 12], 1, 6)  => False
|     |     |____ isSubsetSum([3, 1, 5, 9, 12], 2, 4)
|     |           |____ isSubsetSum([3, 1, 5, 9, 12], 1, 4) => False
|     |           |____ isSubsetSum([3, 1, 5, 9, 12], 1, 3) => True
|     |____ isSubsetSum([3, 1, 5, 9, 12], 3, 0) => True
|____ isSubsetSum([3, 1, 5, 9, 12], 4, -3)  => False
```

So, the function will return `true` as there exists a subset (3, 1, 5) that sums up to `9`.

By recursively breaking down the problem, `isSubsetSum` can effectively and efficiently find whether there's a subset that matches the desired sum. This illustrates how recursive functions can solve complex problems that might be non-intuitive or hard to solve with iterative methods.

18.2.3 Example 3: Basketball Scoring Combinations

In this example, we create a function `paths` to calculate the number of ways to reach a given score in a basketball game, given that points can be accumulated in increments of 1, 2, or 3. If `n = 3`, for instance, `paths` will return 4, since there are four different ways to accumulate 3 points: `1+1+1`, `1+2`, `2+1`, and `3`.

```
int paths(int n) {
  if (n == 1) {
    return 1;
  }
  if (n == 2) {
    return 2;
```

```
  }
  if (n == 3) {
    return 4;
  }
  return paths(n - 1) + paths(n - 2) + paths(n - 3);
}
```

The function uses three base cases (`n == 1`, `n == 2`, `n == 3`) and three recursive calls, each corresponding to scoring 1, 2, or 3 points respectively. Let's dive into how this function works. This recursive function is a little more complex than the previous factorial example because it has three base cases and makes three recursive calls. However, the principle is the same: we build up a complex calculation from simpler parts, and then start "unwinding" once we hit the base cases.

The `paths` function takes an integer `n` as its argument and returns the number of possible ways to reach `n` points in a basketball game. This is a classic combinatorial problem: we're essentially counting the number of unique combinations of 1s, 2s, and 3s that can add up to `n`.

Let's trace the function call `paths(4)` as an example:

- First, the function checks if `n` is 1, 2, or 3 — these are our base cases. If `n` is one of these numbers, the function can immediately return the result. But in our case, `n` is 4, so we proceed to the recursive part of the function.
- Now the function makes three recursive calls: `paths(n - 1)`, `paths(n - 2)`, and `paths(n - 3)`, or `paths(3)`, `paths(2)`, and `paths(1)`. These represent the three ways we could reach 4 points: by scoring a point when we had 3, by scoring 2 points when we had 2, or by scoring 3 points when we had 1.
- Each of these calls will in turn make their own recursive calls, until they eventually hit one of the base cases. At that point, the function will start returning values back up the call stack.

So for `paths(4)`, we would have:

- `paths(4) = paths(3) + paths(2) + paths(1)`
- `paths(3) = 4` (from the base case)
- `paths(2) = 2` (from the base case)
- `paths(1) = 1` (from the base case)
- Therefore, `paths(4) = 4 + 2 + 1 = 7`

So there are seven ways to reach a score of 4 points.

This kind of tracing can help you understand how recursion builds up complex computations from simpler parts, and how the function's call stack unwinds once it hits the base cases. Try applying this method of tracing to other recursive functions to get a feel for how they work.

Remember to take the time to understand how each of these functions work. Recursion can seem complex at first, but with practice, it becomes a powerful tool in your programming arsenal.

18.3 Tracing Recursive Code

While creating recursive functions, it is beneficial to approach the task in a top-down manner. Trust that the recursive call will correctly solve the subproblem, and then use that result, as you would with any other function, to solve the original problem.

However, when analyzing or tracing a recursive function, you do need to understand how the function operates. Tracing a few recursive functions helps you grasp the behavior of recursion. With experience, you won't need to trace through every detail. Your understanding of recursion will gradually solidify, and your confidence will grow.

Recursive calls operate in two phases: the "winding" phase, where information is passed from one recursive call to the next, and the "unwinding" phase, where return values are passed back. These phases are sometimes overlooked, but they are essential to understanding recursion.

In the winding phase, any parameter passed through the recursive call travels forward until the base case is reached. In the unwinding phase, the function's return value (if any) travels backward to the calling function.

Let's illustrate this with an example: a recursive function to compute the factorial of a number.

```
int factorial(int n) {
  if (n == 0) {
    return 1;  // base case
  } else {
    return n * factorial(n - 1);  // recursive case
  }
}
```

In this case, the information (the decremented value of `n`) flows forward during the winding phase, and the computed factorial flows backward during the unwinding phase.

A recursive function can also pass multiple parameters forward. For instance, a function that recursively sums the values in an array might pass the array and the current index in the winding phase, and return the cumulative sum during the unwinding phase.

```
int arraySum(int[] arr, int n) {
  if (n <= 0) {
```

```
    return 0;  // base case
  } else {
    return arr[n - 1] + arraySum(arr, n - 1);  // recursive case
  }
}
```

In this case, the array and the index (`n - 1`) are passed forward in the winding phase, while the summed value so far is passed back during the unwinding phase.

To better understand the idea of recursion, consider it as a domino effect where each recursive call is like tipping over a domino. These rules can guide your thinking:

1. Just like the first domino has to be tipped manually, the base case solution in recursion is computed non-recursively.
2. Before any given domino can fall, all preceding dominos must have been tipped over. Similarly, before any recursive call can complete, all deeper recursive calls must complete.

Through this process of tracing and understanding, recursion becomes a powerful tool in problem-solving.

18.4 Practice Exercises

Now that we've laid out the theory of recursion and tracing recursive calls, let's get some hands-on practice with some exercises. Remember, the more you practice, the better you'll understand how recursion works.

18.4.1 Exercise 1: Factorial Function

Consider the recursive function for calculating the factorial of a number:

```
int factorial(int n) {
  if (n == 0) {
    return 1;
  }
  else {
    return n * factorial(n - 1);
  }
}
```

Task: Trace the recursive calls made when `factorial(4)` is called.

18.4.2 Exercise 2: Fibonacci Series

Consider the recursive function for generating the Fibonacci series:

```
int fibonacci(int n) {
  if (n <= 1) {
    return n;
  }
  else {
    return fibonacci(n - 1) + fibonacci(n - 2);
  }
}
```

Task: Trace the recursive calls made when `fibonacci(5)` is called. Pay close attention to how the same sub-problems are solved multiple times.

18.4.3 Exercise 3: Sum of Array Elements

Consider the recursive function for finding the sum of all elements in an array:

```
int sumArray(int[] arr, int n) {
  if (n <= 0) {
    return 0;
  }
  else {
    return sumArray(arr, n-1) + arr[n-1];
  }
}
```

Task: Given the array `{1, 2, 3, 4, 5}`, trace the recursive calls made when `sumArray(arr, 5)` is called.

Take your time working through these exercises and understand how each recursive call is made. After you've completed them, try creating your own recursive functions and tracing through their calls.

If you're still finding it difficult to understand how the tracing is done, don't hesitate to ask for help. Recursion is a difficult concept to grasp initially, but with practice and patience, it'll become much clearer.

18.5 Solution Walkthrough: Factorial Function

Let's trace the recursive calls made when we calculate `factorial(4)` using our recursive factorial function. Here's the function again for reference:

```
int factorial(int n) {
  if (n == 0) {
    return 1;
  }
  else {
    return n * factorial(n - 1);
  }
}
```

When we call `factorial(4)`, we're not at the base case (`n == 0`), so we go to the else clause. Here, we calculate `4 * factorial(4 - 1)`, or `4 * factorial(3)`. Now, we need to calculate `factorial(3)` before we can continue. This process repeats for each recursive call:

- `factorial(4) = 4 * factorial(3)`
- `factorial(3) = 3 * factorial(2)`
- `factorial(2) = 2 * factorial(1)`
- `factorial(1) = 1 * factorial(0)`

Once we reach `factorial(0)`, we've hit our base case, and the function returns `1`:

- `factorial(0) = 1`

Now we can start unwinding the stack of calls:

- `factorial(1) = 1 * 1 = 1`
- `factorial(2) = 2 * 1 = 2`
- `factorial(3) = 3 * 2 = 6`
- `factorial(4) = 4 * 6 = 24`

So, `factorial(4)` returns `24`. You can see how each recursive call builds on the previous one, and the base case provides the starting point for the calculation to unwind.

This same approach can be applied to other recursive functions to understand how they work. Try it out with the remaining exercises to practice tracing recursive calls.

19 Binary Search

19.1 Recursive Linear Search

Let's imagine the following scenario: You're in a well-organized library, each shelf categorized by the author's last name in alphabetical order from A to Z. You're tasked to find a book by your favorite author, say, 'Margaret Atwood'.

So, you start at 'A' and proceed book by book, author by author, until you eventually find the book by Atwood. This, in essence, is a prime example of the linear search algorithm. Simple, right?

Let's formalize this process with Java:

```
public int linearSearch(int[] arr, int key){
    for(int i=0;i<arr.length;i++){
        if(arr[i] == key){
            return i;
        }
    }
    return -1;
}
```

This method receives an array `arr` and a key `key`. It uses a loop to traverse the array, checking if the current element is equal to the searched key. If it is, it returns the current index `i`. If it doesn't find the key in the entire array, it returns -1.

We start the search at the beginning of the array, just like we started at the 'A' section in our library, and examine each element just like we looked at each book. When we find a match, we stop searching, just like we did when we found our Atwood's book.

Despite its simplicity, the linear search algorithm isn't very efficient. For example, let's say the library decided to include all authors worldwide. With the linear search strategy, you might end up examining books by millions of authors before you reach Margaret Atwood!

This brings us to our problem – How can we make this search more efficient?

Since our last chapter was on recursion, let's try to solve this problem using recursion. We'll call our method `linearSearchRecursive`:

```java
public int linearSearchRecursive(int[] arr, int key, int index){
    if(index >= arr.length){
        return -1;
    }
    else if(arr[index] == key){
        return index;
    }
    else{
        return linearSearchRecursive(arr, key, index+1);
    }
}
```

The `linearSearchRecursive` method is a recursive function that performs a linear search on an array of integers. A linear search is a simple algorithm that checks each element of the array in order until it finds the target value or reaches the end of the array. The method takes three parameters: `arr`, which is the array to be searched, `key`, which is the value to be found, and `index`, which is the current position in the array.

The method has three cases:

- The base case 1: If `index` is equal to or greater than the length of the array, it means that the end of the array has been reached and the key was not found. In this case, the method returns `-1` to indicate that the search was unsuccessful.
- The base case 2: If the element at `arr[index]` is equal to `key`, it means that the key has been found at the current position. In this case, the method returns `index` to indicate the location of the key in the array.
- The recursive case: If neither of the base cases are true, it means that the key has not been found yet and there are more elements to check. In this case, the method makes a recursive call to itself with the same array and key, but with an incremented index (`index+1`). This way, the method moves forward in the array until it either finds the key or reaches the end.

The data flows through the recursive calls via the parameters and return values. The parameters (`arr`, `key`, and `index`) help in propagating information forward through the recursive calls until reaching one of the base cases. The return values (`-1` or `index`) relay information back from the base case to the original caller.

Let's illustrate this with an example: Suppose we want to search for `5` in the array `{3, 1, 4, 2, 5}` using `linearSearchRecursive`. We start by calling `linearSearchRecursive(arr, 5, 0)`, where `arr` is `{3, 1, 4, 2, 5}`, `key` is `5`, and `index` is `0`. The recursive tree will look like this:

```
linearSearchRecursive(arr, 5, 0)
|____ linearSearchRecursive(arr, 5, 1)
```

```
|____ linearSearchRecursive(arr, 5, 2)
      |____ linearSearchRecursive(arr, 5, 3)
            |____ linearSearchRecursive(arr, 5, 4) => returns 4
```

At each recursive call, we check if we have reached one of the base cases. If not, we make another recursive call with an incremented index. When we reach `linearSearchRecursive(arr, 5, 4)`, we find that `arr[4]` is equal to `key`, so we return `4`. This value is then passed back up the call stack until it reaches the original caller. So, `linearSearchRecursive(arr, 5, 0)` returns `4`, indicating that `5` is found at index `4` in the array.

19.2 Optimizing Linear Search

Our recursive linear search algorithm successfully finds a given element within an array. However, in each recursive call, we reduce our search space by just one element. It's akin to taking one step at a time when we could be leaping. Wouldn't it be nice if we could shrink this search space by more than a single element at each step?

Let's imagine again that we're in our library from earlier, but this time, we have a partner. You start at 'A' and your partner starts at 'Z'. Now you're moving towards each other, each drawing closer to 'M', where Atwood's books are likely to reside. If either of you find the book, you signal the other, and the process hinges to a stop.

Let's translate this idea into code:

```java
public int optimizedLinearSearch(int[] arr, int left, int right, int key){
    if(left>right) {
        return -1;
    }
    if(arr[left] == key){
        return left;
    }
    if(arr[right] == key){
        return right;
    }
    return optimizedLinearSearch(arr, left + 1, right - 1, key);
}
```

The `optimizedLinearSearch` method is another recursive function that performs a linear search on an array of integers, but with some optimizations. The method takes four parameters: `arr`, which is the array to be searched, `left`, which is the leftmost index of the subarray to be searched, `right`, which is the rightmost index of the subarray to be searched, and `key`, which is the value to be found.

The method has three cases:

- The base case 1: If `left` is greater than `right`, it means that the subarray is empty and the key was not found. In this case, the method returns `-1` to indicate that the search was unsuccessful.
- The base case 2: If the element at `arr[left]` or `arr[right]` is equal to `key`, it means that the key has been found at one of the ends of the subarray. In this case, the method returns `left` or `right` to indicate the location of the key in the array.
- The recursive case: If neither of the base cases are true, it means that the key has not been found yet and there are more elements to check. In this case, the method makes a recursive call to itself with the same array and key, but with a smaller subarray (`left + 1` to `right - 1`). This way, the method eliminates two elements from the search space at each recursive call until it either finds the key or reaches an empty subarray.

The data flows through the recursive calls via the parameters and return values. The parameters (`arr`, `left`, `right`, and `key`) help in propagating information forward through the recursive calls until reaching one of the base cases. The return values (`-1`, `left`, or `right`) relay information back from the base case to the original caller.

The optimization in this method comes from checking both ends of the subarray at each recursive call, instead of just one element as in `linearSearchRecursive`. This reduces the number of recursive calls needed to find the key or reach an empty subarray. For example, if we want to search for `5` in the array `{3, 1, 4, 2, 5}` using `optimizedLinearSearch`, we start by calling `optimizedLinearSearch(arr, 0, 4, 5)`, where `arr` is `{3, 1, 4, 2, 5}`, `left` is `0`, `right` is `4`, and `key` is `5`. The recursive tree will look like this:

```
optimizedLinearSearch(arr, 0, 4, 5)
|____ optimizedLinearSearch(arr, 1, 3, 5)
      |____ optimizedLinearSearch(arr, 2, 2, 5) => returns -1
```

At each recursive call, we check if we have reached one of the base cases. If not, we make another recursive call with a smaller subarray. When we reach `optimizedLinearSearch(arr, 2, 2, 5)`, we find that neither `arr[2]` nor `arr[2]` is equal to `key`, and that `left > right`. So we return `-1`. This value is then passed back up the call stack until it reaches the original caller. So, `optimizedLinearSearch(arr, 0, 4, 5)` returns `-1`, indicating that `5` is not found in the array.

Note that this method works correctly only if there are no duplicates in the array. If there are duplicates, it may return a different index than expected or miss some occurrences of the key. For example, if we want to search for `4` in the array `{3, 4, 4, 2}`, using `optimizedLinearSearch(arr, 0, 3, 4)`, we will get `-1` as a result instead of `1` or `2`. This is because both ends of the subarray are eliminated at each recursive call until an empty subarray is reached.

The worst case complexity analysis of both snippets is as follows:

- The `linearSearchRecursive` method has a worst-case complexity of **O(n)**, where **n** is the length of the array. This is because in the worst case, the key is not present in the array or is the last element of the array, and the method has to check every element of the array until it reaches the end. The number of recursive calls is equal to **n** in this case.
- The `optimizedLinearSearch` method has a worst-case complexity of **O(n/2)** - because in the worst case, the key is not present in the array or is somewhere in the middle of the array, and the method has to check half of the elements of the array until it reaches an empty subarray - which ends up being also **O(n)** in Big-O notation.

So it looks like the worst-case scenario hasn't vastly improved. We'd still potentially have to traverse half the array size if the element resides in the middle. In the worst-case scenario, the time complexity remains O(n).

This leads us to a realization: minor tweaks to our approach may not provide the drastic improvement we hope for. Is there some drastic change we can make to eliminate substantial amounts of data from our search space, not just one or two elements, with every recursive call?

19.3 Divide and Conquer

When faced with a complex problem, one of the most effective strategies is often to break it down into smaller, more manageable parts. This is the essence of the 'divide and conquer' strategy. We divide the problem into several sub-problems that are similar to the original but smaller in size. We conquer the sub-problems by solving them recursively. Finally, we combine the solutions of the sub-problems to solve the original problem.

The divide and conquer strategy is especially effective when the sub-problems can be solved independently and the solutions can be combined efficiently. In the context of searching, we can apply this strategy by dividing the search space into smaller parts and searching each part separately. If we can eliminate large parts of the search space at each step, we can find the target value much faster.

Let's see how we can apply this to Search, and our original problem of looking for books in a library!

The journey to our favorite author's book in the library hasn't been as fast as we initially desired. Checking every book, or even every other book, can still be cumbersome in a large library.

So, let's revisit our library metaphor and imagine a different scenario. Now, you have an idea. Instead of searching from 'A' to 'Z' linearly, you begin your search in the middle of the library, around 'M.' If Atwood falls in the latter half, ignore everything from 'A to L.' And voila! Half the library gets eliminated instantly.

But in order for this to work, we need to make an assumption – the library is sorted alphabetically. If it isn't, we'll have to sort it first, which is a whole other problem. But if it is, we can apply the divide and conquer strategy to our search.

Now let's put it into practice by creating a new method called `drasticallyFasterSearch`:

```
public int drasticallyFasterSearch(int[] arr, int left, int right, int key) {
    int mid = left + (right - left) / 2;
    if (arr[mid] == key)
        return mid;
    else if (arr[mid] > key)
        return optimizedLinearSearch(
          java.utils.Arrays.copyOfRange(arr, 0, mid - 1), 0, mid - 1, key
        );
    else
        return optimizedLinearSearch(
          java.utils.Arrays.copyOfRange(arr, mid + 1, right), 0, right - mid - 1, key
        );
}
```

The `drasticallyFasterSearch` method is a hybrid function that combines our earlier `optimizedLinearSearch` to our new Divide and Conquer strategy. The method takes four parameters: `arr`, which is the array to be searched, `left`, which is the leftmost index of the subarray to be searched, `right`, which is the rightmost index of the subarray to be searched, and `key`, which is the value to be found.

The method has three cases:

- The base case: If the element at the middle of the subarray (`arr[mid]`) is equal to `key`, it means that the key has been found at that position. In this case, the method returns `mid` to indicate the location of the key in the array.
- The recursive case 1: If the element at the middle of the subarray (`arr[mid]`) is greater than `key`, it means that the key must be in the left half of the subarray. In this case, the method makes a recursive call to `optimizedLinearSearch` with a smaller subarray (`arr[0]` to `arr[mid-1]`). This way, the method eliminates half of the elements from the search space and then applies optimized linear search on the remaining elements.
- The recursive case 2: If the element at the middle of the subarray (`arr[mid]`) is less than `key`, it means that the key must be in the right half of the subarray. In this case, the method makes a recursive call to `optimizedLinearSearch` with a smaller subarray (`arr[mid+1]` to `arr[right]`). This way, the method eliminates half of the elements from the search space and then applies optimized linear search on the remaining elements.

The data flows through the recursive calls via the parameters and return values. The parameters (`arr`, `left`, `right`, and `key`) help in propagating information forward through the

recursive calls until reaching one of the base cases. The return values (`mid`, `-1`, `left`, or `right`) relay information back from the base case to the original caller.

Let's illustrate this with an example: Suppose we want to search for `5` in the sorted array `{1, 2, 3, 4, 5}` using `drasticallyFasterSearch`. We start by calling `drasticallyFasterSearch(arr, 0, 4, 5)`, where `arr` is `{1, 2, 3, 4, 5}`, `left` is `0`, `right` is `4`, and `key` is `5`. The recursive tree will look like this:

```
drasticallyFasterSearch(arr, 0, 4, 5)
|____ drasticallyFasterSearch(arr, 3, 4, 5)
      |____ optimizedLinearSearch(arr[3..4], 0 ,1 ,5) => returns 1
```

At each recursive call, we check if we have reached one of the base cases. If not, we make another recursive call with a smaller subarray. When we reach `optimizedLinearSearch(arr[3..4], 0 ,1 ,5)`, we find that `arr[4]` is equal to `key`, so we return `1`. This value is then added to `left` (which is `3`) and passed back up the call stack until it reaches the original caller. So, `drasticallyFasterSearch(arr, 0 ,4 ,5)` returns `(3 + 1) = 4`, indicating that `5` is found at index `4` in the array.

19.4 Even faster `drasticallyFasterSearch`

Building on our previous exploration, we can see a pattern emerging. We are using our `drasticallyFasterSearch` to divide our problem, but when it comes to conquering, we are falling back to `optimizedLinearSearch`. This means we are not fully utilizing the strength of our divide and conquer approach. We have an opportunity for optimization here: what if, instead of resorting to linear search, we applied the same divide and conquer strategy again? This observation leads us to the Binary Search algorithm, a more efficient way to search sorted arrays, which would look like this:

```java
public int drasticallyFasterSearch(int[] arr, int left, int right, int key) {
    int mid = left + (right - left) / 2;
    if (arr[mid] == key)
        return mid;
    else if (arr[mid] > key)
        return drasticallyFasterSearch(
          java.utils.Arrays.copyOfRange(arr, 0, mid - 1), 0, mid - 1, key
        );
    else
        return drasticallyFasterSearch(
          java.utils.Arrays.copyOfRange(arr, mid + 1, right), 0, right - mid - 1, key
        );
```

```
}
```

But would this actually work? We need to use what we learnt about writing recursive functions to find out - what are the base cases? What are the recursive cases? How does the data flow through the recursive calls?

> The analysis of the `drasticallyFasterSearch` method is left as an exercise for the reader.

One fatal flaw with the `drasticallyFasterSearch` function, as it is currently written, is that there is only one base case - when we find the element and return the `mid` index. If the element we're searching for is not in the array, the function will enter into an infinite loop because it lacks a base case to handle this situation.

So, we need an additional base case to handle the scenario where the element is not in the array. This needs to be a condition that detects if we've run out of elements in the array that we haven't yet searched.

This is where the skill of tracing recursive functions comes in handy. We can trace the function to see what happens when we run out of elements to search. Let's trace the function for the example we used earlier: searching for `5` in the sorted array `{1, 2, 3, 4, 5}` results in the following recursive call sequence:

```
drasticallyFasterSearch(arr, 0, 4, 5)
|____ drasticallyFasterSearch(arr, 3, 4, 5)
      |____ drasticallyFasterSearch(arr, 4, 4, 5)
            => returns 4
```

But what happens if the number we're searching for isn't in the array? What if we're looking for `6` in the same array?

```
drasticallyFasterSearch(arr, 0, 4, 6)
|____ drasticallyFasterSearch(arr, 3, 4, 6)
      |____ drasticallyFasterSearch(arr, 4, 4, 6)
            |____ drasticallyFasterSearch(arr, 5, 4, 6)
```

This last recursive call, `drasticallyFasterSearch(arr, 5, 4, 6)` is problematic because it makes no sense to start at an index higher than our end index. This means we've exhausted all potential elements in our search.

This suggests that we need an additional base case to handle when our `left` index exceeds our `right` index. This makes sense because there is no reason for `left` to be on the right of `right`! So, we should modify our search method to be:

```
public int drasticallyFasterSearch(int[] arr, int left, int right, int key) {
    if (right < left)
        return -1;

    int mid = left + (right - left) / 2;

    if (arr[mid] == key)
        return mid;
    else if (arr[mid] > key)
        return drasticallyFasterSearch(
          java.utils.Arrays.copyOfRange(arr, 0, mid - 1), 0, mid - 1, key
        );
    else
        return drasticallyFasterSearch(
          java.utils.Arrays.copyOfRange(arr, mid + 1, right), 0, right - mid - 1, key
        );
}
```

Now, we can see that when `right` becomes less than `left`, the function will correctly return `-1`, signaling that our intended key isn't present. Note that in the case where `left` equals `right`, we still want to check the `arr[mid]`, so `right < left`, not `right <= left`, is our base case.

As we apply the divide and conquer methodology here, our drasticallyFasterSearch method is now a working implementation of the Binary Search algorithm. In each recursive call, we cut our search space in half, drastically reducing the number of elements searched. For a list of length n, in the worst-case, we'll make roughly log(n) recursive calls, scaling our runtime far better than a linear search would.

Indeed, this is a powerful tool for searching sorted data. By fully utilizing divide and conquer, we've been able to drastically speed up our search. This emphasizes the importance of understanding the fundamentals of recursion and problem decompositions as key building blocks in algorithm and data structure design.

As the reader (or student!) progresses, they may find more elegant ways to implement this algorithm, or variations of it, which solve complex problems in different contexts. Here is a variation of the this dearch algorithm that is more efficient and doesn't require copying the array at each recursive call:

```
public int drasticallyFasterSearch(int[] arr, int left, int right, int key) {
    if (right >= left) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == key)
```

```
            return mid;
        else if (arr[mid] > key)
            return drasticallyFasterSearch(arr, left, mid - 1, key);
        else
            return drasticallyFasterSearch(arr, mid + 1, right, key);
    } else {
        return -1;
    }
}
```

19.5 Unveiling Binary Search

The name "Binary Search" comes from the fact that this method divides the search space into two (binary) at each step of the algorithm. It is a fundamental technique used in computer science and a cornerstone in algorithm design.

The final implementation of the Binary Search algorithm is as follows:

```
public int binarySearch(int[] arr, int left, int right, int key) {
    if (right >= left) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == key)
            return mid;
        else if (arr[mid] > key)
            return binarySearch(arr, left, mid - 1, key);
        else
            return binarySearch(arr, mid + 1, right, key);
    } else {
        return -1;
    }
}
```

By rethinking our approach and exploiting the sorted nature of our data, we devised a strategy that significantly reduces the time spent searching. This concludes the core of Binary Search, demonstrating how powerful a shift in perspective can be in problem-solving.

Part VIII

Trees

20 Tree Data Structure

20.1 Background

When we first begin to learn about data structures, we often encounter linear structures such as arrays, linked lists, stacks, and queues. These structures, as the term 'linear' suggests, store data in a sequential manner (Figure 20.1). This makes them great for situations where the data can be naturally expressed as a sequence, for instance, a to-do list or a queue at a coffee shop.

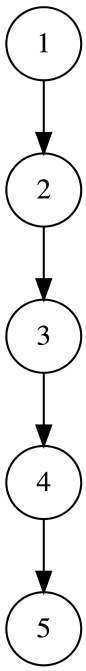


Figure 20.1: A representation of a linked list structure, highlighting its linear nature.

Let's consider the linked list. Each node in the linked list has a reference to the next node, creating a chain of elements. This works well when the data in the structure has a natural sequential or linear relationship. However, as we delve deeper into more complex data problems, we begin to see that not all data fits neatly into a linear relationship.

Consider an organizational hierarchy within a company. Each person (except the CEO) reports to exactly one person. However, each manager can have multiple direct reports. In this case, each node (person) might need to reference more than one node (Figure 20.2). Using a linked list to represent this data would be cumbersome, as it would not capture the multiple relationships that exist between a manager and their direct reports.

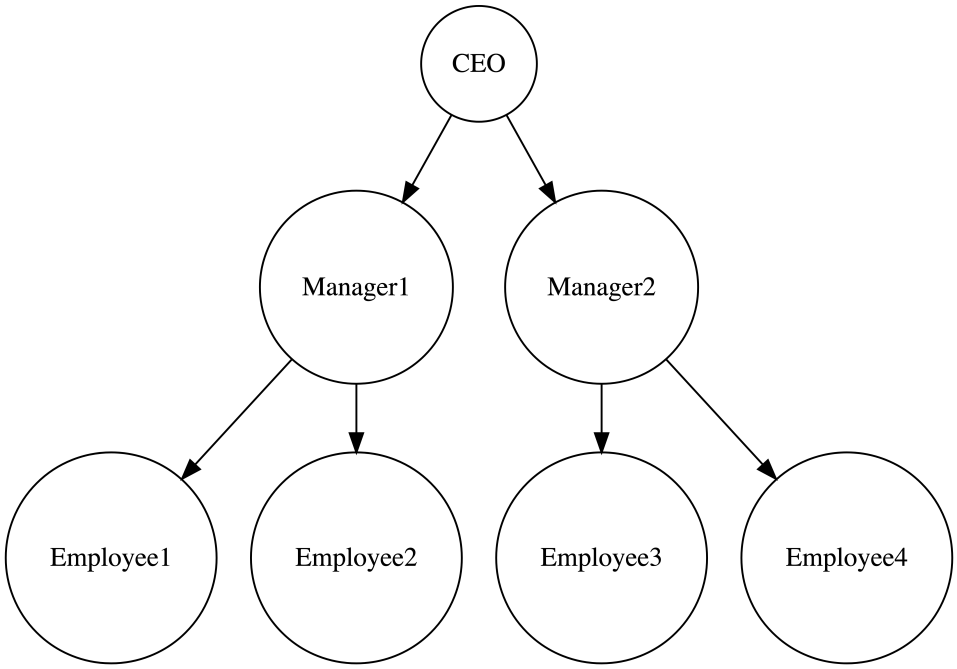


Figure 20.2: An example of an organizational hierarchy within a company, illustrating the non-linear relationships that exist.

Another example is a file system on your computer. Each folder can contain multiple files or other folders. But, each file or folder is contained within exactly one other folder. Once again, we see the need for a data structure that can capture these multiple relationships (Figure 20.3).

These examples bring to light a common pattern: there are situations where our data is not linear, but hierarchical. Hierarchies exist in various forms in the real world, from biological classifications to the layout of web pages, and modeling these hierarchies accurately in our programs allows us to reflect the reality more truthfully.

Enter the concept of Trees.

A tree in computer science is a hierarchical data structure that can model these kinds of relationships (Figure 20.4). This structure gets its name from a real-world tree, but with a

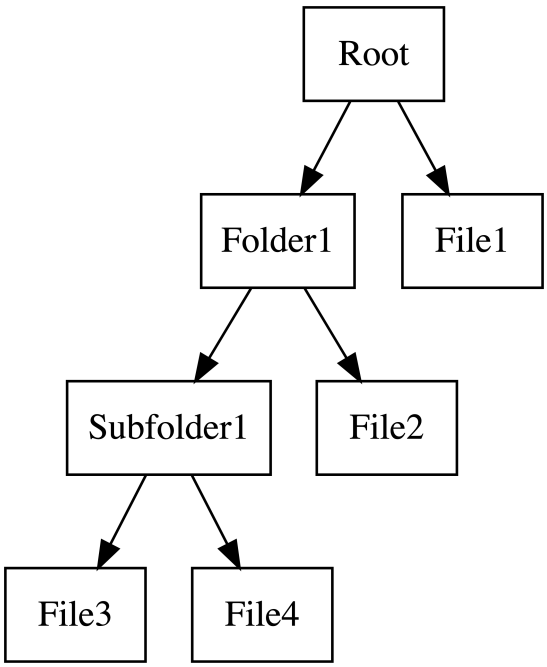


Figure 20.3: A representation of a simple file system, showing how folders can contain multiple other folders or files.

twist - the tree data structure is often visualized with the root at the top and the leaves at the bottom, resembling an upside-down real-world tree.

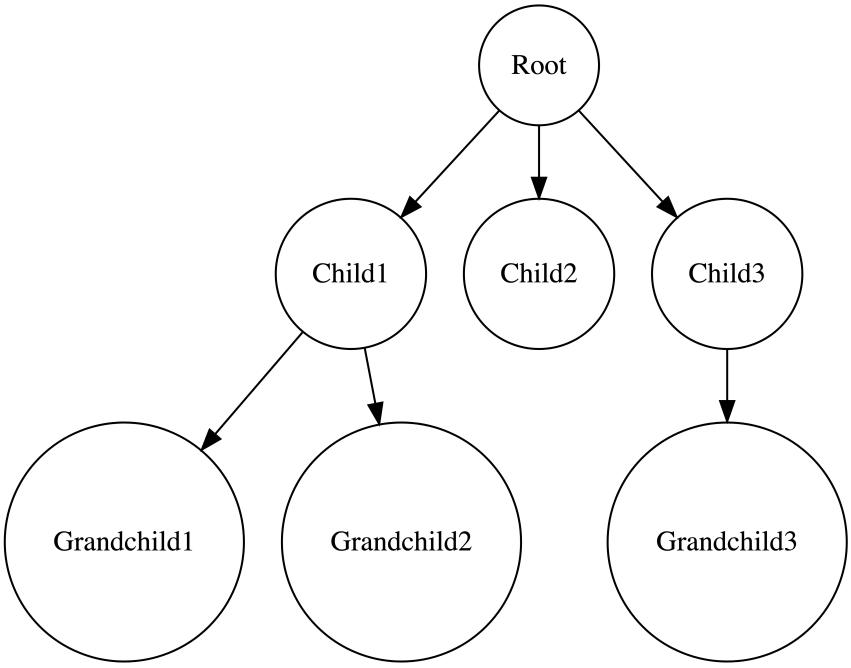


Figure 20.4: An illustration of a simple tree data structure, with key components labeled.

The tree structure is a fundamental shift from our previously understood data structures. It can open new doors to solving problems and provide an efficient way to organize and model complex relationships in our data.

In the following sections, we'll take a deeper dive into the world of trees, understanding their properties, variations, and how they can revolutionize the way we think about hierarchical data relationships.

20.2 Introduction to Trees

After establishing the need for a data structure that can handle more complex relationships, we can now formally introduce the concept of trees.

A tree in computer science is an abstract data type that simulates a hierarchical structure with a set of interconnected nodes. The key characteristic that separates trees from other data structures is that no node in the tree may be connected to more than one parent node, thereby preventing any possibility of a loop in the data structure.

Let's get familiar with some fundamental tree terminologies (see Figure 20.5):

- **Node**: A unit or location in the tree where data is stored. It's similar to a link in a linked list or an element in an array.
- **Root**: The topmost node in the tree that doesn't have a parent.
- **Edge**: The connection between two nodes.
- **Parent**: A node which has one or more child nodes.
- **Child**: A node which has a parent node.
- **Leaf**: A node which does not have any child nodes.
- **Subtree**: A tree consisting of a node and its descendants.
- **Degree**: The number of children of a node.
- **Level**: The distance from the root, measured in edges.
- **Height**: The distance from the node to its furthest leaf.
- **Depth**: The distance from the node to the root.

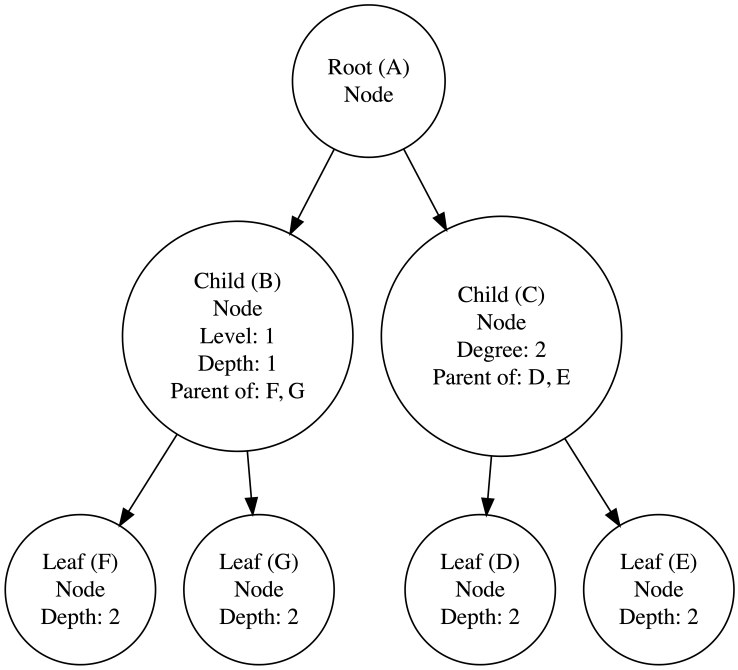


Figure 20.5: A simple tree illustrating various tree terminologies. Node A is the root, nodes B and C are children of A. Node C, having two children, demonstrates the degree of a node. Nodes D, E, F, G are leaf nodes. The subtree consists of Node C and its descendants (D, E). The level is demonstrated by the level of Node B (Level 1). The depth of Node D is 2 (from Root A). The height of the tree is 2 (distance from the Root A to furthest leaf).

The concept of trees becomes truly interesting when we delve into the various types of trees. Different types of trees can be used to solve different types of problems, and it's crucial to choose the right type for a given problem.

The simplest form of a tree is a **General Tree**, where each node can have any number of child nodes. An example of where a general tree might be useful is in the representation of a file system on a computer. In this system, each folder can contain any number of files or folders. We can illustrate this with a diagram (Figure 20.6).

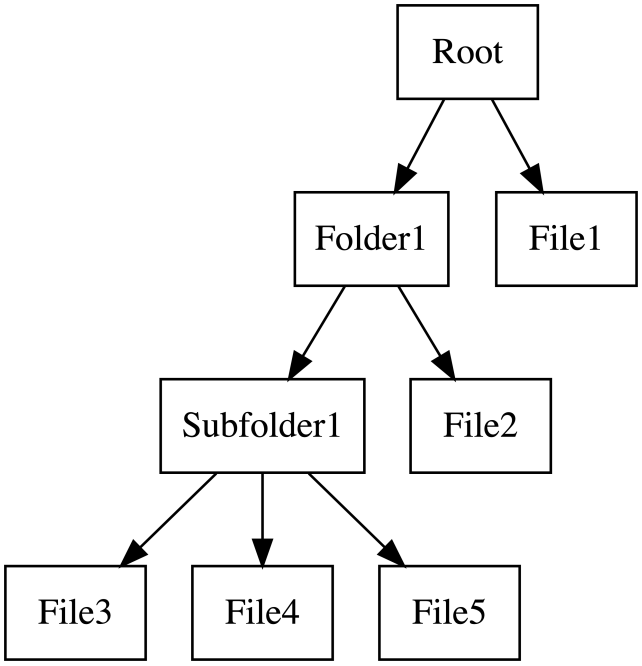


Figure 20.6: An example of a general tree representing a file system, where each node can have any number of child nodes.

However, there are also more specialized types of trees, like **Binary Trees**, where each node can have at most two children, commonly referred to as left and right child. The binary tree can be visualized in the form of a family tree where each parent (node) can have at most two children (Figure 20.7).

A further specialization is the **Binary Search Tree (BST)**, where the nodes are arranged in a manner such that for every node, nodes in the left subtree have values less than the node, and nodes in the right subtree have values greater than the node. This property makes BSTs useful for search operations, such as finding a book in a library system, where books are organized based on their identifying information (Figure 20.8).

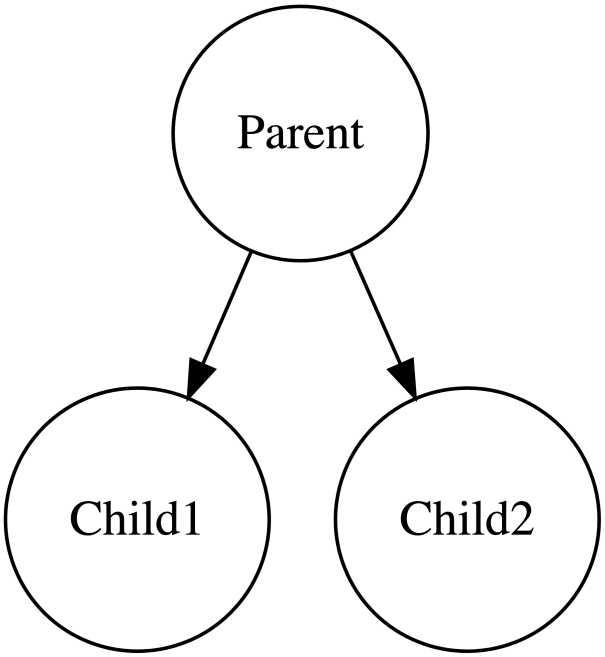


Figure 20.7: A binary tree representing a family tree, where each parent can have at most two children.

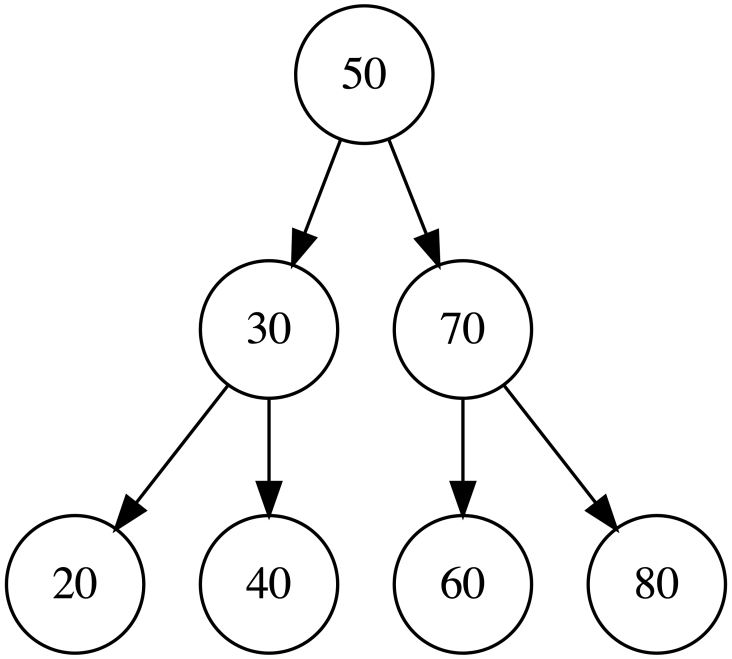


Figure 20.8: A binary search tree (BST) used in a library system for organizing books. The key property of BST is that nodes in the left subtree have values less than the parent node, and nodes in the right subtree have values greater.

There are also self-balancing trees like **AVL Trees** and **Red-Black Trees** that maintain their balance as new nodes are inserted or existing nodes are removed, which ensures that the tree remains efficient for operations like insertion, deletion, and search.

Another common type of tree is the **B-tree**, used in databases and filesystems. In a B-tree, each node can have more than two children, unlike in a binary tree. This structure allows for efficient access and modification of large blocks of data, which makes B-trees particularly suitable for use in disk-based storage systems, such as databases (Figure 20.9).

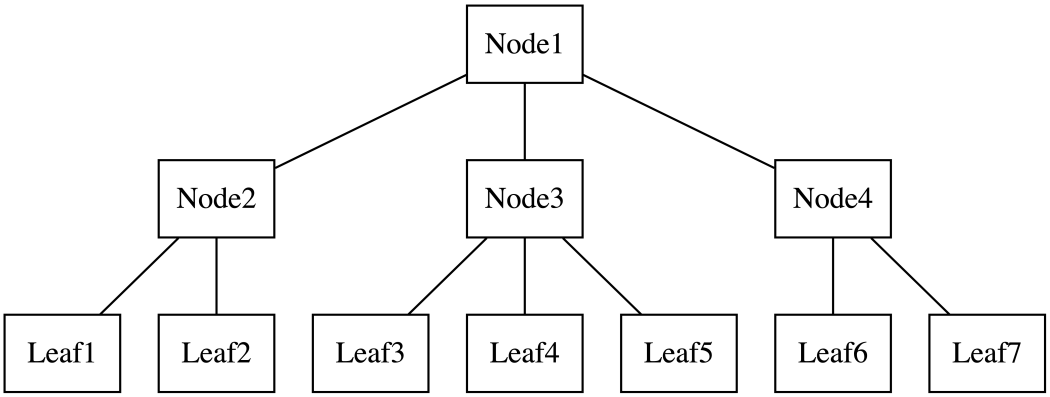


Figure 20.9: A representation of a B-tree used in a database. Each node can have more than two children, providing efficient access and modification of large data blocks.

Understanding these types of trees and how they work is crucial to using them effectively. In the next sections, we will delve deeper into some of these tree types and explore their properties and uses.

20.3 Binary Tree

After understanding the basic concept of trees, let's now focus on a special kind of tree, the Binary Tree. As we have mentioned earlier, in a binary tree, a node can have at most two children - commonly referred to as the left child and the right child. This property leads to some interesting and useful characteristics that we'll explore in this section.

The structure of a binary tree is relatively straightforward. Each node in a binary tree contains some data, a reference to the left child node, and a reference to the right child node. If a node has no left or right child, the corresponding reference is set to `null`.

In a binary tree, each level has twice as many nodes as the previous level, creating a rapidly expanding structure. This property enables binary trees to store large amounts of data in a

relatively small number of levels, which can lead to efficient search and insertion operations, given the tree is balanced.

Let's take a look at some different types of binary trees:

- **Full Binary Tree**: In this tree, every node has either 0 or 2 children. This type of tree is useful when you know that every internal node in your tree will have exactly two children. For example, in certain types of decision trees used in machine learning, each decision leads to two subsequent decisions, forming a full binary tree.

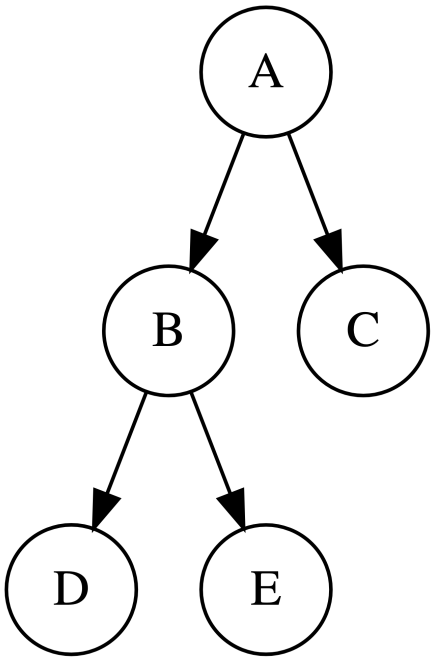


Figure 20.10: A Full Binary Tree. Every node has either 0 or 2 children.

- **Complete Binary Tree**: Every level, except possibly the last, is fully filled, and all nodes are as far left as possible. Heaps are an example of complete binary trees. In a heap, all levels are fully filled except for the last level, which is filled from left to right.

- **Perfect Binary Tree**: A perfect binary tree is both full and complete. Every level is fully filled. This kind of tree appears in some specialized data structures or algorithms, but is not common in general use.

- **Balanced Binary Tree**: The difference in height of left and right subtrees for every node is not more than k (mostly 1). Most of the tree algorithms require the tree to be balanced to maintain their optimal time complexity.

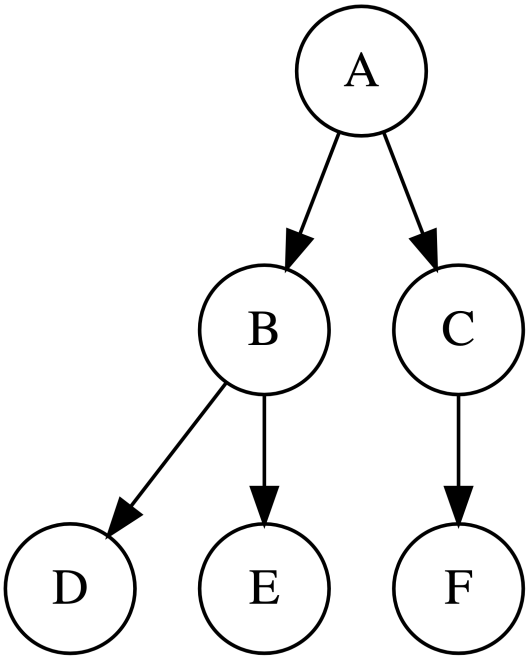


Figure 20.11: A Complete Binary Tree. Every level is fully filled except for the last, which is filled from left to right.

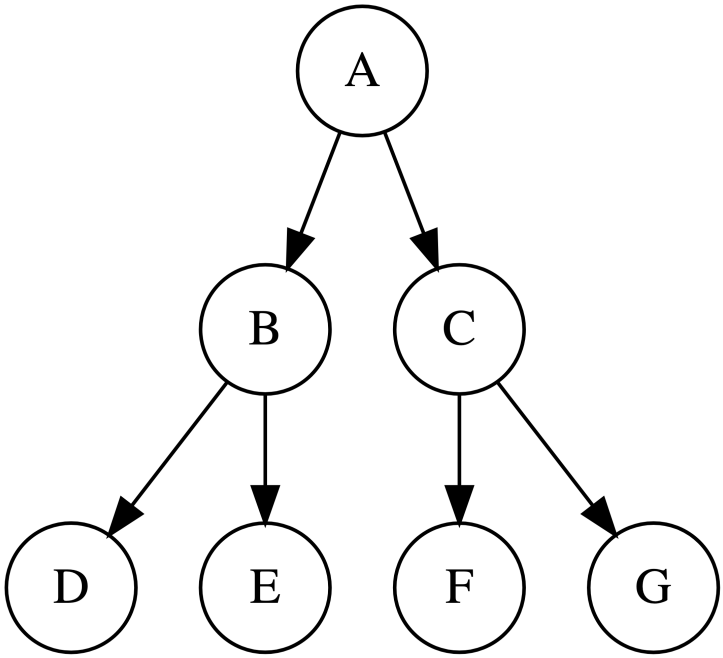


Figure 20.12: A Perfect Binary Tree. Every level is fully filled.

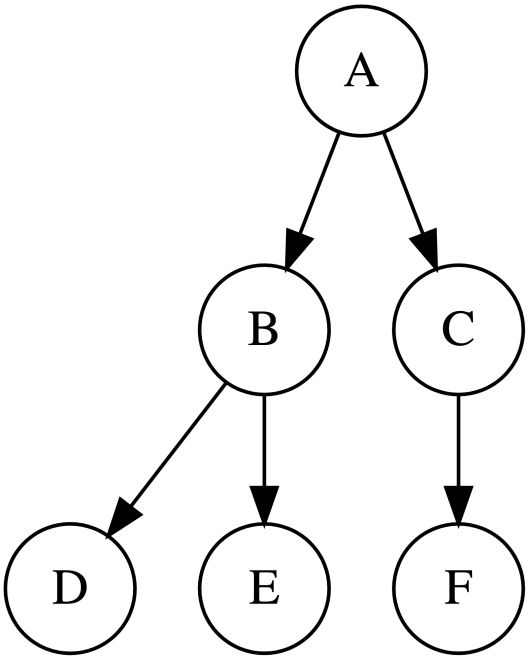


Figure 20.13: A Balanced Binary Tree. The difference in height of left and right subtrees for every node is not more than 1.

- **Degenerate (or pathological) Tree**: Each parent node has only one child. The tree behaves like a linked list, and we lose the advantages of binary trees.

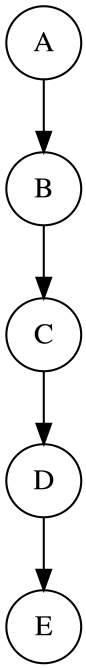


Figure 20.14: A Degenerate (or pathological) Tree. Each parent node has only one child.

20.3.1 Implementing a Binary Tree

In this section, we will be implementing a simple Binary Tree structure using Java. We'll start with the definition of the `Node` class.

```
/**
 * This class represents a node in the binary tree.
 * @param <T> This is the type parameter which represents the type of value the node will
 */
public class Node<T> {
    T value;
    Node<T> left;
    Node<T> right;

    /**
```

```
     * Node class constructor.
     * @param value This is the value that the node will hold.
     */
    public Node(T value) {
        this.value = value;
        this.left = null;
        this.right = null;
    }
}
```

In the code above, we defined a generic `Node` class with a type parameter `T`. The `Node` class has three fields: `value` of type `T` representing the data the node holds, and `left` and `right` of type `Node<T>` representing the left and right children of the node.

Next, let's define our `BinaryTree` class.

```
/**
 * This class represents a binary tree.
 * @param <T> This is the type parameter which represents the type of value the nodes in t
 */
public class BinaryTree<T> {
    Node<T> root;

    /**
     * BinaryTree class constructor.
     */
    public BinaryTree() {
        this.root = null;
    }
}
```

In the `BinaryTree` class, we defined a `root` of type `Node<T>` representing the root of the tree. Practical applications of binary trees usually include additional operations like adding a node, deleting a node, checking if a value exists in the tree, and various ways of traversing the tree (in-order, pre-order, post-order).

20.4 Common Operations on Binary Trees

20.4.1 Traversals and Orderings

Trees, being a nonlinear data structure, cannot be traversed in a single run like arrays, linked lists, or other linear data structures. Therefore, they require some specific methods to traverse

their nodes, visit each node once, and perform some operations such as search, update, or accumulate.

Three common types of depth-first traversal are **in-order**, **pre-order**, and **post-order**.

20.4.1.1 In-Order Traversal

In in-order traversal, the process is as follows: 1. Traverse the left subtree 2. Visit the root node 3. Traverse the right subtree

This traversal technique is widely used, and in binary search trees, it retrieves data in ascending order.

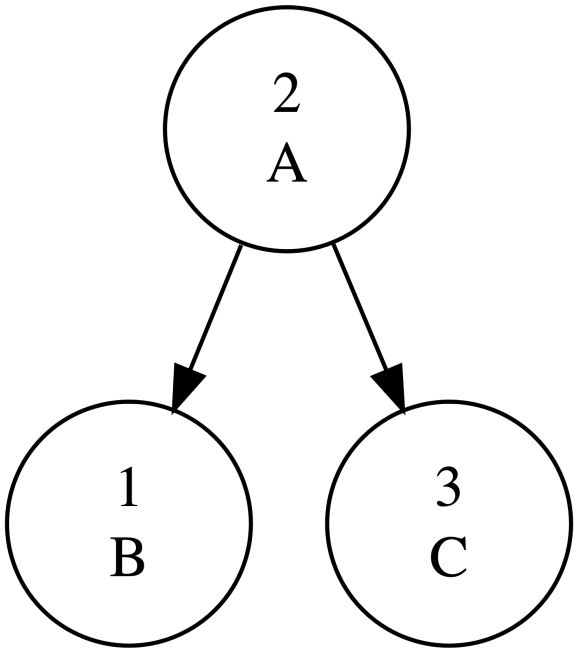


Figure 20.15: In-Order Traversal: Traverse left subtree, Visit root, Traverse right subtree. The numbers indicate the order of traversal.

Pseudocode for in-order traversal can look like this:

```
function inOrderTraversal(node) {
  if (node != null) {
    inOrderTraversal(node.left)
    visit(node)
```

```
    inOrderTraversal(node.right)
  }
}
```

20.4.1.2 Pre-Order Traversal

In pre-order traversal, the process is as follows: 1. Visit the root node 2. Traverse the left subtree 3. Traverse the right subtree

This method can be used to create a copy of the tree, or to get a prefix expression of an expression tree.

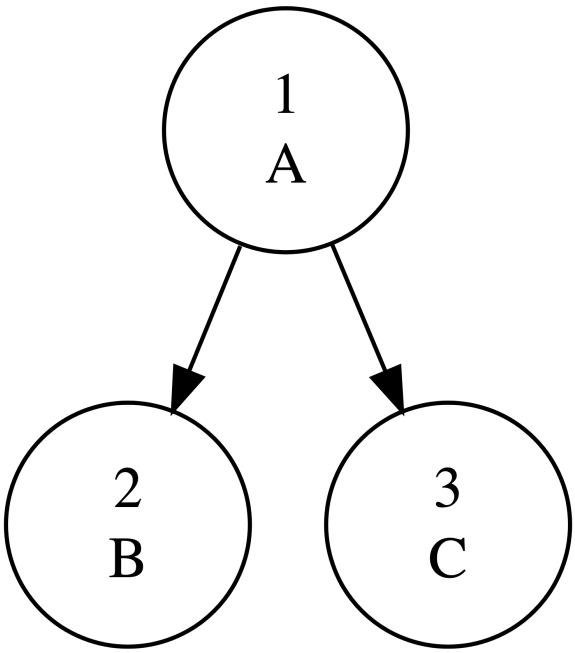


Figure 20.16: Pre-Order Traversal: Visit root, Traverse left subtree, Traverse right subtree. The numbers indicate the order of traversal.

Pseudocode for pre-order traversal can look like this:

```
function preOrderTraversal(node) {
  if (node != null) {
    visit(node)
```

```
    preOrderTraversal(node.left)
    preOrderTraversal(node.right)
  }
}
```

20.4.1.3 Post-Order Traversal

In post-order traversal, the process is as follows: 1. Traverse the left subtree 2. Traverse the right subtree 3. Visit the root node

This method is used to delete or deallocate nodes of a tree, or to get a postfix expression of an expression tree.

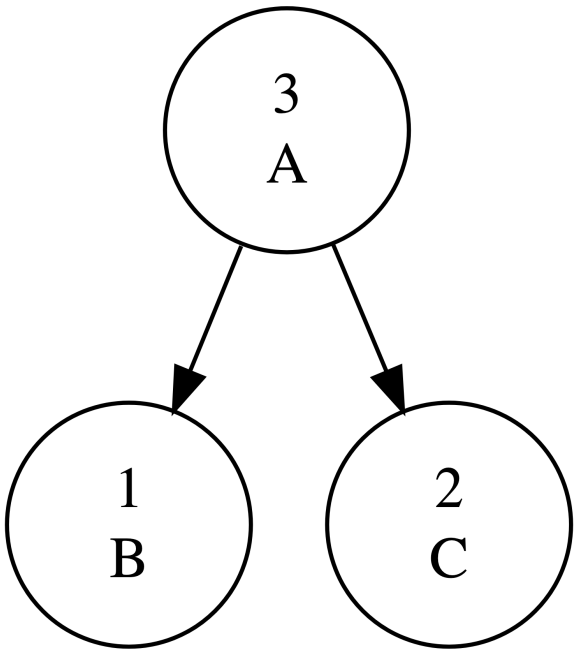


Figure 20.17: Post-Order Traversal: Traverse left subtree, Traverse right subtree, Visit root. The numbers indicate the order of traversal.

Pseudocode for post-order traversal can look like this:

```
function postOrderTraversal(node) {
  if (node != null) {
```

```
    postOrderTraversal(node.left)
    postOrderTraversal(node.right)
    visit(node)
  }
}
```

In each of the traversal methods above, `visit(node)` is an operation that performs some computation on `node`, such as printing its value.

It's important to note that traversal operations have a time complexity of O(n), where n is the number of nodes in the tree. This is because every node must be visited once and only once during the traversal.

In addition to traversal methods, other common operations on binary trees include `insertion`, `deletion`, and `searching`. These are vital operations for maintaining and interacting with the binary tree structure.

20.5 Binary Search Trees (BST)

Before delving into Binary Search Trees (BSTs), let's take a step back and contemplate the significance of search operations in computer science. A search operation is fundamental to many applications, from looking up a contact in your smartphone to searching for a specific book in a library database. Therefore, making search operations efficient has always been a central challenge in computer science.

Traditional search approaches like linear search, which checks each item one by one, can be time-consuming especially when dealing with large datasets, as it has a time complexity of O(n). A considerable improvement is the binary search algorithm, which works by repeatedly dividing the sorted list of elements in half until the target element is found, giving it a time complexity of O(log n). However, binary search requires the data to be sorted, and maintaining a sorted list of elements can be expensive for insert and delete operations.

This is where Binary Search Trees come in. A Binary Search Tree is a special type of binary tree that maintains a specific ordering property — for each node, all elements in the left subtree are less than the node, and all elements in the right subtree are greater than the node (see Figure 20.18). This property of BSTs enables us to perform search, insert, and delete operations efficiently, maintaining a time complexity of O(log n) in an ideal situation where the tree is balanced. Let's explore these concepts in more detail.

20.5.1 Definition, Properties, and Structure of BSTs

A Binary Search Tree (BST) is a binary tree where for every node:

- The values in its left subtree are less than the node's value.
- The values in its right subtree are greater than the node's value.

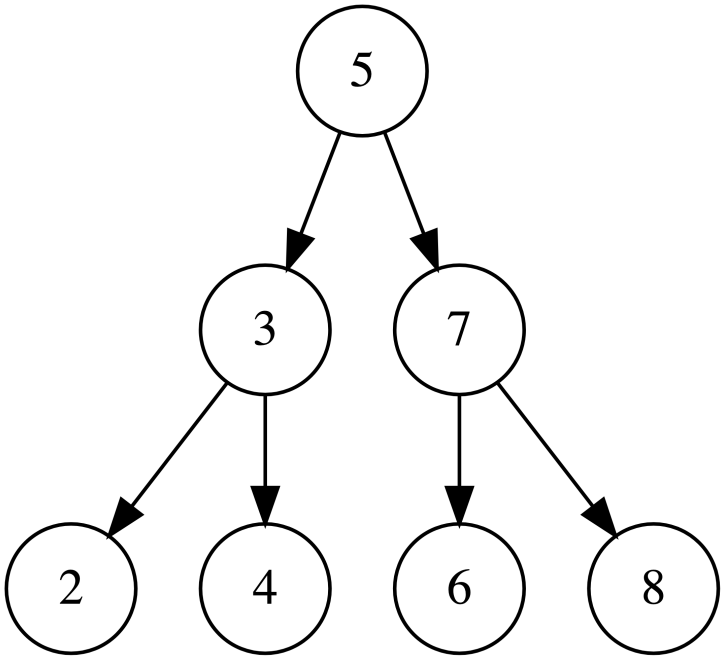


Figure 20.18: The Binary Search Tree property: for every node, all elements in the left subtree are less than the node, and all elements in the right subtree are greater than the node.

This property ensures that the "in-order" traversal of a BST results in a sorted sequence of values, which is the key to efficient search, insertion, and deletion operations.

20.5.2 Advantages of BSTs over other tree structures

The BST property provides some advantages over other tree structures:

- **Efficient search**: If the tree is balanced, we can find, insert, or delete entries in O(log n) time where n is the total number of nodes. This is significantly faster than linear search, and it doesn't require the maintenance of a sorted list like binary search.
- **Sorted traversal**: In-order traversal of a BST yields the nodes in sorted order, which can be useful in various applications.
- **Flexibility**: Unlike an array, BSTs do not have a fixed size. They can grow and shrink during the execution of the program, allowing efficient use of memory.

20.5.3 Detailed description of BST operations: insertion, deletion, searching

BSTs support three fundamental operations: search, insertion, and deletion, all of which leverage the BST property to operate efficiently.

Search: Starting from the root, if the target value is less than the current node, we move to the left child; if it's greater, we move to the right child. We repeat this process until we either find the target value or reach a null where the target value should be if it were in the tree (see Figure 20.19).

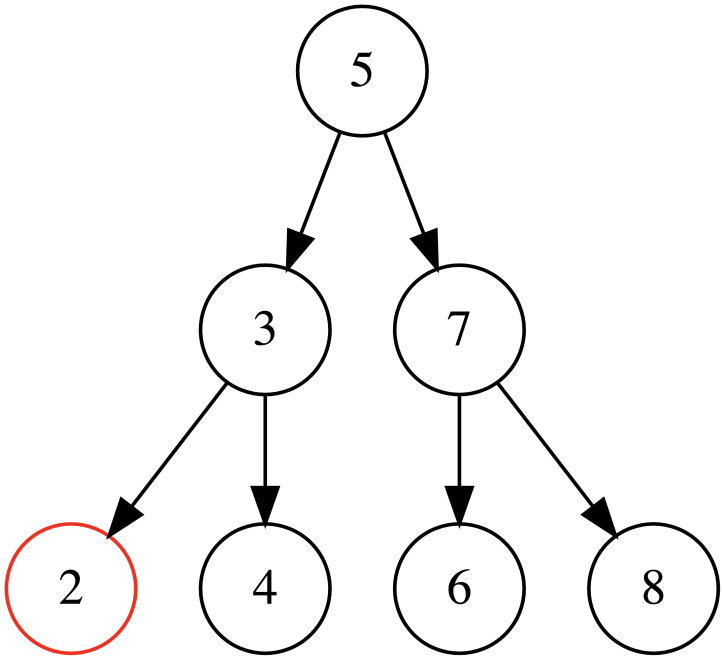


Figure 20.19: BST search operation: Starting from the root, the algorithm compares the target value with the current node and moves left or right based on the comparison. The search ends when the target is found or when a null is encountered.

Insertion: Similar to search, but when we reach a null location, we create a new node with the target value at that spot (see Figure 20.20).

Deletion: A bit more complicated as we need to maintain the BST property after removing a node. The process varies based on the number of children the node to be deleted has

, similar to deletion in a binary tree (see Figure 20.21).

Absolutely. Let's break down the implementation of BST operations into both recursive and iterative approaches using Java.

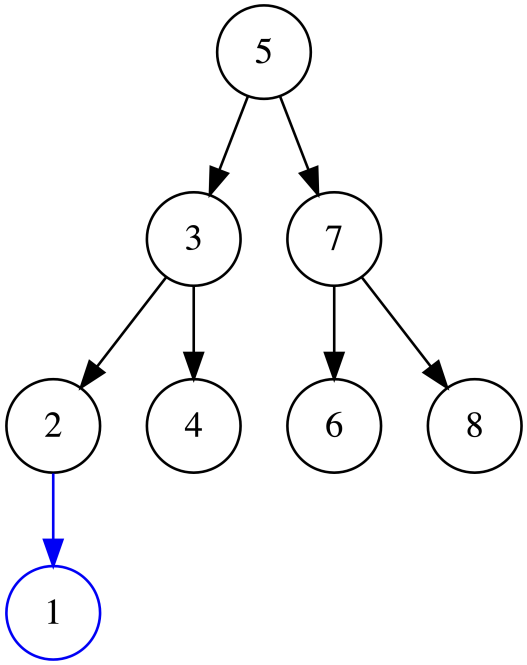


Figure 20.20: BST insertion operation: Similar to the search operation, the algorithm moves down the tree starting from the root. When a null location is encountered, a new node is created at that spot.

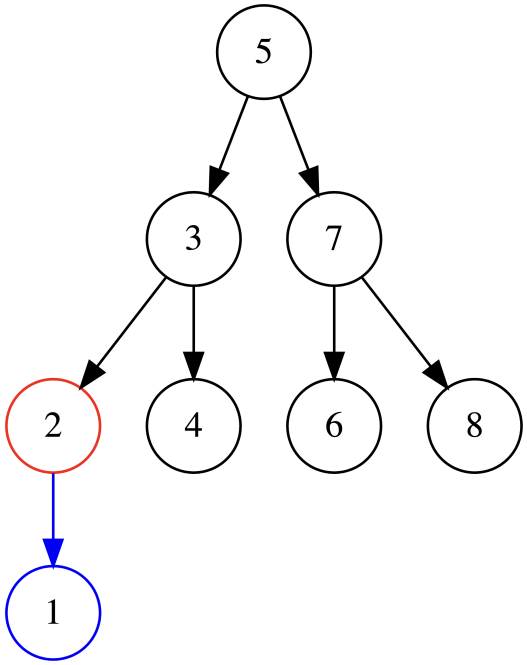


Figure 20.21: BST deletion operation: Deletion in a BST can be complex as it must maintain the BST property after removing a node. The process varies depending on whether the node to be deleted has no children, one child, or two children.

20.5.4 Recursive and Iterative Implementations in Java

While recursion offers a straightforward and elegant approach to implementing BST operations, the iterative method can often provide better performance in terms of space complexity. Let's look at both methods for each of the main BST operations: search, insertion, and deletion.

20.5.5 Search Operation

The objective of the search operation in a binary search tree (BST) is to find a node that contains a specified value. If the value is present in the BST, the function will return that node. Otherwise, it will return null.

To find a node in a BST, we can use one of two main approaches: recursion or iteration. Both have their advantages and disadvantages, which will depend on the situation at hand.

20.5.5.1 Recursive Approach

Consider the following Java code that uses a recursive method to search a node in a BST.

```
public TreeNode search(TreeNode node, int value) {
    if (node == null || node.val == value)
        return node;

    if (node.val > value)
        return search(node.left, value);

    return search(node.right, value);
}
```

The method `search` receives a TreeNode `node` (that represents a current node in the BST) and an integer `value` (that is the value to be searched).

The search operation begins by checking whether the node is null (which indicates that we've traversed a path in the tree where the value does not exist) or the value of the current node is equal to the desired value. If either condition is true, the node is returned.

If the value to be found is less than the current node's value, the function recursively calls itself, continuing the search on the left subtree (`node.left`).

Conversely, if the value is greater than the current node's value, the function calls itself to continue the search on the right subtree (`node.right`).

20.5.5.2 Iterative Approach

Now, let's look at an iterative approach to search a node in a BST. The iterative approach can be more space-efficient as it does not require stack space for recursive calls.

```
public TreeNode search(TreeNode root, int value) {
    while (root != null) {
        if (root.val > value)
            root = root.left;
        else if (root.val < value)
            root = root.right;
        else
            return root;
    }
    return null;
}
```

In the iterative approach, we begin with the root node and enter a while loop that continues as long as the current node is not null.

If the value of the current node (`root.val`) is greater than the desired value, we move to the left child of the current node (`root = root.left`).

If the value of the current node is less than the desired value, we move to the right child (`root = root.right`).

The loop breaks and the current node is returned if the value of the current node is equal to the desired value.

If the value is not found in the tree, the function will return null.

20.5.5.3 Visualizing the Search Operation

We'll use a simple binary search tree with the values 2, 1, and 3, with 2 being the root node, 1 the left child, and 3 the right child. We'll search for the value 3.

Here, the search operation starts at node 2 (the root node). In both recursive and iterative approaches, the search function compares the value to be found (3) with the current node's value (2). As 3 is greater than 2, the function moves to the right child (node 3) for both recursive and iterative approaches. This search path is highlighted in blue for recursive and green for iterative.

The search function now finds that the value of the current node (3) is equal to the desired value (3). Therefore, the function returns this node.

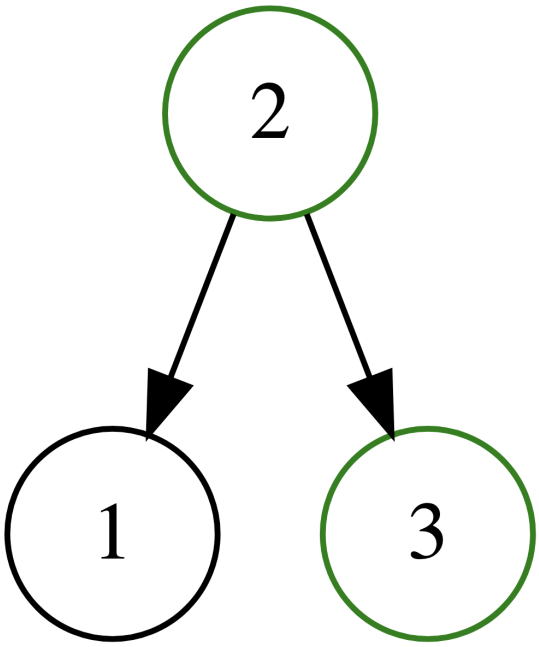


Figure 20.22: Illustration of the search operation process in a binary search tree, comparing recursive and iterative approaches. The recursive path is shown in blue, while the iterative path is shown in green.

Please note that in this simple example, the recursive and iterative paths are identical. This won't always be the case, especially in larger trees, but the key takeaway is that both methods are following the same fundamental logic, albeit through different operational mechanics.

20.5.6 Insertion Operation

Insertion operation in a Binary Search Tree (BST) aims to add a new node with a specific value to the tree. The position of the new node is determined by the fundamental property of BST: for any given node, all nodes in its left subtree have values less than its value, and all nodes in its right subtree have values greater than its value.

20.5.6.1 Recursive Approach

Let's start with the recursive approach to insert a node in a BST. Here is a snippet of Java code implementing this approach:

```
public TreeNode insert(TreeNode node, int value) {
    if (node == null) {
        return new TreeNode(value);
    }

    if (value < node.val) {
        node.left = insert(node.left, value);
    } else if (value > node.val) {
        node.right = insert(node.right, value);
    }

    return node;
}
```

This `insert` function takes a TreeNode `node` (representing the current node in the BST) and an integer `value` (the value to be inserted).

The function first checks if the node is null. If it is (indicating that we've reached an empty spot in the tree where a new node should be inserted), it creates a new TreeNode with the given value and returns it.

If the node is not null, it checks if the value to be inserted is less than the value of the current node. If so, it recursively calls the `insert` function to insert the value into the left subtree (`node.left`).

If the value to be inserted is greater than the value of the current node, it recursively calls `insert` to insert the value into the right subtree (`node.right`).

After the value is inserted, it returns the (potentially modified) current node, maintaining the structure of the tree.

20.5.6.2 Iterative Approach

Now let's consider an iterative approach to insert a node in a BST, which can be more space-efficient as it doesn't require stack space for recursive calls:

```
public TreeNode insert(TreeNode root, int value) {
    TreeNode node = new TreeNode(value);
    if (root == null) {
        return node;
    }

    TreeNode parent = null, current = root;
    while (current != null) {
        parent = current;
        if (current.val > value) {
            current = current.left;
        } else {
            current = current.right;
        }
    }

    if (parent.val > value) {
        parent.left = node;
    } else {
        parent.right = node;
    }

    return root;
}
```

In the iterative `insert` function, we start by creating a new TreeNode `node` with the specified value.

If the BST is empty (i.e., the root is null), we return this new node as the root of the BST.

If the BST is not empty, we start from the root and iterate down the tree to find the correct position for the new node.

We maintain two pointers during the traversal: `parent` (representing the parent of the current node) and `current` (the current node in the traversal). We update `parent` to be `current` before moving `current` in each iteration.

When the `current` node becomes null, we've found the correct spot for the new node. If the new value is less than the parent's value, we insert the new node as the left child of the parent; otherwise, we insert it as the right child.

The function then returns the root of the (now modified) BST.

20.5.6.3 Visualizing the Insertion Operation

Consider a simple binary search tree with values 2, 1, and 3, where 2 is the root node, 1 is the left child, and 3 is the right child. Let's try to insert a new node with the value 4 into this tree.

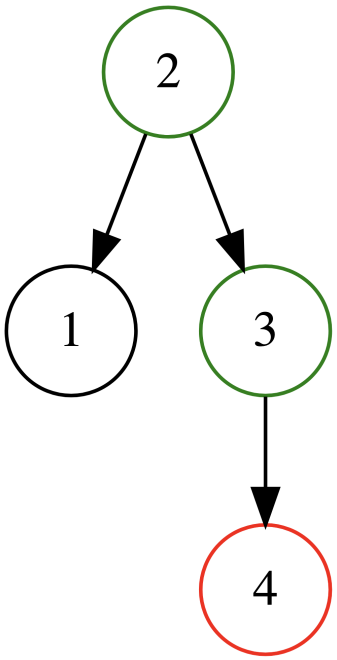


Figure 20.23: Illustration of the node insertion process in a binary search tree, showcasing both recursive and iterative approaches. The path followed by the recursive approach is shown in blue, while the path followed by the iterative approach is shown in green.

In this diagram, we start at node 2 (the root node) and want to insert a node with the value 4 (shown in red). In both the recursive and iterative methods, we start by comparing the value to be inserted (4) with the value of the current node (2). Since 4 is greater than 2, we move to the right child (node 3).

We then compare 4 with 3. Since 4 is greater than 3, and node 3 doesn't have a right child, we add the new node as the right child of node 3. The paths taken by the recursive and iterative approaches are shown in blue and green respectively.

Just like the search operation, in this simple example, the recursive and iterative paths for the insertion operation are identical. But in a larger tree, the paths may differ depending on the current state of the tree and the value to be inserted. However, both methods adhere to the same principle - the new node is added in such a way that it maintains the property of the binary search tree.

20.5.7 Deletion Operation

Deleting or removing a node from a binary tree is a slightly complex operation compared to insertion. This is because when we remove a node, we need to ensure that the remaining nodes still form a binary tree. The process differs depending on the number of children the node has:

1. **No child**: If the node is a leaf node (i.e., it does not have any children), we can directly remove the node.

2. **One child**: If the node has a single child, we can replace the node with its subtree.

3. **Two children**: This is the most complex case. If the node has two children, we generally seek a node from the tree to replace the node to be deleted. The replacement node is typically the in-order predecessor or the in-order successor of the node. These are the node's immediate previous and next nodes in in-order traversal (see Figure 20.15).

    - **In-order predecessor**: This is the node with the highest value that is smaller than the value of the node to be removed. It can be found by moving one step to the left (to the left child), and then as far right as possible.
    - **In-order successor**: This is the node with the smallest value that is larger than the value of the node to be removed. It can be found by moving one step to the right (to the right child), and then as far left as possible.

Once we find the in-order predecessor (or successor), we replace the node to be deleted with the predecessor (or successor) and then recursively delete the predecessor (or successor) from its original position.

The reason we choose the in-order predecessor or successor is to maintain the binary search tree property, i.e., for every node, nodes in the left subtree have values less than the node, and nodes in the right subtree have values greater than the node.

Here is a basic outline of the removal process in pseudocode:

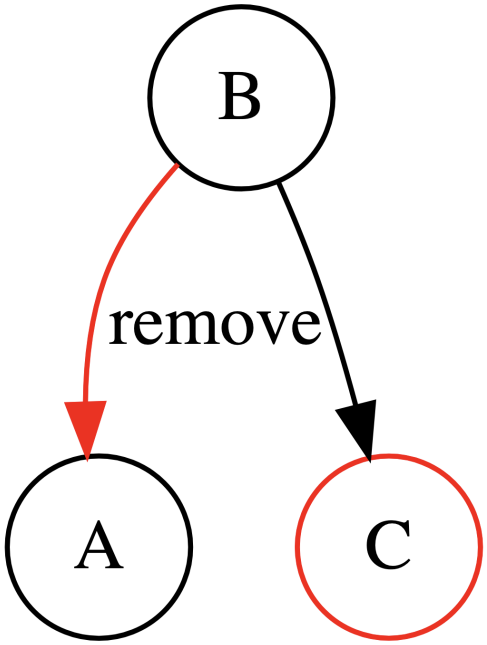


Figure 20.24: Removing a node with no children. The node ‘C’ is simply removed from the tree.

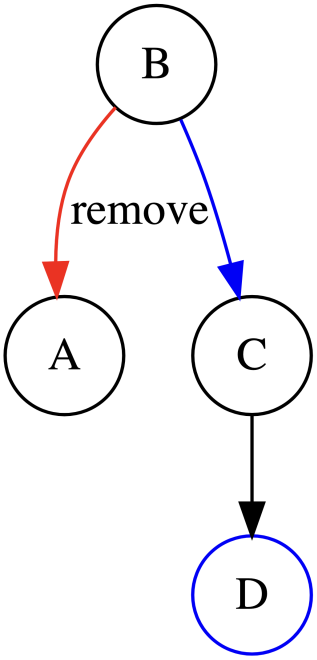


Figure 20.25: Removing a node with one child. The node ‘B’ is replaced by its subtree rooted at ‘C’.

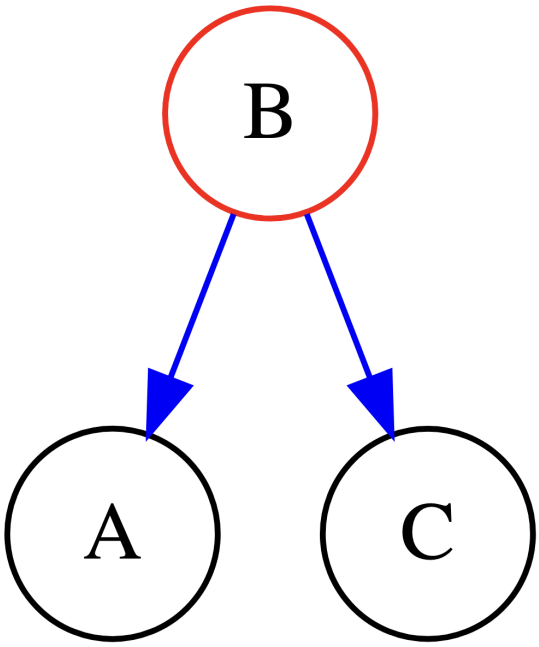


Figure 20.26: Removing a node with two children. The node ‘B’ is replaced by its in-order predecessor ‘A’ or in-order successor ‘C’. The replacement node is then recursively removed from its original position.

```
function remove(node, key) {
  if (node is null) {
    return node
  }
  if (key < node.key) {
    node.left = remove(node.left, key)
  } else if (key > node.key) {
    node.right = remove(node.right, key)
  } else {
    if (node.left is null) {
      return node.right
    } else if (node.right is null) {
      return node.left
    }
    node.key = minValue(node.right)
    node.right = remove(node.right, node.key)
  }
  return node
}

function minValue(node) {
  current = node
  while (current.left is not null) {
    current = current.left
  }
  return current.key
}
```

In this pseudocode, `remove()` is a recursive function that deletes the node with the specified key from the tree and returns the new root of the subtree, and `minValue()` is a helper function that returns the minimum value in a non-empty binary search tree.

The time complexity for deletion in a binary tree can range from O(log n) to O(n), where n is the number of nodes. In a balanced tree, the time complexity is O(log n) because we would be traversing the height of the tree, which is logarithmic in a balanced tree. However, in the worst-case scenario, such as when the tree is a degenerate or skewed tree, the time complexity would be O(n) because each removal could potentially involve traversing all the nodes of the tree.

20.5.7.1 Recursive Approach

```
public TreeNode delete(TreeNode root, int value) {
    if (root == null) {
        return root;
    }

    if (value < root.val) {
        root.left = delete(root.left, value);
    } else if (value > root.val) {
        root.right = delete(root.right, value);
    } else {
        if (root.left == null)
            return root.right;
        else if (root.right == null)
            return root.left;

        root.val = findMinValue(root.right);
        root.right = delete(root.right, root.val);
    }

    return root;
}

private int findMinValue(TreeNode root) {
    int min = root.val;
    while (root.left != null) {
        min = root.left.val;
        root = root.left;
    }
    return min;
}
```

Please note that the iterative implementation of delete operation is relatively complex and can detract from the understanding of how the delete operation fundamentally works. Therefore, we generally prefer the recursive approach for teaching purposes. However, once you are comfortable with the recursive implementation, it's worthwhile to try to implement the iterative version on your own for practice.

The presented examples illustrate how different BST operations can be performed in both recursive and iterative manner. Each has its own pros and cons - recursive methods are often easier to comprehend and write, whereas iterative methods might provide better performance.

20.5.8 Performance and Time Complexity

The time complexity of binary search tree operations such as search, insert, and delete depends on the height of the binary search tree. In the best-case scenario, the tree is perfectly balanced, and its height is log(n), where n is the number of nodes. In this case, search, insert, and delete operations can be performed in O(log n) time. However, in the worst-case scenario, the tree can become skewed, which means it resembles a linked list more than a tree. In this case, the height of the tree is n, and operations can take up to O(n) time.

Let's see how this plays out for each operation:

Search Operation

- Best-case performance: O(log n) – This is when the tree is balanced, and we can eliminate half of the nodes from our search at each step.
- Worst-case performance: O(n) – This happens when the tree is skewed, and our search path includes most or all nodes in the tree.

Insert Operation

- Best-case performance: O(log n) – When the tree is balanced, the location for a new node is found by traversing from the root to the appropriate leaf node.
- Worst-case performance: O(n) – In a skewed tree, the new node could end up being a child of the deepest leaf node.

Delete Operation

- Best-case performance: O(log n) – In a balanced tree, we find the node to delete quickly. If the node has two children, we also need time to find the in-order predecessor or successor.
- Worst-case performance: O(n) – In a skewed tree, deletion can involve traversing most of the tree.

One thing to note is that the best-case scenario, a balanced tree, is not the average case. Unless measures are taken to balance the tree, binary search trees can become skewed from sequences of insertions and deletions.

The next topic that we'll discuss, self-balancing binary search trees, will address this limitation of the basic binary search tree by ensuring that the tree remains approximately balanced at all times. As a result, self-balancing binary search trees can guarantee that the time complexity of major operations is always O(log n), which is a significant improvement in the worst-case scenario.

20.6 Balanced Binary Search Trees

Binary search trees (BSTs) are powerful data structures that support efficient search, insertion, and deletion operations. However, if the tree becomes unbalanced, these operations could degrade to linear time complexity. One effective way to mitigate this issue is by employing balanced binary search trees, such as AVL Trees and Red-Black Trees. These trees aim to maintain balance, ensuring efficient performance regardless of the input sequence.

The concept of a "balanced" tree might sound abstract, so let's make it tangible with an example (Figure 20.27). A balanced binary tree is one where the difference between the heights of the left and right subtrees of every node is either -1, 0, or 1. This balance ensures that the tree doesn't lean too heavily on one side, optimizing the path to every node and promoting efficiency.

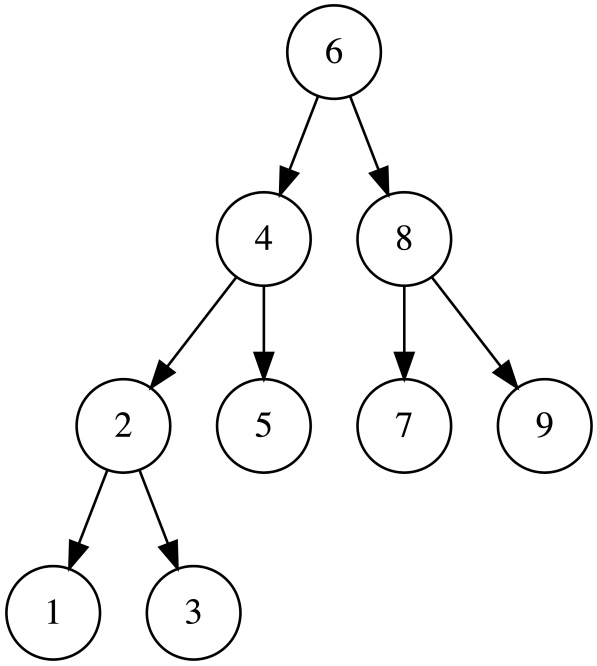


Figure 20.27: A balanced binary search tree. The numbers show the height of each node.

Consider a balance scale, with nodes as the weights. When equally balanced, the scale maintains equilibrium. However, adding or subtracting weights (or nodes) disturbs this balance, causing the scale to lean towards the heavier side. AVL Trees and Red-Black Trees work similarly - they maintain equilibrium by redistributing the weights, which we call "rotations" in the context of trees.

AVL Trees, named after their inventors Adelson-Velsky and Landis, adjust their balance through a process called rotation. This operation preserves the in-order property of the tree. If the balance factor of a node in an AVL tree becomes 2 or -2, a rotation is performed to regain balance. Let's illustrate this with a simple example (Figure 20.31).

Consider inserting 1, 2, 3 into an initially empty AVL tree.

1. Inserting 1 gives us:

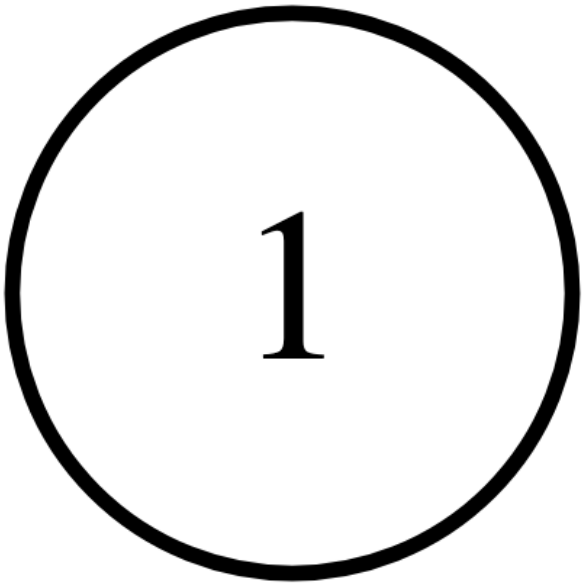


Figure 20.28: Inserting 1 into an empty AVL tree.

2. Inserting 2 gives us:

3. Inserting 3 unbalances the tree:

This structure is unbalanced and equivalent to a linked list. The balance factor for node 1 is -2, indicating a required rotation.

4. We perform a left rotation around the root (node 1), resulting in a balanced tree:

The AVL Tree is now balanced, and all operations are guaranteed logarithmic time complexity.

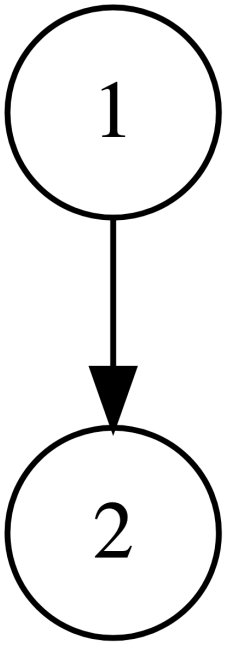


Figure 20.29: Inserting 2 into the AVL tree.

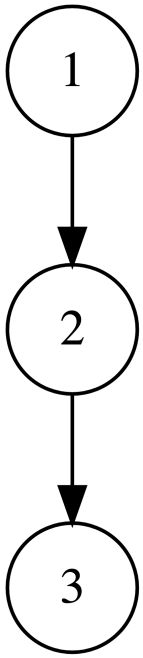


Figure 20.30: Inserting 3 into the AVL tree results in an unbalanced tree.

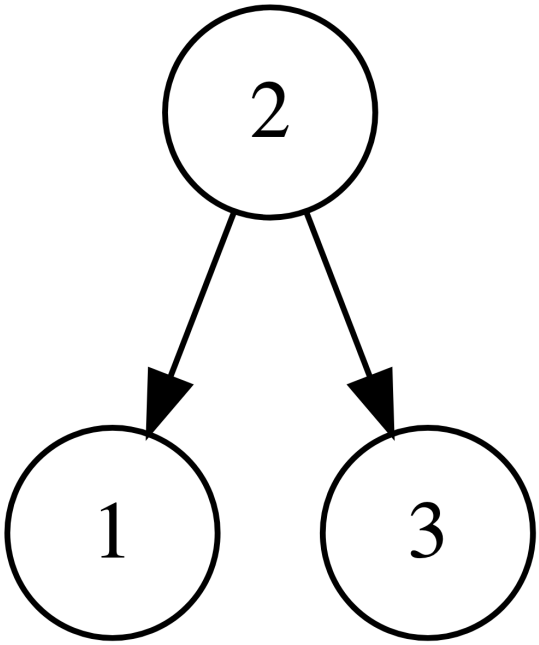


Figure 20.31: A left rotation around the root balances the AVL tree.

Red-Black Trees offer another solution, painting each node one of two colors – red or black – and maintaining balance by adhering to red-black properties. Although the rules and operations are more complex, they also ensure the tree remains approximately balanced during additions and deletions. A Red-Black Tree would be great to visualize, but given its complexity, it's better to explore it interactively or in more advanced courses.

These tree structures may seem complicated, but they highlight how well-understood tools like a binary tree can be further optimized for efficiency. While we've only scratched the surface here, you'll delve deeper into these and other types of self-balancing trees in advanced data structures courses. The goal is not to memorize every detail but to appreciate the breadth of tools available for different use cases.

In your journey as a computer scientist, you'll encounter various real-world scenarios where maintaining efficiency is vital. From databases to file systems, different applications require different tools. Understanding the principles behind these tools, such as the importance of balance in BSTs, will empower you to make informed decisions in your work.

With this, we conclude our journey into the world of binary trees and binary search trees. We hope that the concepts, examples, and code shared in this chapter help illuminate these fundamental data structures. Remember that learning is a process - with every step, you're building a stronger foundation in computer science. Keep practicing, keep questioning, and keep exploring.

Part IX

Graphs

21 Graphs

Consider a typical day scrolling through Instagram. You tap on a photo from a friend, then another from an influencer you admire, followed by a click on the profile of someone who commented, leading you to yet another profile you find interesting. This scenario illustrates the complex web of connections that is characteristic of social networks. In order to analyze and make sense of such connections, we need an appropriate data structure to model them.

Take, for instance, your Instagram followers. We could start by simply representing you and your followers as entities, or "vertices," connected by lines or "edges." The diagram below (Figure 21.1) shows such a simple representation of your followers.

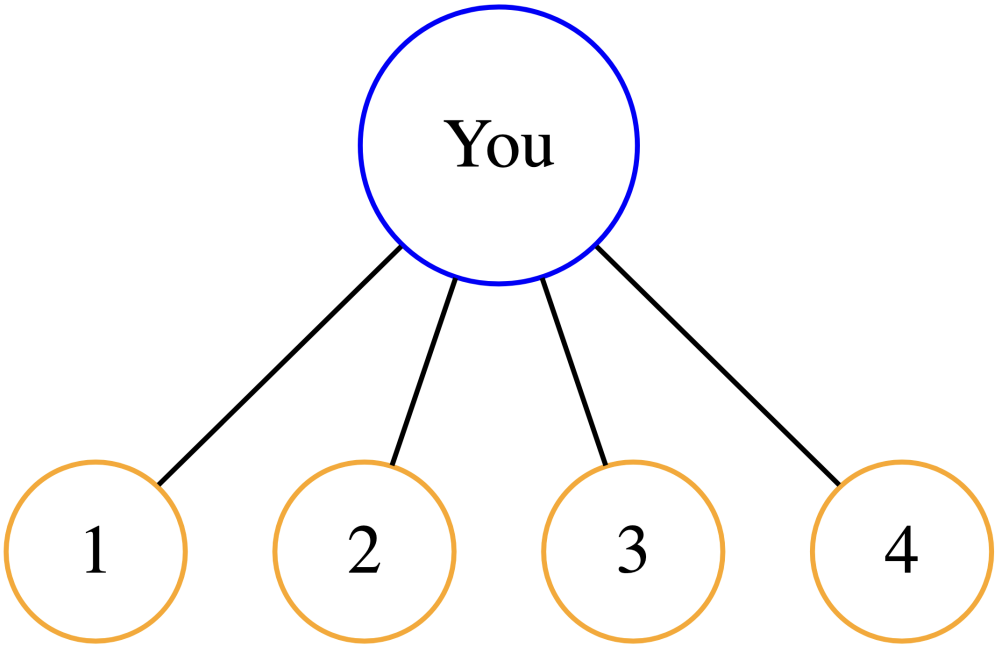


Figure 21.1: A simplified representation of your Instagram followers. You (Y, colored in blue) are connected to four followers (1-4, colored in orange). Each line represents a connection (follows).

At first glance, this might appear to be a tree structure, where you are the root, and your followers are the branches. However, this isn't the most accurate representation. In reality, your followers can also follow other people, and those individuals can have their own followers, creating a multi-tiered and mutual relationship that cannot be represented using a tree structure. Tree structures are fundamentally hierarchical, with each node having one parent at most, making them inadequate to model the complex interconnections found in social networks (Figure 21.2).

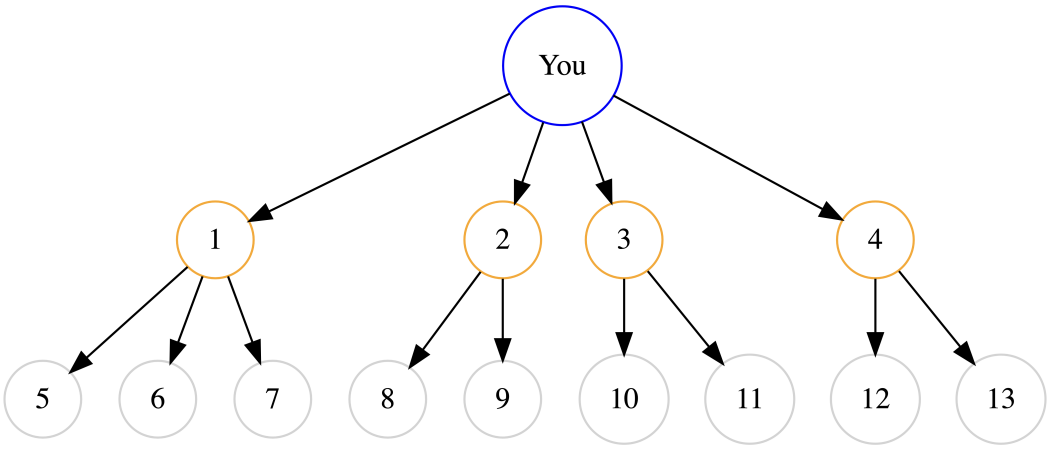


Figure 21.2: A representation of Instagram followers with multiple levels of connections. Each follower can have their own followers, creating a network of connections. Arrows indicate the direction of following, showing that the relationship is not always reciprocal.

To accurately capture these complex relationships, we use a graph data structure. Graphs consist of a set of vertices (or nodes) and a set of edges that connect them. In the context of Instagram followers, the vertices represent the users, and the edges represent the connections between them.

Furthermore, the relationships between Instagram users are not always reciprocal. You might follow a celebrity who does not follow you back. This creates a directed relationship, represented by an edge with a specific direction. Such relationships can be modeled using directed graphs. Figure 21.3 shows a graph where the edges represent the direction of follower relationships.

Understanding and modeling such intricate networks is critical to analyzing social media behavior. From detecting trends and predicting future connections to understanding influence within a network, the applications of this model are vast and significant. In the subsequent sections, we will delve deeper into how graph data structures work, how to implement them, and how to analyze the interesting problems they can help solve.

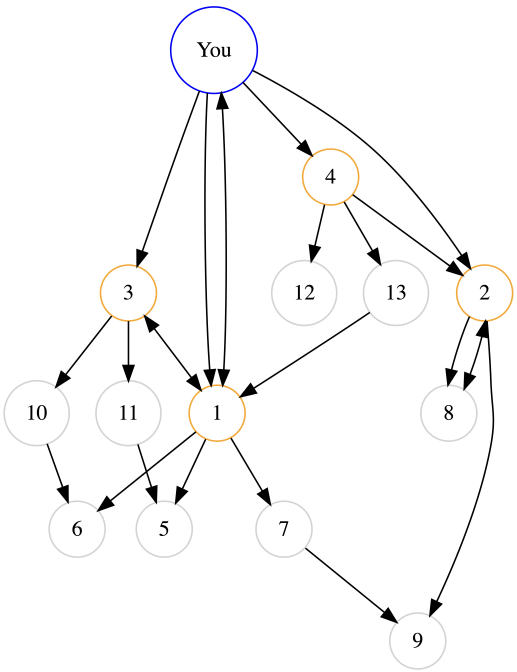


Figure 21.3: A directed representation of your Instagram followers. Here, an arrow going from vertex A to vertex B indicates that A follows B, but B does not necessarily follow A. The users are color-coded: blue for you, orange for direct followers, and light grey for followers of followers. Note that the graph reflects mutual follows, unreciprocated follows, and connections among followers, which a tree structure would not capture.

21.1 Introduction

Imagine standing at a subway station, tracing the colorful lines on the map that depict the various routes. Each line represents a different subway route, while each station serves as a meeting point for different lines. The interplay of lines and stations illustrates the essence of a **graph** - a non-linear data structure that consists of vertices (stations) and edges (subway routes) connecting these vertices.

Adjacency and **incidence** are two fundamental terms in the realm of graphs. Two vertices, say station A and station B, are considered adjacent if a subway line (edge) directly connects them. On the other hand, incidence refers to a station (vertex) being served by a subway line (edge).

Graphs are not merely abstract mathematical constructs but have tangible real-world applications. They model social networks, where vertices are people, and edges signify their connections. In transportation networks, vertices represent locations interconnected by edges symbolizing routes. In the era of a global pandemic, graphs can even represent Coronavirus transmission networks, where vertices symbolize individuals, and edges depict transmission paths.

21.2 Graph Terminology

To navigate through the expansive world of graphs, one must be acquainted with its language. Let's traverse through the fundamental terms and properties that define graphs.

21.2.1 Basic Terms and Properties

Imagine a city map representing connections between various landmarks - each landmark is a **vertex**, and the roads linking them are the **edges**. The connections established are not always two-way streets. While some roads may be traversed in both directions, symbolizing an **undirected graph**, others might be one-way streets, indicative of a **directed graph**. Here, edges have a specific direction, indicating a unidirectional relationship between vertices. The diagrams Figure 21.4 and Figure 21.5 vividly illustrate this concept.

In some cities, not all roads are created equal. Some roads might be longer or more congested, carrying different costs for traversal. This scenario reflects a **weighted graph** where edges are not equally important but carry different weights. On the contrary, in an **unweighted graph**, all roads carry the same weight - a short stroll in the park or a long cross-country journey holds equal significance. The illustrations Figure 21.6 and Figure 21.7 demonstrate this.

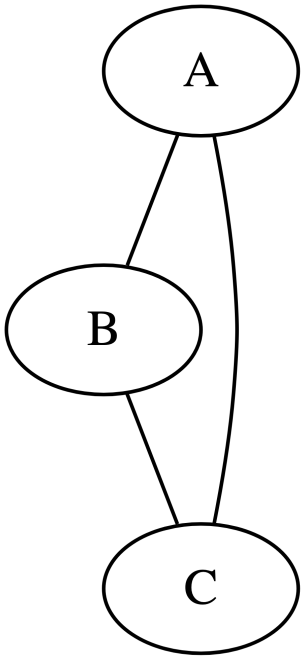


Figure 21.4: Example of an undirected graph. Vertices A, B, and C are interconnected, demonstrating bi-directional relationships.

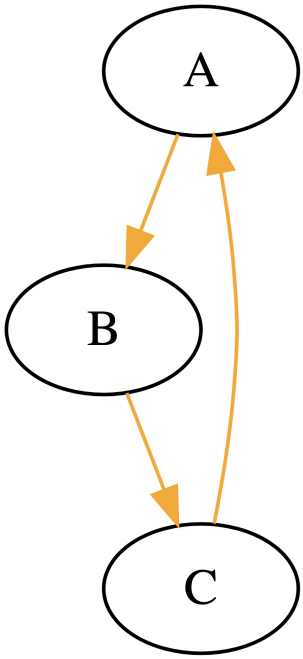


Figure 21.5: Example of a directed graph. The arrowheads (highlighted in orange) demonstrate the direction of the relationships between vertices A, B, and C.

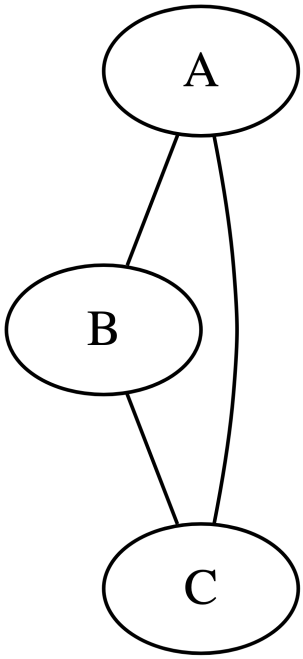


Figure 21.6: Example of an unweighted graph. The edges between vertices A, B, and C carry no weights.

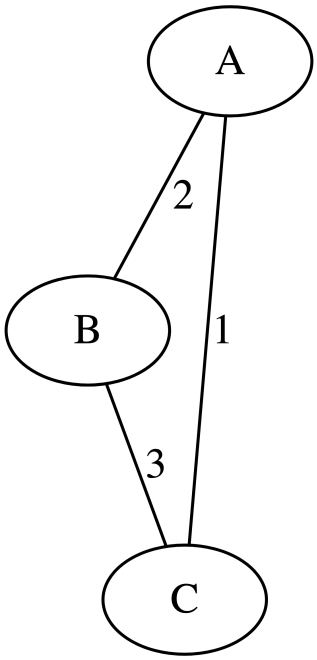


Figure 21.7: Example of a weighted graph. The weights (2, 3, 1) assigned to the edges between vertices A, B, and C differentiate their importance or cost.

As we traverse through the city, we encounter different traffic configurations. Some intersections allow only a single connection between landmarks, reflecting a **simple graph** with no self-loops. However, other intersections might have multiple routes connecting the same pair of landmarks or even allow U-turns, creating self-loops. This scenario portrays a **multigraph**. The diagrams Figure 21.8 and Figure 21.9 encapsulate these ideas.

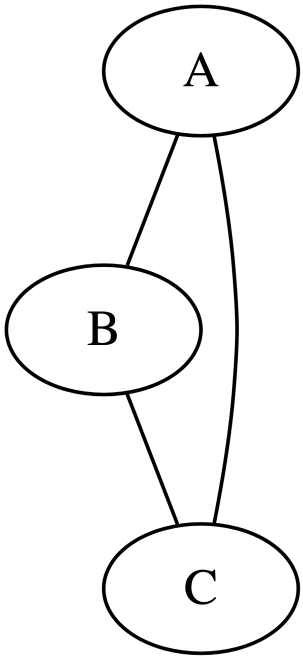


Figure 21.8: Example of a simple graph. Vertices A, B, and C are interconnected by simple edges.

In the city of graphs, the popularity of a landmark (vertex) might be judged by the number of roads (edges) leading to it, a concept known as the **degree** of a vertex. A busy intersection with roads pouring in and out signifies a high-degree vertex. (See Figure 21.10 for an example.)

To travel from one landmark to another, one must choose a **path**, a sequence of landmarks connected by roads. Occasionally, this path might lead back to the starting point, creating a **cycle**, a closed path where the journey starts and ends at the same landmark. (See Figure 21.11 and Figure 21.12 for examples.)

Cities can be busy, bustling with interconnected landmarks, or they can be quiet towns with isolated points of interest. A city with a path between every pair of landmarks is a **connected graph**, while a city with isolated landmarks, unreachable from others, is a **disconnected graph**. This concept is illustrated in Figure 21.13 and Figure 21.14.

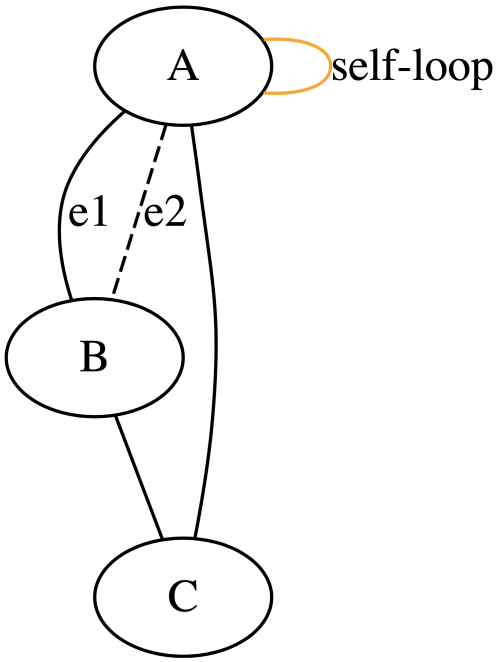


Figure 21.9: Example of a multigraph. Vertex A is connected to vertex B through two edges (a solid and a dashed line) and to itself via a self-loop (in orange).

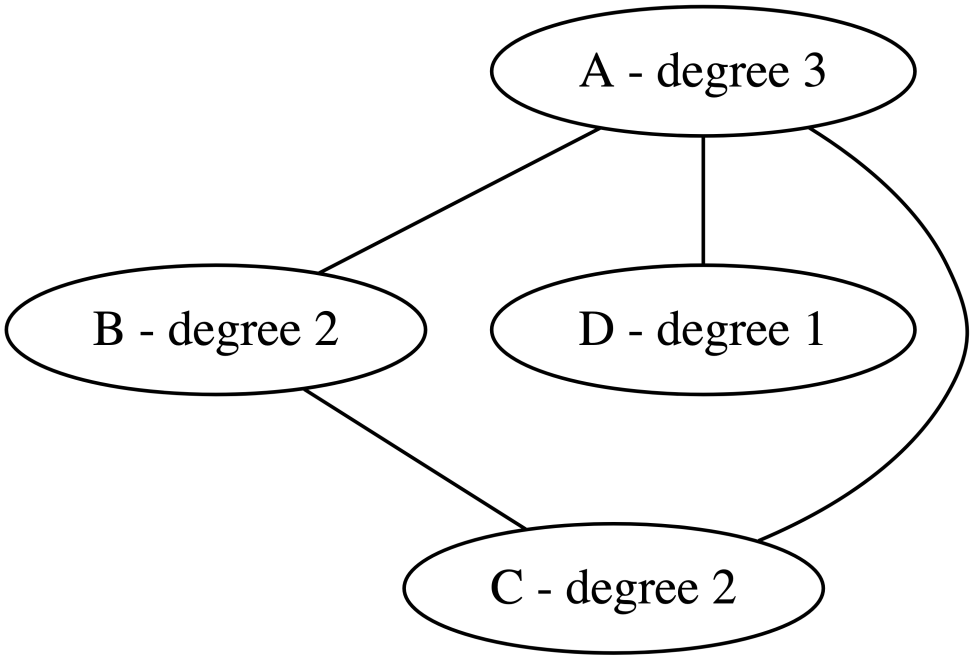


Figure 21.10: Example graph with vertex degrees. The degree of each vertex is indicated by the number in parentheses in the labels on the edges.

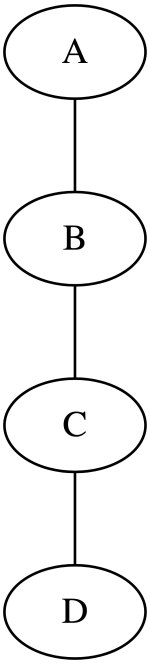


Figure 21.11: Example graph with a path from A to D.

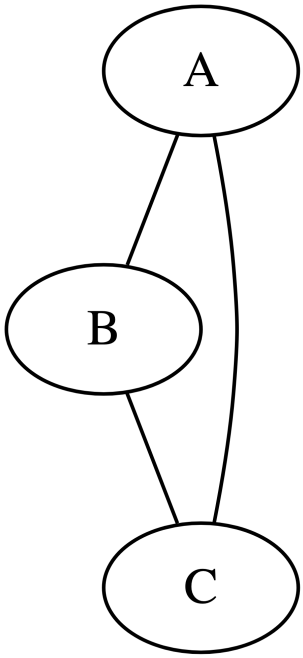


Figure 21.12: Example graph with a cycle. (A-B-C)

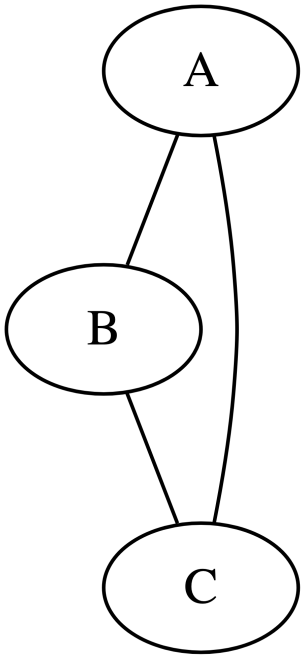


Figure 21.13: Example of a connected graph.

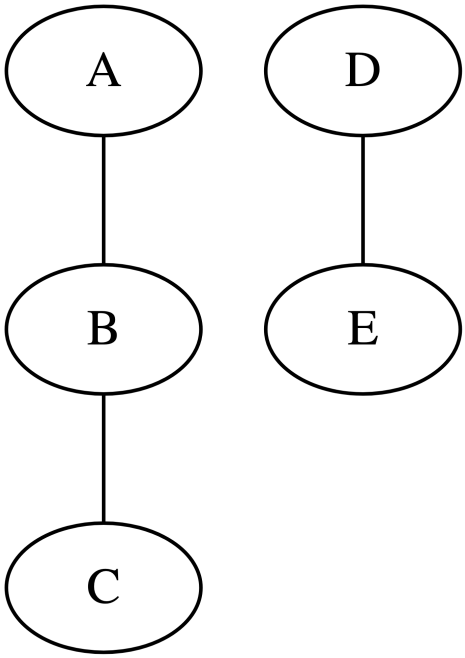


Figure 21.14: Example of a disconnected graph.

Understanding these terms and properties provides the vocabulary to talk about graphs, setting the stage for exploring more advanced concepts and applications. As we delve deeper, we will start seeing how these properties interplay and shape the complex world of graphs.

21.2.2 Graph Notation

In mathematical terms, a graph can be represented by the notation $G(V, E)$, where G stands for the graph, V is the set of vertices (or nodes), and E is the set of edges (or connections). This notation provides a concise, abstract representation of the graph, which aids in understanding and analyzing its structure.

21.2.3 Special Types of Graphs

Certain types of graphs, distinguished by specific properties or structures, have earned unique classifications. Three such types are: complete graphs, bipartite graphs, and trees.

- **Complete Graph**: A complete graph, denoted by K_n where n is the number of vertices, is a simple graph where every pair of distinct vertices is connected by a unique edge. In a complete graph, each vertex is directly connected to every other vertex, signifying a fully interconnected network. This type of graph is commonly seen in problems concerning network connectivity and graph coloring. See Figure 21.15.

- **Bipartite Graph**: A bipartite graph can be partitioned into two distinct sets, where vertices within the same set share no edges. Each edge connects a vertex from one set to a vertex in the other set, and never two vertices within the same set. This special class of graphs has extensive applications, including matching problems, scheduling problems, and in the study of molecular structures. See Figure 21.16.

- **Tree**: A tree is a special type of graph that holds a significant place in computer science due to its simplicity and versatility. A tree is an undirected, connected graph with no cycles. Often, one vertex is designated as the root, and the other vertices are arranged in a parent-child relationship, forming a hierarchical structure. Trees are utilized in various areas such as organizing hierarchical data, structuring web pages, and in algorithms like searches and sorts. See Figure 21.17.

21.3 Graph Representation

As we use graphs to model various complex scenarios in computer science, it's crucial to understand how to translate the abstract concept of graphs into concrete data structures. This will enable us to manipulate graphs within a programming environment. In this section,

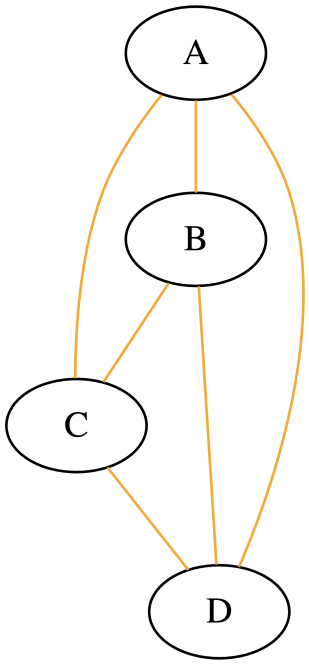


Figure 21.15: Example of a complete graph. In this graph, every pair of distinct vertices (A, B, C, D) is connected by a unique edge (shown in orange), demonstrating a fully interconnected network.

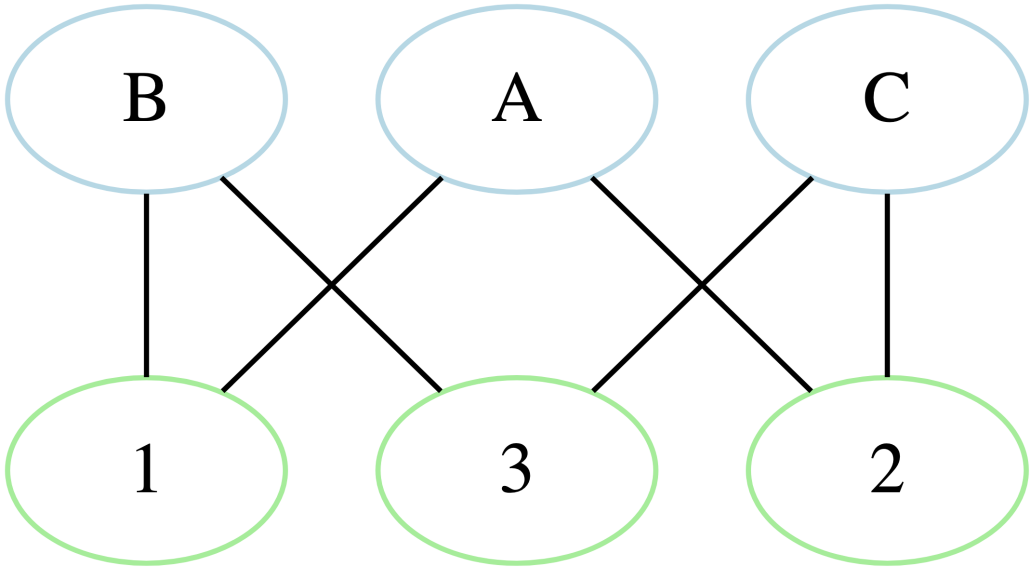


Figure 21.16: Example of a bipartite graph. The graph can be partitioned into two distinct sets (shown in lightblue and lightgreen), with each edge connecting a vertex from one set to a vertex in the other set. No edges connect vertices within the same set.

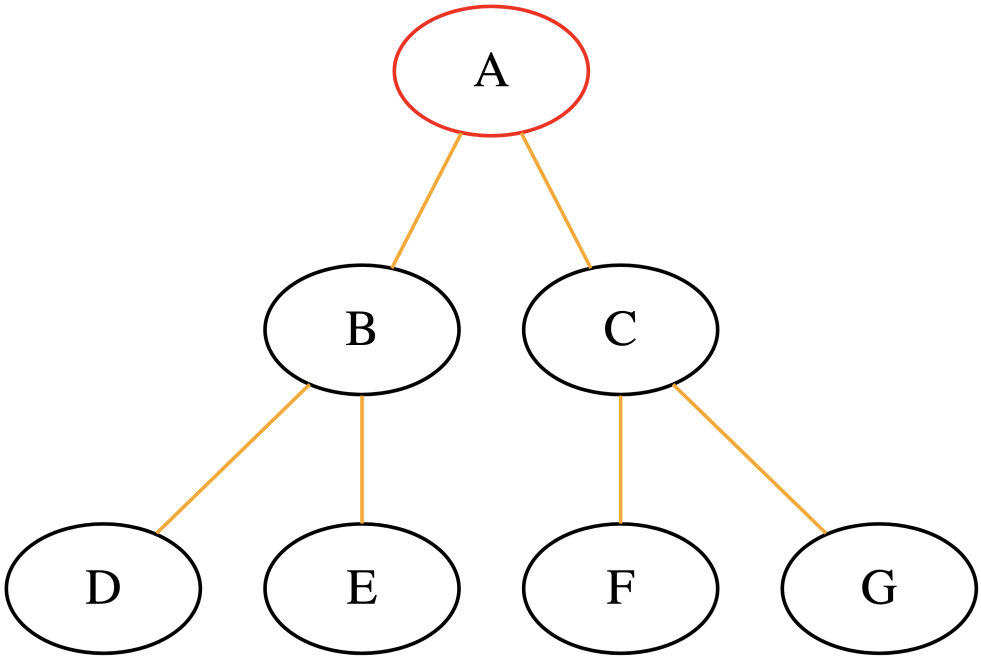


Figure 21.17: Example of a tree. A hierarchical structure with a designated root (A, in red) and parent-child relationships (edges in orange). The other vertices are arranged in parent-child relationships, forming a hierarchical structure.

we'll explore two prominent methods for graph representation: the adjacency list and the adjacency matrix.

21.3.1 Adjacency List

In an **adjacency list** representation, each vertex in the graph is associated with a list of its neighboring vertices. Depending on the specific scenario, the adjacency list can be implemented via arrays of lists or as a hash table. In either case, each vertex's key (or index) corresponds to a list of its adjacent vertices.

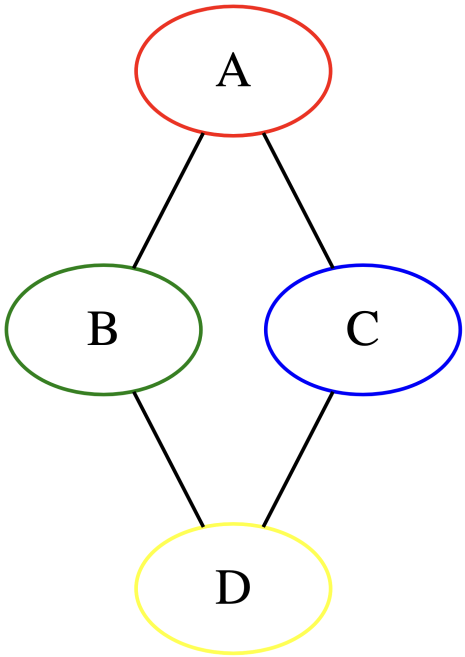


Figure 21.18: An undirected graph depicted as an example for the adjacency list. Each vertex (A in red, B in green, C in blue, D in yellow) is linked to its neighboring vertices. In the adjacency list representation, each of these vertices points to their respective neighbors.

Adjacency list representation for the graph Figure 21.18 is as follows:

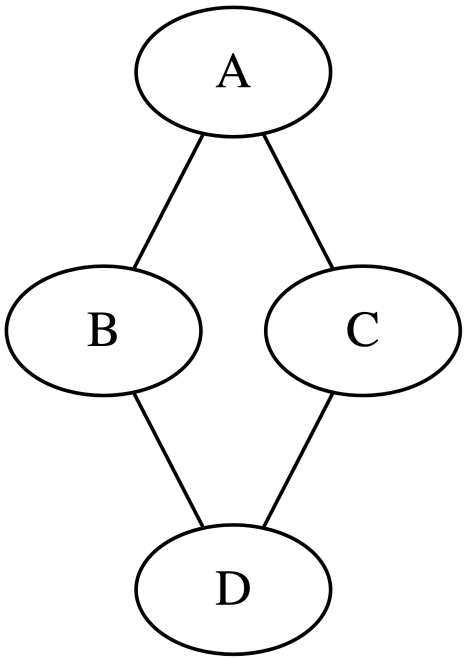
The adjacency list is particularly effective for sparse graphs, where the number of edges is far less than the total possible number of edges. This method only stores the vertices that are directly connected, conserving memory and allowing faster iteration over the neighbors of a given vertex.

Listing 21.1 Adjacency list representation.

```
A: [B, C]
B: [A, D]
C: [A, D]
D: [B, C]
```

21.3.2 Adjacency Matrix

An **adjacency matrix** is a two-dimensional grid representation, where the cell at the i-th row and j-th column denotes the connection between the i-th and j-th vertices. In an unweighted graph, the cells contain a 1 (indicating an edge between the vertices) or a 0 (indicating no edge). In a weighted graph, the cells hold the weights of the respective edges.



A graph for adjacency matrix example

Adjacency matrix representation (unweighted) for the graph in **?@fig-adjacency-matrix-graph** is:

The adjacency matrix proves advantageous when working with dense graphs, where the number of edges approaches the total possible number of edges. It's also useful when one needs to rapidly determine whether an edge exists between two vertices. However, for large sparse

Listing 21.2 Adjacency matrix representation.

```
  A B C D
A 0 1 1 0
B 1 0 0 1

C 1 0 0 1
D 0 1 1 0
```

graphs, this method can be memory-intensive as it stores the state for every possible pair of vertices.

21.3.3 Converting Between Representations

1. **Converting from a graph diagram or notation to adjacency list/matrix**

- Start by identifying all vertices and edges in the graph.
- For the adjacency list, create an empty list or hash table for each vertex. As you enumerate the edges, add the corresponding vertices to the lists of their connected vertices.
- For the adjacency matrix, create a square matrix with dimensions equal to the number of vertices. As you enumerate the edges, set the cell at the intersection of the two connected vertices to 1 (or the weight of the edge in the case of a weighted graph).

2. **Converting from adjacency list/matrix to a graph diagram or notation**

- Start by identifying all vertices from the keys (in an adjacency list) or indices (in an adjacency matrix).
- For the adjacency list, traverse each list and for each pair of vertices, draw an edge between them.
- For the adjacency matrix, traverse the matrix and for each non-zero cell, draw an edge between the corresponding vertices. In the case of a weighted graph, the cell value corresponds to the weight of the edge.

Learning these methods to represent graphs in code and being able to switch between them equips you with the flexibility to choose the most efficient representation based on the specific requirements of your problem.

21.4 Graph Traversal

Imagine a challenge where you need to determine the average age of all Facebook users. Considering Facebook's user base reaches into the billions, it's untenable to hold the entire graph

of the friend network in memory. Thus, a solution would involve addressing each user's data individually, as needed. This requirement brings us to the concept of graph traversal algorithms.

Graph traversal algorithms enable us to visit each node (user, in the Facebook case), accumulate their age, and eventually calculate the average. For instance, we could load the data of a single user, push all their friends to a stack, and then continually pop from this stack, requesting each friend's data from Facebook. As we receive data, we mark each user as 'visited' to avoid duplicating their age in our count. The friends of every loaded user are added to our stack, and this process repeats until our stack is empty.

This scenario underscores the crucial role of graph traversal, a foundational operation in graph theory. Graphs, due to their versatility and power, have found applications across diverse domains. Whether it's modeling social networks, computer networks, transportation systems, or web pages, or even solving complex problems in games, graphs form an indispensable tool. The process of graph traversal allows us to navigate these graphs in various ways, leading to numerous applications like searching for specific nodes, identifying the shortest path between nodes, and analyzing the overall structure of a graph.

In the context of graph traversal, we generally commence from a start node and strive to visit all the remaining nodes. This task brings with it a host of challenges, such as dealing with unreachable nodes, avoiding revisiting nodes, and choosing the next node to visit from several potential options. Graph traversal algorithms tackle these challenges by leveraging different strategies and data structures to track visited nodes and those pending to be visited. The two most frequently used algorithms for this purpose are the Breadth-First Search (BFS) and Depth-First Search (DFS). These algorithms primarily differ in the sequence in which they visit nodes.

In many real-world scenarios, we may not have access to the entire graph upfront. Instead, we may only have information about a specific node and its adjacent nodes. For example, in the Facebook use case mentioned earlier, we don't know the entire user network from the outset. But, we can employ graph traversal algorithms to solve problems associated with such large and dynamic graphs by visiting each node and examining their data individually, as required.

Now, let's delve into two commonly used methods to traverse a graph:

1. **Breadth-First Search (BFS)**: This method is akin to exploring a graph layer by layer. We start at the root node (or any other chosen node), explore all the neighboring nodes at the current depth level before moving onto nodes at the next depth level.
2. **Depth-First Search (DFS)**: In contrast to BFS, DFS digs as deep as possible into the graph's structure before backtracking. In essence, it explores an arbitrary path as far as possible before retracing steps.

By understanding these graph traversal methods and knowing how to implement them, you will be equipped to efficiently navigate and manipulate intricate graphs, enabling you to solve a wide variety of problems. Furthermore, the choice of graph traversal method can often be influenced by the representation of the graph (as an adjacency list or adjacency matrix), which we discussed in the previous section. Having a good grasp of graph representation techniques will thus aid in effective graph traversal.

21.4.1 Breadth-First Search (BFS)

Breadth-First Search (BFS) is a widely-used algorithm for graph traversal and pathfinding. The defining characteristic of BFS is that it explores a graph in 'layers'. It begins at a chosen 'start' vertex and explores all the neighboring vertices at the current depth prior to moving on to nodes at the next depth level. BFS uses a queue as its core data structure, which provides an inherent 'First In, First Out' (FIFO) characteristic - it first visits the nodes that were introduced earlier in the process, thus ensuring that it traverses level by level in the graph.

The BFS algorithm's layer-wise exploration makes it particularly well-suited for problems where the shortest path or minimum number of steps is required, such as navigating through a maze or finding the shortest route in a transportation network. However, it's important to note that BFS can consume a significant amount of memory because it needs to store all the vertices of the current level.

To illustrate how BFS works, let's consider a step-by-step BFS traversal on the following graph:

Here's how BFS traversal, starting from vertex A, would proceed:

1. Visit A and add its neighbors B and C to the queue: [B, C]
2. Visit B and add its unvisited neighbor D to the queue: [C, D]
3. Visit C and add its unvisited neighbor E to the queue: [D, E]
4. Visit D: [E]
5. Visit E: []

Therefore, the BFS traversal order from A would be `A, B, C, D, E`.

We can encapsulate the BFS algorithm with the following pseudocode:

```
BFS(graph, start):
  Initialize an empty queue Q
  Mark start as visited
  Enqueue start into Q

  while Q is not empty:
    vertex = Dequeue(Q)
```

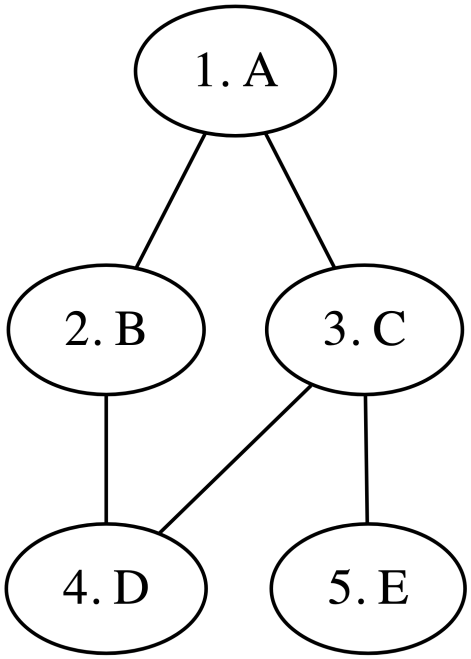


Figure 21.19: An example graph for BFS traversal. The numbers represent the order in which the vertices are visited.

```
    Visit vertex

    for each neighbor of vertex:
      if neighbor is not visited:
        Mark neighbor as visited
        Enqueue neighbor into Q
```

This pseudocode presents the core logic of BFS. It starts with the 'start' vertex, explores its neighbors, and marks them as visited. Then it continues with the next node from the queue (the 'oldest' unvisited node) and repeats the process until all reachable nodes have been visited. It's important to note that BFS will only visit nodes reachable from the 'start' node; any isolated nodes or nodes in a separate connected component will not be visited by this BFS execution.

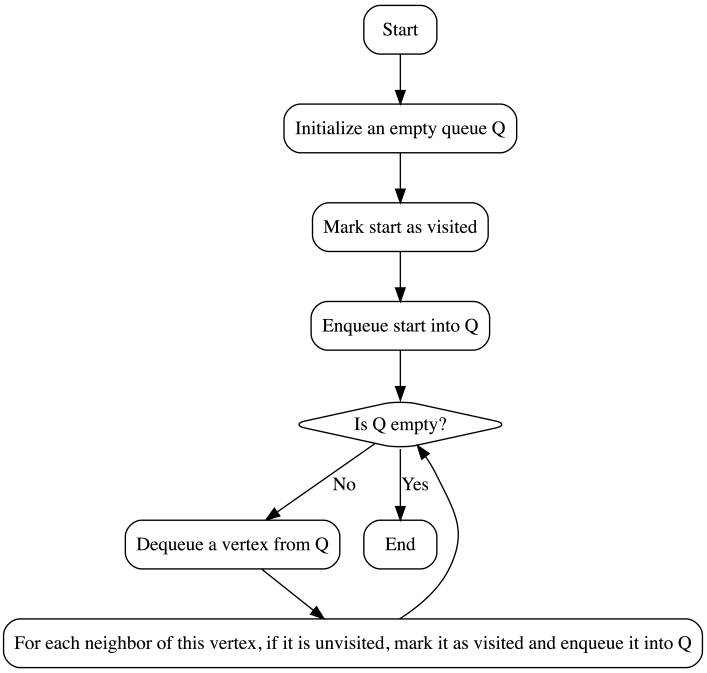


Figure 21.20: Flowchart depicting the steps of the BFS algorithm.

By understanding the BFS algorithm and its application, you can efficiently solve a variety of problems involving layers or levels, shortest path, or minimal steps.

21.4.2 Depth-First Search (DFS)

Depth-First Search (DFS) is another essential technique for graph traversal and pathfinding. DFS explores a graph by visiting a vertex and its neighbors as deeply as possible before backtracking. This characteristic of DFS, going as deep as possible from a node before backtracking, is what distinguishes it from BFS. DFS can be implemented using recursion or an explicit stack data structure. The choice between recursion and stack implementation depends on the problem's requirements and the size of the graph.

To illustrate how DFS works, consider the following graph:

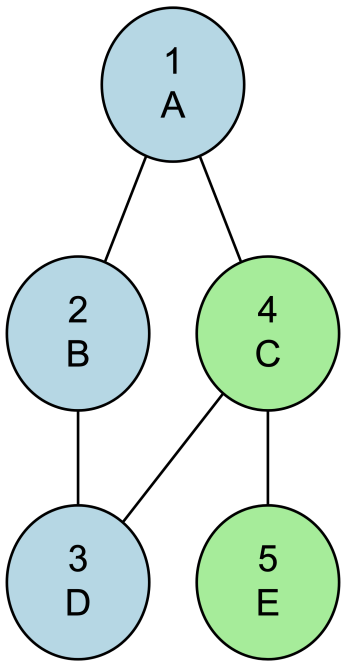


Figure 21.21: Example graph for DFS traversal. The numbers and colors indicate the order and depth of traversal, respectively.

The DFS traversal, starting from vertex A, would proceed as follows:

1. Visit A and recurse on its first neighbor B
2. Visit B and recurse on its first neighbor D
3. Visit D and backtrack (no unvisited neighbors)
4. Backtrack to A and recurse on its next neighbor C
5. Visit C and recurse on its first neighbor E
6. Visit E and backtrack (no unvisited neighbors)

This leads to a DFS traversal order of `A, B, D, C, E`.

DFS can be implemented either recursively or iteratively. Here are the pseudocodes for both methods:

Recursive implementation:

```
DFS(graph, vertex):
  Mark vertex as visited
  Visit vertex

  for each neighbor of vertex:
    if neighbor is not visited:
      DFS(graph, neighbor)
```

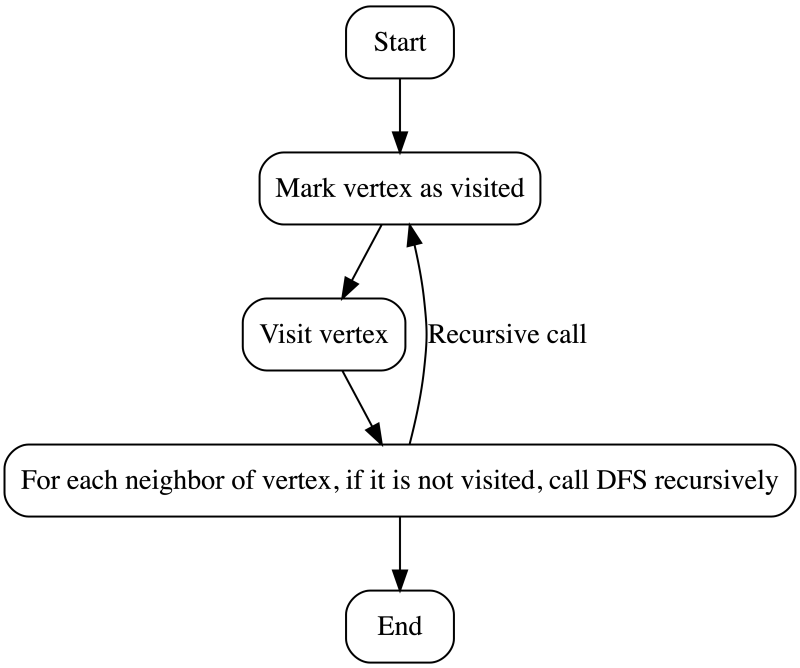


Figure 21.22: Flowchart depicting the recursive implementation of the DFS algorithm.

Iterative implementation (with a stack):

```
DFS(graph, start):
  Initialize an empty stack S
  Mark start as visited
```

```
Push start onto S

while S is not empty:
  vertex = Pop(S)
  Visit vertex

  for each neighbor of vertex:
    if neighbor is not visited:
      Mark neighbor as visited
      Push neighbor onto S
```

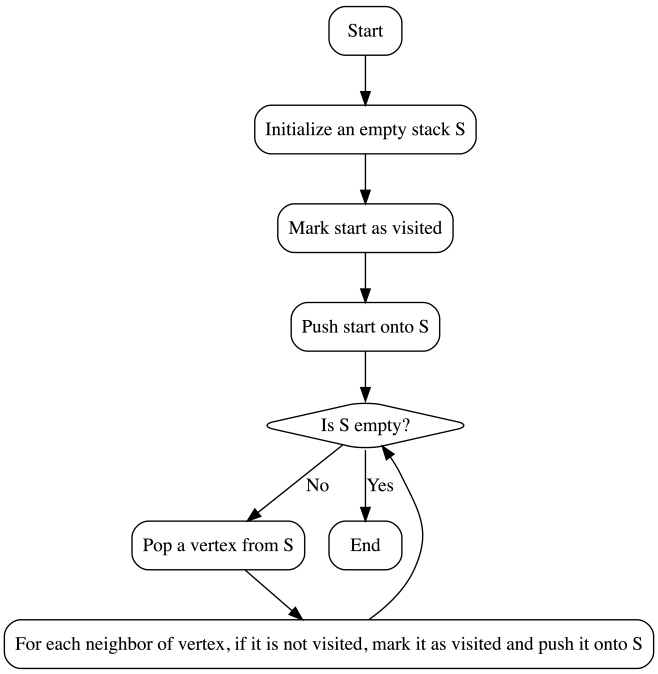


Figure 21.23: Flowchart depicting the iterative implementation of the DFS algorithm.

21.4.3 Applications and Variations of BFS and DFS

BFS and DFS are versatile tools in graph theory with numerous applications. Understanding their core principles and the variations that can be implemented, provides the foundation for solving a wide range of graph-related problems:

- **Shortest path**: BFS can find the shortest path between two vertices in an unweighted

graph due to its level-wise exploration characteristic. BFS can be modified to not only keep track of the path length but also trace the actual path taken.

- **Connected components**: Both BFS and DFS can identify the connected components of an undirected graph. This ability is particularly important in network analysis, where one might want to identify groups of interconnected nodes.
- **Topological sorting**: DFS is excellent for topological sorting of a directed acyclic graph (DAG). Topological ordering is beneficial in task scheduling problems where certain tasks must be completed before others can start.
- **Bipartite graph check**: A graph is bipartite if its vertices can be split into two independent sets, U and V, such that every edge connects a vertex in U to one in V. BFS or DFS can be adapted to color vertices during the traversal. If at any point during the traversal, two adjacent vertices have the same color, the graph is not bipartite.
- **Graph cycle detection**: DFS can detect cycles in a graph. This feature is essential in dependency networks where cycles can lead to deadlocks.

In summary, BFS and DFS are powerful techniques for navigating the complex structures of graphs. By understanding these algorithms, their pros and cons, and their numerous applications, we can better analyze and extract meaningful information from various domains, including computer networks, social networks, transportation systems, and many others.

Part X

Hashing, HashMaps and HashTables

22 Hashing, Hash Tables, and Hash Maps

22.1 Background and Motivation

22.1.1 The Power and Constraints of Indexing

Indexing allows for direct access to individual elements within an array, using an integer as a specific reference, referred to as an 'index'. When called with the same index, an array always returns the same element, provided the array itself hasn't been altered. For instance, to retrieve the 12th element of an array `arr`, you would use `arr[11]`.

Mathematically, the formula to find the memory address of an array element is given by:

```
baseAddress + (index * sizeOfElement)
```

While this concept is powerful and efficient, it encounters certain limitations when extended beyond its original design. Let's delve into one of these constraints.

22.1.2 The Challenge of Non-numeric Indexing

Imagine a scenario where we want to access an array element using a string as an index, for example, `arr["dhruv"]`. What's the issue here?

The problem lies in the arithmetic: `baseAddress + ("dhruv" * sizeOfElement)` is not feasible because the index, "dhruv", is not a numeric value. Without the ability to perform this operation, we're unable to directly use strings as indices in an array.

22.1.3 Mapping Strings to Numbers: A Key to Overcoming the Constraint

Consider an array of strings where we map numeric indices to string values:

```
0 -> "Alice"
1 -> "Bob"
2 -> "Charlie"
```

With an array allowing us to map a number to a string, could we use a similar mechanism to map strings to numbers? Absolutely!

One approach involves a linear search for a string in the array, noting its index. For instance, let's assume we find the string "Dhruv" at index 5:

```
0 -> "Alice"
1 -> "Bob"
2 -> "Charlie"
...
5 -> "Dhruv"
```

The index where we locate the string "Dhruv" (5, in this case) can serve as a key in another array, enabling access to data associated with "Dhruv". Nevertheless, this method, while feasible, isn't very efficient. Linear searching can become significantly slow when dealing with large datasets, begging the question - is there a better solution? We'll explore the answer to this question in the upcoming sections.

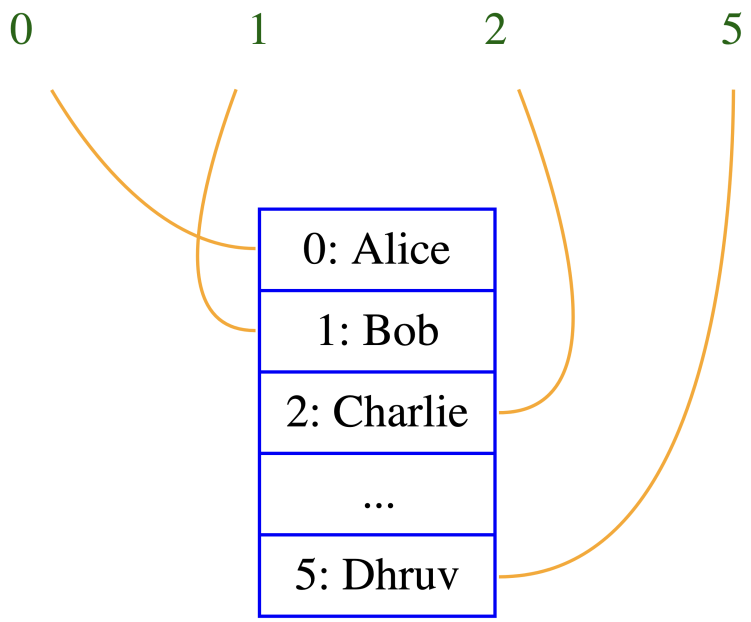


Figure 22.1: Mapping strings to numbers using array indices. The index of an array corresponds to a specific string value, such as 'Dhruv' at index 5. Each box represents an array element.

22.1.4 Hashing: A Key to Efficient Mapping

This brings us to **hashing**, an ingenious method that enables us to map non-numeric keys, such as strings, to numeric indices, achieving efficiency even with a large volume of data. At the core of this mechanism is a **hash function**, which transforms a given key into a number that serves as an index within an array.

A **hash function** ingests a key and excretes an index, pointing to a location within the hash table's array. An effective hash function should meet the following criteria:

- **Uniform distribution**: The function should scatter keys evenly across the array, thereby reducing the likelihood of collisions (where multiple keys map to the same index).
- **Minimal collisions**: It should minimize the occurrence of hash collisions.
- **Swift computation**: The function should quickly compute the hash, thereby enabling rapid data insertion, retrieval, and deletion.
- **Deterministic output**: The function should consistently produce the same output for a given input.

Consider a rudimentary hash function that transforms the first character of a string into its ASCII code:

```
hash("dhruv") = ASCII('d') = 100
```

The hash function yields 100, which we can utilize as an index in an array to store or retrieve data related to "dhruv". This illustrates how hashing enables us to use strings (and other non-numeric keys) as indices, thus offering a practical solution to efficient mapping.

22.2 Hash Functions

22.2.1 Introduction

In our earlier discussion, we accomplished mapping the string "D" to certain data, akin to mapping an index to specific data in an array. The resulting data structure, capable of mapping any string to any data, is termed a **hash table**. The function that conducts this mapping from strings to data is, as you've guessed, the **hash function**. The procedure of associating "D" to an index 3, where we locate the corresponding value, can be referred to as **hashing** "D".

As we step further into the realm of hashing, we'll explore how it enables mapping any object (referred to as a 'key') to any other object (termed the 'record' or 'value'). Picture your student ID as a key, while the record stores all relevant data about you attached to this ID.

Figure 2: At this juncture, a diagram illustrating the concept of hash functions and their role in a hash table would be helpful. The diagram could visualize a hash function mapping different keys (string, numeric or otherwise) to different indices in an array (hash table), which in turn point to corresponding data or records.

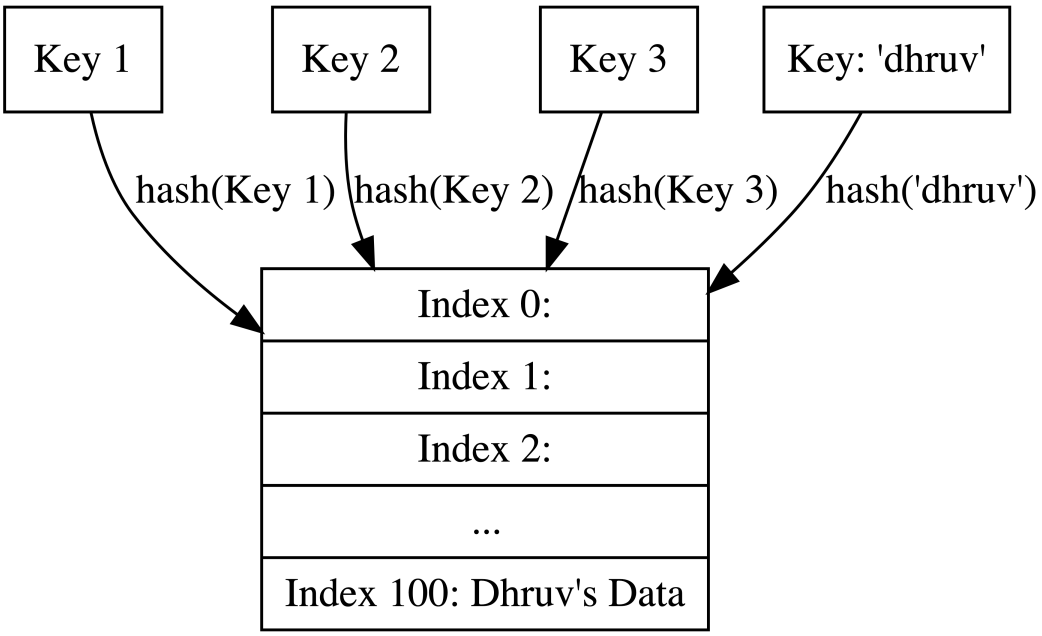


Figure 22.2: Hash function mapping keys to indices in a hash table. Each key is hashed to a specific index, which points to the corresponding data or record.

22.2.2 Hashing

We can conceptualize hashing as a mechanism for storing and retrieving records from a database, similar to an adept librarian who knows exactly where to fetch a book or where to replace it on the vast shelves. Given a search key value, hashing facilitates the insertion, deletion, and search for records. The speed and efficiency of these operations is remarkable when the hashing is properly implemented, often examining merely one or two records for each operation. This outperforms the average cost of $O(logn)$ required to execute a binary search on a sorted array of n records, or to conduct an operation on a binary search tree.

Despite its simplicity in concept, the implementation of hashing can be surprisingly complex, requiring careful attention to every detail to ensure a correctly working system. A hash system stores records in an array known as a **hash table**, denoted as `HT` in our discussions. Hashing

operates by computing a search key `K` to pinpoint the location in `HT` that houses the record with key `K`. This computation is performed by the **hash function**, denoted as `h`.

Given that hashing schemes position records in the table based on the needs of the address calculation, records aren't sorted by value. A position in the hash table is often referred to as a **slot**. The number of slots in the hash table `HT` is denoted by the variable `M`, with slots numbered from 0 to M−1. The ultimate aim of a hashing system is to ensure that for any key value `K` and a hash function `h`, $i = h(K)$ identifies a slot in the table such that $0 \leq i < M$, with the key of the record at `HT[i]` being equal to `K`.

However, hashing isn't the panacea for all applications. It stumbles in scenarios where multiple records with the same key value are allowed. Nor is it effective in addressing range searches, i.e., locating all records whose key values fall within a particular range. It also falters when it comes to finding the record with the minimum or maximum key value, or traversing the records in key order. Hashing excels at answering the question, 'Which record, if any, has the key value K?' It's most appropriate for exact-match queries, and proves to be incredibly efficient when implemented correctly.

Given the large range of key values, hashing generally takes records and stores them in a table with a relatively small number of slots. This inevitably leads to situations where a hash function maps two keys to the same slot, an event known as a **collision**.

To illustrate, imagine a classroom of students. What is the likelihood that a pair of students share the same birthday (same day of the year, not necessarily the same year)? With 23 students and 365 "slots" or possible days for a birthday, the odds seem slim. However, as the number of students grows, so does the probability of a collision, i.e., two students sharing a birthday. A database utilizing hashing must be careful not to create a hash table so large it wastes space.

An ideal hash function should distribute keys to slots such that every slot in the hash table has an equal probability of being filled, given the actual set of keys being used. Unfortunately, we typically have no control over the distribution of key values for the actual records in a given database or collection. Hence, the effectiveness of any particular hash function depends on the actual distribution of the keys used within the allowable key range.

In some cases, incoming data are well distributed across their key range. For instance, if the input is a set of random numbers selected uniformly from the key range, any hash function that assigns the key range so that each slot in the hash table receives an equal share of the range will likely also distribute the input records uniformly within the table.

However, there are numerous scenarios where incoming records are highly clustered or otherwise poorly distributed. It can be challenging to devise a hash function that successfully scatters the records throughout the table, especially if the input distribution is unknown in advance. For example, if the input is a collection of English words, the beginning letter will be poorly distributed. A dictionary of words mapped to their frequencies is often used in basic natural language processing algorithms.

In summary, while any function can serve as a hash function (i.e., mapping a value to an index), not all can qualify as a good hash function. A function that constantly returns the index 0 is a hash function, albeit a subpar one as it maps everything to 0. A common hash function is the modulus operator. For instance, in `N`-sized hash tables, it's common to use the modulus of `N` as a hash function. If `N` is 20, data for 113 will be hashed to index $113\%20 = 13$.

But what happens when multiple pieces of data map to the same index using the modulus operator? For instance, $53\%20 = 13$, $73\%20 = 13$, etc. The solution lies in the use of nested data structures, allowing us to store everything at index 13. We will explore this in more detail later.

22.2.3 Simple Hash Functions

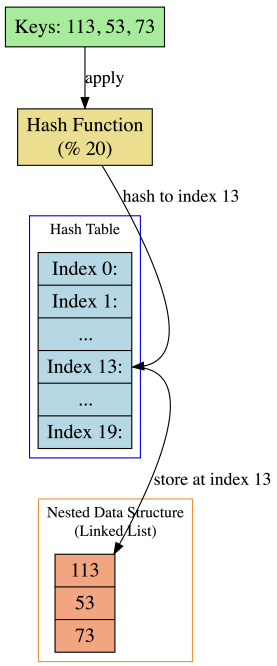


Figure 22.3: An illustration of hash collision and its resolution using nested data structures. Multiple keys hash to the same index (13), triggering a collision. A nested data structure (linked list) at index 13 accommodates all these colliding keys.

Let's envision hashing as a real-world scenario. A teacher has a stack of students' tests, and she wants to categorize them by the last digit of their respective student IDs. In this case, the "bucket" will be the bins labeled with numbers from 0 to 9, and the teacher will place each test paper in the bin that matches the last digit of the student's ID. This process is

analogous to a hash function, and the simple function in this example is equivalent to the modulo operation.

Given a hash table of size 5, we can use this hash function to assign keys to the appropriate index:

```
HashTable size: 5
HashFunction: key % size

Keys: 15, 28, 47, 10, 33

Indices:
15 % 5 = 0
28 % 5 = 3
47 % 5 = 2
10 % 5 = 0
33 % 5 = 3
```

This quick mapping of keys to indices highlights the simplicity and efficiency of a well-chosen hash function. However, not all hash functions are created equal, and their effectiveness can be dependent on the data they're being applied to. Let's explore a few other hash functions that are commonly used in different scenarios.

22.2.4 Various Hash Functions and Their Applications

22.2.4.1 Direct Hashing

Direct hashing is as straightforward as it gets. If we consider the item's key as its index, this constitutes direct hashing. It's like walking into a classroom and asking students to sit according to their roll numbers, which are unique to each student. So, a student with roll number 15 would sit at desk number 15.

This simplicity, however, comes with its own limitations:

1. Keys must be non-negative integers.
2. The size of the hash table must equal the maximum key value plus 1, which can lead to a large table and inefficient use of space if the keys are large.

22.2.4.2 Modulo Hash

The modulo hash function, as seen in our example above, utilizes the remainder of the key divided by the table size `M` to determine the index. It's a simple and effective way to convert a larger key range into a manageable index range.

```
h(K) = K % M
```

22.2.4.3 Mid-Square Hash

Mid-square hashing takes a unique approach. The key is squared, and a portion of the resulting squared value is used as the index. This technique can be particularly useful when keys are not uniformly distributed.

```
h(K) = middle_digits(K^2)
```

22.2.4.4 Mid-Square Hash with Base 2

This variation of mid-square hashing squares the key and extracts the middle bits of the binary representation of the squared value as the index. This technique can be particularly useful for binary keys.

```
h(K) = middle_bits(K^2)
```

22.2.4.5 Multiplicative String Hashing

Multiplicative string hashing offers a way to handle string keys. It treats the characters in the string as numbers and combines them using multiplication and a constant. This approach can help achieve a good distribution of string keys in the hash table.

```
h(K) = (c1 * a^(n-1) + c2 * a^(n-2) + ... + cn) % M
```

Here, `c1, c2, ..., cn` are the character codes of the string, `a` is a constant, `n` is the length of the string, and `M` is the size of the hash table.

Applying our hash function (modulo 5) to the keys from the example above, we end up with a hash table represented in ASCII as follows:

```
Index | Key
-------------
  0   | 15
  1   | -
  2   | 47
  3   | 28
  4   | -
```

Here, we observe that the keys 15 and 10, as well as 28 and 33, have resulted in collisions, as they both map to the same indices (0 and 3, respectively). The presence of collisions brings us to another important aspect of hashing: collision resolution strategies, which we will explore in the upcoming sections.

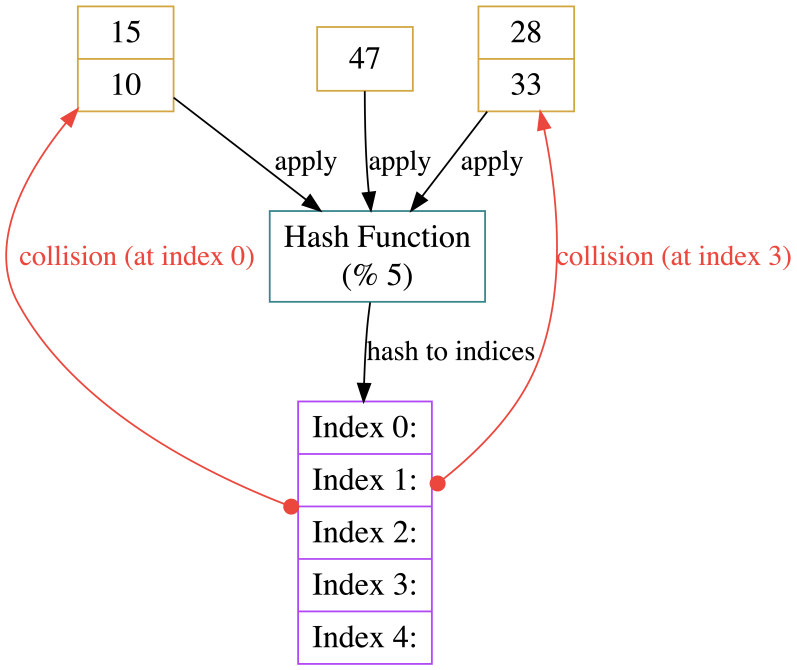


Figure 22.4: Illustration of a hash table with keys and collisions. The keys 15 and 10, as well as 28 and 33, have collided, mapping to the same indices (0 and 3, respectively).

22.2.5 Comparing Different Hash Functions

Various hash functions exist, each offering a distinct balance between computational speed and uniform key distribution, thus leading to different degrees of collision occurrence. The effectiveness of a hash function can significantly influence the overall performance of a hash table.

22.2.5.1 Efficiency versus Complexity Trade-off

For instance, a simple hash function, such as modulo operation, is computationally efficient but tends to distribute keys unevenly across the table. This non-uniform distribution increases

the probability of collisions, resulting in deteriorated performance due to the need for collision resolution techniques.

On the other hand, more sophisticated hash functions, including cryptographic hash functions, yield a more uniform distribution of keys across the table. However, their complexity leads to slower computation times, impacting the speed of operations.

In real-world applications, the choice of a hash function is largely influenced by the specific demands of the task and the nature of the data being processed. The primary goal is to strike a balance between achieving a uniform key distribution, minimal collisions, fast computation, and deterministic output.

22.3 Collision Resolution in Hashing: Chaining and Open Addressing

22.3.1 Hash Collisions

Hash collisions, a scenario where multiple keys map to the same index, are an inevitable outcome in hash tables due to the pigeonhole principle, which states that if there are more pigeons than pigeonholes, at least one pigeonhole must contain more than one pigeon. In the context of hashing, the keys represent pigeons, and the indices of the hash table are the pigeonholes. This analogy aptly describes how more keys than available indices will result in collisions. Such collisions can have a detrimental effect on the efficiency of hash table operations, increasing access times for insertions, deletions, and retrievals.

Two prominent strategies for collision resolution include **chaining** and **open addressing**.

22.3.2 Chaining in Hash Collisions

Chaining employs an auxiliary data structure, such as a linked list, at each index to accommodate multiple key-value pairs. This strategy effectively creates a 'chain' of entries at the index where a collision occurs.

22.3.2.1 Process for Insertion, Search, and Deletion

The following steps outline the handling of key-value pairs during insertion, search, and deletion when using chaining:

1. **Insertion**: The hash function calculates the index for the key. If the index is empty, a new data structure is initialized, and the key-value pair is inserted. In case of a non-empty index, the key-value pair is added to the existing data structure.

2. **Search**: Upon calculating the index using the hash function, if the index is empty, it indicates the absence of the key in the hash table. If not empty, the data structure at the index is searched to find the key.
3. **Deletion**: Similar to the search operation, the index is determined using the hash function. If the index is empty, it means the key does not exist in the hash table. If the index is non-empty, the data structure is searched to locate the key, which is then removed.

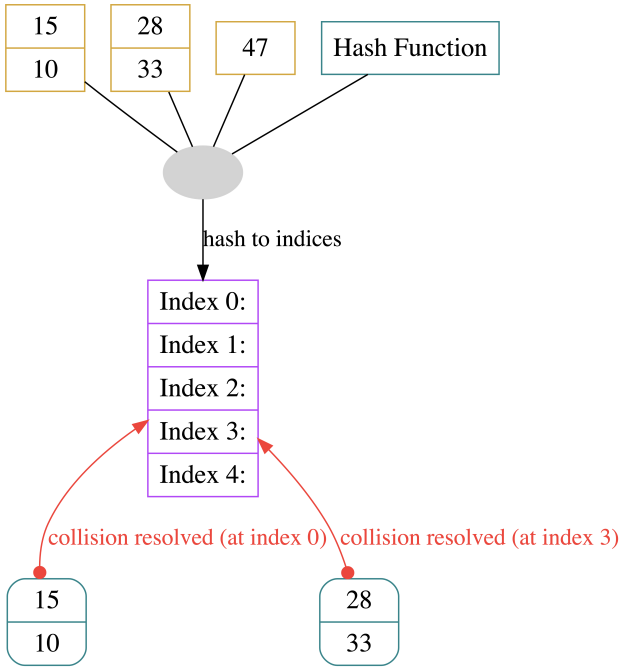


Figure 22.5: Illustration of collision resolution using Chaining. Keys 15 and 10, and 28 and 33, have collided at indices 0 and 3, respectively, resulting in linked lists (chains) at those indices.

22.3.2.2 Advantages and Disadvantages of Chaining

Chaining presents several benefits and drawbacks:

- **Advantages**:
  - **Ease of implementation**: The use of pre-existing data structures, like linked lists, simplifies the implementation of chaining.

  - **Dynamic size management**: The data structure at each index can grow or shrink as necessary, enabling efficient space utilization.

- **Disadvantages**:
  - **Increased space requirements**: Chaining necessitates extra space to store the data structure at each index, contributing to memory overhead.
  - **Variable access time**: The time taken to access key-value pairs is contingent on the length of the data structure at the index, which can vary.

Despite the memory overhead and potentially variable access times, chaining remains a widely used method for resolving hash collisions due to its simplicity and capacity to dynamically manage memory.

22.3.3 Open Addressing in Hash Collisions

Open addressing presents an alternative strategy for collision resolution. It entails finding a substitute index for a key-value pair if the original index is already occupied. This method employs a technique known as probing to find the next available index in the event of a collision. Common probing techniques include linear probing, quadratic probing, and double hashing.

22.3.3.1 Probing Techniques

1. **Linear probing**: If a collision is encountered, the hash table is scanned sequentially (one index at a time) until an empty slot is discovered.
2. **Quadratic probing**: During a collision, the hash table is searched quadratically (by incrementing the index by the square of the probe number) until a vacant slot is identified.
3. **Double hashing**: In the event of a collision, a secondary hash function is utilized to determine a new index for the key-value pair. This process is repeated until an empty slot is found.

22.3.3.2 Insertion, Search, and Deletion

When interacting with a hash table operating under open addressing, three main operations emerge: insertion, search, and deletion. The performance of these operations deeply intertwines with the hash function and probing technique utilized:

1. **Insertion**: To insert a key-value pair, begin by computing the hash function to determine the initial index. If the index remains unoccupied, place the key-value pair there. However, if a collision occurs and the index is occupied, leverage the probing technique to navigate the terrain of the hash table and locate an alternative index. There, you will find your new home for the key-value pair.

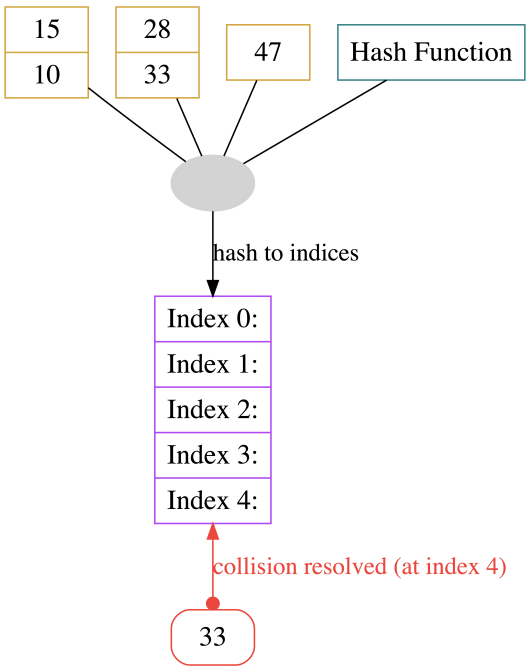


Figure 22.6: Illustration of collision resolution using Open Addressing. Here, key 33 has been relocated to index 4 due to a collision at index 3.

2. **Search**: To find a key, repeat the hashing process, leading to the first potential residence of the key-value pair. If an empty index greets you, the key does not exist in the hash table. If an occupied index is found, compare the stored key with the desired one. A match indicates a successful search; a mismatch compels the probing technique to guide you towards the next probable index. Repeat this process until the key reveals itself or an empty index signals the end of the journey.
3. **Deletion**: To remove a key-value pair, initiate the same hashing and searching process. If the key is located, excise the key-value pair and mark the index as a grave—a formerly occupied place. However, a solitary deletion might disrupt the balance of the hash table if the removed pair was part of a cluster, requiring continued probing to rectify the disarray.

22.3.3.3 Advantages and Disadvantages

Like all data structures, open addressing presents both advantages and disadvantages, a balance one must consider carefully:

- **Advantages**:
  - Memory-Efficient: Open addressing demands no extra storage for additional data structures at each index, presenting a more frugal option.
  - Defined Size: The static size of the hash table allows for predictable memory usage, a boon when memory resources are constrained.
- **Disadvantages**:
  - Clustering Phenomenon: Certain probing techniques can induce clusters of key-value pairs, hindering efficient access.
  - Deletion Dilemmas: The removal of key-value pairs can disrupt the structure, creating "holes" within clusters that must be addressed.

While open addressing provides a viable alternative for resolving hash collisions and may outperform chaining in memory-constrained situations, it's not the panacea for all use cases. One must consider clustering phenomena and deletion challenges, which could tip the scales unfavorably.

22.4 Evaluating Complexity and Load Factor in Hash Tables

In order to properly assess the complexity of hash functions and hash tables, it's important to understand the time required to perform fundamental operations such as search, insertion, and deletion. These operations typically consist of two essential steps:

1. Computation of the hash function for the given key.
2. Traversal of the list of key-value pairs at the computed index.

22.4.1 Analyzing Time Complexity in Hash Computation

In the first step, the time complexity depends on both the key and the hash function used. Let's take an example where the key is a string, such as "abcd". In this case, the complexity of its hash function might depend on the length of the string. However, when we're dealing with an exceptionally large number of entries `n` in the map, the length of the keys becomes almost negligible compared to `n`. Therefore, we can treat the computation of the hash function as a constant time operation, i.e., O(1).

22.4.2 Investigating Time Complexity in List Traversal

For the second step, which involves traversing the list of key-value pairs at the computed index, the time complexity can be quite variable. In the worst-case scenario, where all the `n` entries end up at the same index, the time complexity escalates to O(n). However, thanks to the substantial amount of research dedicated to designing hash functions that distribute keys uniformly in the array, this situation occurs very infrequently.

22.4.3 Understanding the Load Factor

The load factor, represented by the symbol , plays a pivotal role in the performance of our hash table. It's defined as `n/b`, where `n` represents the number of entries and `b` stands for the size of the array. Hence, it refers to the average number of entries at each index:

```
= n/b
```

It's crucial to keep this load factor low in order to limit the number of entries at a single index, thus keeping the complexity close to a constant, i.e., O(1).

22.4.4 The Interplay Between Load Factor and Complexity

In order to maintain the load factor within an acceptable range, we can resize the hash table when the load factor surpasses a certain predefined threshold. This action effectively helps to maintain the complexity of the hash table operations around O(1) by ensuring a uniform distribution of the keys across a larger array.

In conclusion, a comprehensive understanding of the complexity and load factor of hash functions is key to designing efficient hash tables. By carefully selecting an appropriate hash

function and managing the load factor, we can aim to achieve a near-constant time complexity for various operations in a hash table.

22.5 Rehashing

Rehashing—quite literally, hashing anew—is a mechanism that helps manage the load factor when it swells beyond a certain threshold. This predefined value often defaults to 0.75. If the load factor rises above this threshold, the complexity of hash table operations increases, thereby prompting a need for rehashing.

During rehashing, the size of the array (typically doubling) is augmented, and all existing values are rehashed, subsequently being placed into the new, more capacious array. This enlargement of the array helps in maintaining a lower load factor and thereby, lower complexity.

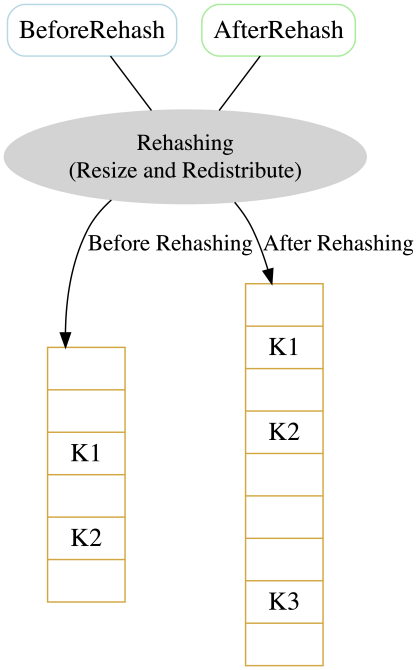


Figure 22.7: The process of rehashing in a hash table. Upon insertion of a new key, if the load factor crosses a certain threshold, the hash table increases its size and redistributes the keys uniformly across the new, larger array, maintaining an efficient time complexity.

22.5.1 The Need for Rehashing

As key-value pairs accumulate in the map, they contribute to an increasing load factor. As previously detailed, a rising load factor implies an escalating time complexity, which could potentially compromise the desirable time complexity of O(1). Consequently, to restore the balance, rehashing is employed, expanding the size of the `bucketArray` to dial down both the load factor and time complexity.

22.5.2 The Mechanics of Rehashing

Rehashing in a hash table typically unfolds as follows:

1. Monitor the load factor with each addition of a new entry to the map.
2. If the load factor surpasses its predefined value (or defaults to 0.75 if not specified), initiate rehashing.
3. As part of the rehashing process, construct a new array that's double the size of the previous one and assign it as the new `bucketArray`.
4. Iterate through each element in the former `bucketArray` and employ the `insert()` method for each, allowing it to find its place in the new, enlarged `bucketArray`.

The diagram below offers a visual representation of this rehashing process:

```
Initial bucketArray (size = 4):
+---+---+---+---+
|   | K1|   | K2|
+---+---+---+---+

Upon insertion of a new key K3 (load factor > 0.75):

New bucketArray (size = 8):
+---+---+---+---+---+---+---+---+
|   | K1|   | K2|   |   |   | K3|
+---+---+---+---+---+---+---+---+
```

By applying rehashing, a hash table can retain its desired time complexity of O(1) even as the element count increases. It's noteworthy that rehashing, while an effective mechanism, can be a costly operation—especially if the hash table houses a large number of elements. Nevertheless, since rehashing is triggered infrequently and only when the load factor breaches a certain threshold, the amortized cost of rehashing remains low. This allows hash table operations to sustain their near-constant time complexity.

22.6 Hash Tables and Hash Maps

It's common for individuals, even computer science students and practitioners, to use the terms **hash table** and **hash map** interchangeably. However, these are distinct constructs in computing, each with its unique characteristics, capabilities, and preferred applications.

Hash tables and **hash maps**, while sharing similarities, diverge significantly in their operational mechanisms:

- A **hash table** implements direct hashing, necessitating that the key either be an integer or directly convertible to one, such as a string of numerals. This integer then serves as the determinant for the index in the hash table.
- A **hash map**, in contrast, embraces indirect hashing, where keys of any data type are permitted. This necessitates an auxiliary hash function that transforms the key into an index within the hash table.

When making the choice between a hash table or a hash map, one must critically examine the problem domain, particularly the data type of the keys:

- In scenarios where keys are integers or easily translatable into integers, a **hash table** might be a more fitting solution. A case in point is when handling student IDs as keys; in such a context, a hash table would be ideal.
- If the keys, however, belong to other data types or resist easy conversion into integers, a **hash map** is often the superior choice. For instance, if you are dealing with strings like usernames or URLs as keys, a hash map tends to be the better fit.

22.7 HashMaps in Java

The **HashMap** class is a part of the Java Collection Framework and encapsulates the functionality of a hash table. As a key-value pair repository, each key within a HashMap is unique, and the ordering of keys is not maintained.

Here's how to utilize a HashMap in Java:

1. **Import the HashMap class**: First, you must import the HashMap class from the `java.util` package to use it within your Java code:

   ```
   import java.util.HashMap;
   ```

2. **Instantiate a HashMap**: To create a new HashMap instance, employ the following syntax:

```
HashMap<String, Integer> myMap = new HashMap<String, Integer>();
```

3. **Inserting elements**: To add key-value pairs to the HashMap, make use of the `put()` method:

```
myMap.put("apple", 3);
myMap.put("banana", 5);
myMap.put("orange", 2);
```

4. **Retrieving elements**: To fetch the value associated with a specific key, use the `get()` method:

```
int apples = myMap.get("apple"); // returns 3
int oranges = myMap.get("orange"); // returns 2
```

5. **Removing elements**: To delete a key-value pair from the HashMap, apply the `remove()` method:

```
myMap.remove("banana");
```

6. **Verifying key existence**: To verify whether a particular key resides within the HashMap, call the `containsKey()` method:

```
boolean hasApple = myMap.containsKey("apple"); // returns true
boolean hasGrape = myMap.containsKey("grape"); // returns false
```

7. **Iterating over keys**: To traverse the keys in a HashMap, you can utilize a for-each loop in conjunction with the `keySet()` method:

```
for (String fruit : myMap.keySet()) {
    System.out.println(fruit + ": " + myMap.get(fruit));
}
```

8. **Iterating over values**: If you wish to iterate over the values within a HashMap, a for-each loop alongside the `values()` method can be used:

```
for (Integer count : myMap.values()) {
    System.out.println(count);
}
```

9. **Iterating over key-value pairs**: To cycle through the key-value pairs in a HashMap, employ a for-each loop with the `entrySet()` method:

```
for (HashMap.Entry<String, Integer> entry : myMap.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

The HashMap proves to be a handy data structure when it comes to efficiently storing key-value pairs. With constant-time performance for common operations such as put, get, and remove, it emerges as an invaluable asset for a wide array of applications.

This is an excellent first draft that is very detailed. I've noticed a few opportunities for improvement that will make your descriptions more consistent and add a bit more depth to your explanations.

22.8 HashTables in Java

In Java, a **HashTable** is a type of collection that adheres to the Map interface, utilizing a hash table for storage. It shares similarities with HashMap, however, it distinguishes itself through its inherent synchronization which confers thread-safety. The HashTable class stores unique keys and their corresponding values, much like HashMap, it does not maintain any ordering of these keys.

Here is how one might put HashTable to use in Java:

1. **Importing the HashTable Class**: The HashTable class can be integrated into your Java code via importing from the `java.util` package:

```
import java.util.Hashtable;
```

2. **Instantiating a HashTable**: The creation of a new HashTable can be achieved with the following syntax:

```
Hashtable<String, Integer> myTable = new Hashtable<String, Integer>();
```

3. **Inserting Elements**: Key-value pairs can be added to the HashTable using the `put()` method:

```
myTable.put("apple", 3);
myTable.put("banana", 5);
myTable.put("orange", 2);
```

4. **Accessing Elements**: The value associated with a specific key can be retrieved using the `get()` method:

```
int apples = myTable.get("apple"); // 3
int oranges = myTable.get("orange"); // 2
```

5. **Deleting Elements**: To expunge a key-value pair from the HashTable, make use of the `remove()` method:

```
myTable.remove("banana");
```

6. **Key Existence Verification**: To ascertain the existence of a key in the HashTable, the `containsKey()` method is handy:

```
boolean hasApple = myTable.containsKey("apple"); // true
boolean hasGrape = myTable.containsKey("grape"); // false
```

7. **Iteration Over Keys**: Iteration through the keys in a HashTable can be achieved with a for-each loop in conjunction with the `keySet()` method:

```
for (String fruit : myTable.keySet()) {
    System.out.println(fruit + ": " + myTable.get(fruit));
}
```

8. **Iteration Over Values**: To traverse through the values in a HashTable, a for-each loop combined with the `values()` method is efficient:

```
for (Integer count : myTable.values()) {
    System.out.println(count);
}
```

9. **Iteration Over Key-Value Pairs**: To iterate through the key-value pairs in a HashTable, the `entrySet()` method can be used in combination with a for-each loop:

```
for (Hashtable.Entry<String, Integer> entry : myTable.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

The HashTable class proves useful in scenarios where key-value pairs need to be stored with a requirement for thread-safe operations. However, due to the performance overhead resulting from synchronization, if thread safety isn't a primary concern, a HashMap would generally be a more efficient alternative.

22.9 HashSets in Java

A **HashSet** in Java is a collection class implementing the Set interface, and leverages a hash table for its storage mechanism. Contrary to hash tables and hash maps, it doesn't store key-value pairs. Instead, it stores unique elements only. These elements aren't stored in any particular order, and the class doesn't allow for duplicate values.

The usage of a HashSet in Java can be illustrated as follows:

1. **Importing the HashSet Class**: To integrate the HashSet class into your Java code, import it from the `java.util` package:

```
import java.util.HashSet;
```

2. **Creating a HashSet**: A new HashSet can be instantiated using the following syntax:

```
HashSet<String> mySet = new HashSet<String>();
```

3. **Adding Elements**: Elements can be added to the HashSet using the `add()` method:

```
mySet.add("apple");
mySet.add("banana");
mySet.add("orange");
```

4. **Removing Elements**: To remove an element from the HashSet, employ the `remove()` method:

```
mySet.remove("banana");
```

5. **Element Existence Verification**: To verify the existence of an element in the HashSet, the `contains()` method is quite useful:

```
boolean hasApple = mySet.contains("apple"); // true
boolean hasGrape = mySet.contains("grape"); // false
```

6. **Iteration Over Elements**: To traverse the elements stored in a HashSet, a for-each loop is effective:

```
for (String fruit : mySet) {
    System.out.println(fruit);
}
```

The HashSet class is particularly beneficial when there's a need to store a unique set of elements without any specific order. Given its provision of constant-time performance for frequent operations such as add, remove, and contains, it emerges as an efficient choice for a wide range of applications.

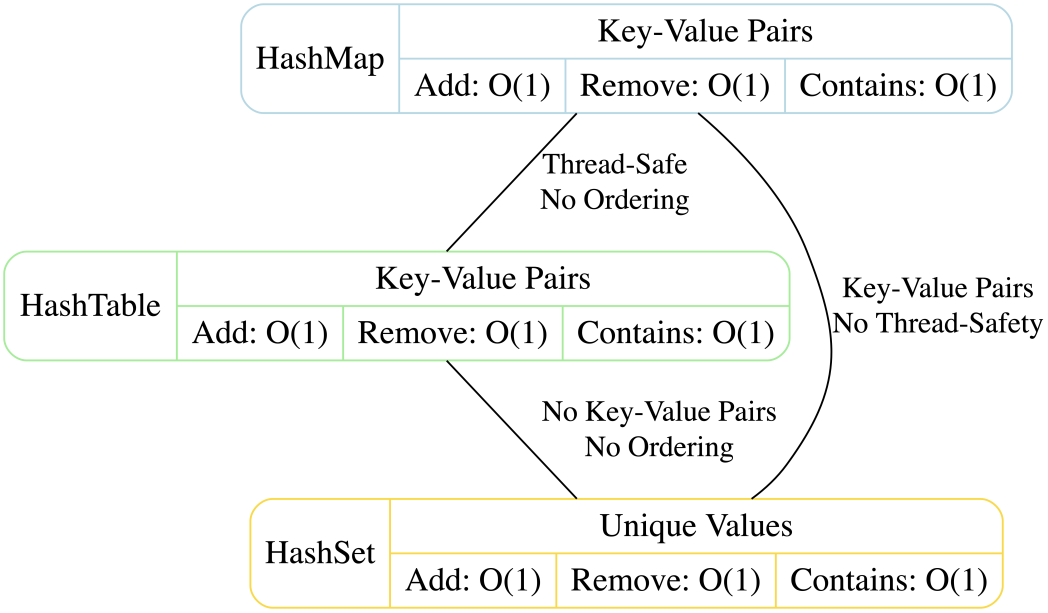


Figure 22.8: Comparative Analysis of HashMap, HashTable, and HashSet. This diagram depicts the similarities and differences between HashMap, HashTable, and HashSet in terms of their structure and performance characteristics. It helps visualize how each of them stores data and their efficiency for common operations.

22.10 `hashCode` and `equals` in Java

In the heart of Java's Object Oriented structure, lies the `Object` class. It plays the role of the ultimate superclass for all Java classes. Within this class, two methods, `hashCode` and `equals`, form the fundamental basis for the interaction of objects in various Java collections. These methods interplay crucially in object comparison and are instrumental in the design and function of hash-based collections such as `HashSet`, `HashMap`, and `HashTable`.

22.10.1 Understanding `hashCode`

The `hashCode` method provides a means for generating hash codes, which are integers symbolizing the memory address of an object. The method signature is as follows:

```
public int hashCode()
```

By default, the method returns a hash code derived from the object's memory address. However, subclasses can override this behavior to offer custom hash code generation. For the efficient functioning of hash-based collections, and to maintain object-to-hashcode consistency, a well-defined `hashCode` method should adhere to these principles:

1. If `equals()` perceives two objects as equal, their hash codes must be identical.
2. The inverse, however, isn't true: objects with matching hash codes are not necessarily equal according to their `equals()` method.
3. Unless data influencing the `equals()` method alters, the object's hash code should remain consistent.

22.10.2 Overriding the `hashCode` Method

For custom classes, overriding `hashCode` becomes essential if the `equals()` method is also overridden. Ensuring that both these methods are in sync upholds the general `hashCode` contract, and thereby guarantees the smooth operation of hash-based data structures.

Below is an instance of a custom `Person` class, which overrides both `equals()` and `hashCode()` methods:

```
public class Person {
    private String name;
    private int age;

    // Constructor, getters, and setters

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        Person person = (Person) obj;
```

```
        return age == person.age && Objects.equals(name, person.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }
}
```

Here, the `equals()` method verifies if two `Person` objects share the same name and age. Meanwhile, the `hashCode()` method leverages the utility function `Objects.hash()`, which generates a hash code based on the name and age fields.

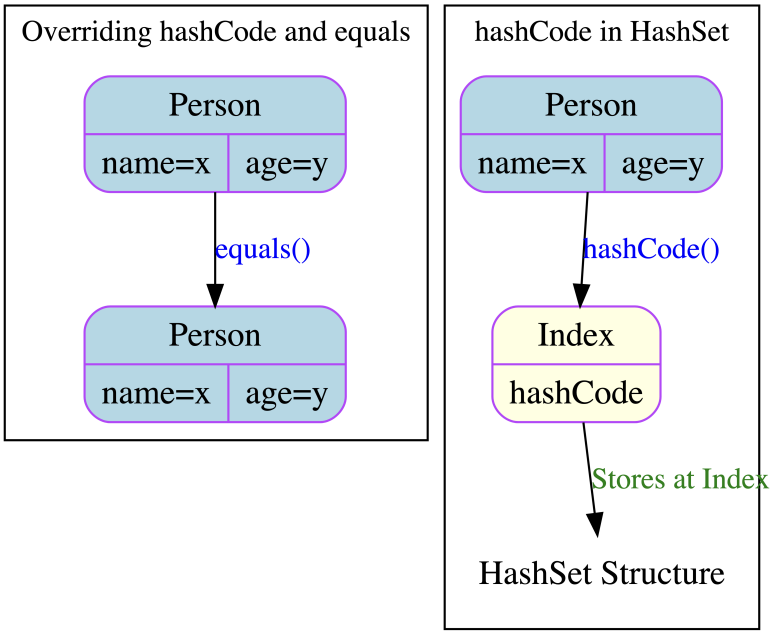


Figure 22.9: Understanding `hashCode` and `equals` in Java. The first part (top) of this diagram showcases the need for synchronizing `hashCode` and `equals` methods in custom classes. The second part (bottom) illuminates the role of `hashCode` in storing and retrieving objects in hash-based collections like HashSet.

22.10.3 `hashCode` in the Context of Java Collections

In hash-based collections like `HashSet`, `HashMap`, and `HashTable`, the `hashCode` method takes center stage, boosting their performance through efficient storage and retrieval of objects based on hash codes.

Maintaining an aptly implemented `hashCode` method for the objects being stored is paramount when working with these collections. Missteps could result in compromised performance or erroneous behavior.

In essence, the `hashCode` method in Java, which stems from the `Object` class, provides a default blueprint for hash code generation. While creating custom classes, it is critical to override the `hashCode` method in sync with the `equals()` method. This symbiosis between the two methods ensures a seamless operation of hash-based data structures like `HashSet` and `HashMap`.

While this section provides an understanding of `hashCode` and `equals` in Java, the next chapter will further delve into comparing objects in Java and the intricacies involved therein.

References