

DATA STRUCTURE AND ALGORITHM

A **data structure** is a way of organizing data in a computer's memory or on a disk so that it can be accessed and manipulated efficiently. It is a collection of data elements that are organized in a particular way, depending on the type of data being stored and the operations that will be performed on the data.

There are several different types of structures, including **arrays**, **linked lists**, **queues**, **trees** and **graphs**. Each data structure has its own unique way of organizing data and provides a different set of operations that can be performed on the data.

For example, an **array** is a **data structure** that stores a **collection of data elements of the same type in contiguous memory locations** (contiguous refers a block of records that are logically adjacent to one another in to physically adjacent sectors). The elements of an array can be accessed using an **index**, which represents the position of the element in the array.

On the other hand, a **linked list** is a data structure that **stores a collection of data elements that are connected by links or pointers**. Each element in the list contains a data field and a **pointer to the next element in the list**.

A **program** is a **set of instructions that is written in a programming language to solve a specific problem**. A solution to a problem can be broken down in two parts: the organization of the data and the sequence of computational steps to solve the problem.

The way data is organized in computer's memory is known as a data structure. A data structure is a collection of data elements that are organized in a specific manner to allow for efficient storage, retrieval and manipulation of individual data elements. The choice of data structures depends on the type of data being stored and the operations that will be performed on the data. For example, if the data is organized in an array, the data elements are stored in contiguous memory locations and accessed using an index. If the data is organized in a linked list, each data element is connected to the next using **pointers** or **links**.

The sequence of computational steps to solve a problem is known as algorithm. An algorithm is a set of instructions that is designed to solve a specific problem efficiently. Algorithms can be designed to perform a variety of tasks, such as searching for data elements, sorting data elements or performing mathematical calculations.

For example, a sorting algorithm can be used to sort a collection of data elements in ascending or descending order. A searching algorithm can be used to find a specific data element in a collection of data elements.

Linear and Non Linear Data structures

In computer science, data structures are classified into two main categories, **linear** and **non linear** data structures. **Linear data structures** are those in which data elements are arranged in a **linear sequence**, such as an **array** or a **linked list**. In contrast, **non-linear** data structures are those in which data elements are not arranged in a **linear sequence**, such as **trees** or **graphs**.

Some common types of **linear data structures** include :-

1) **Arrays** :- An array is a collection of data elements of the **same type** that are stored in **contiguous memory locations**. The elements of an array can be accessed using an **index**, which represents the position of the element in the array. Arrays are efficient for storing and accessing data when the size of the array is fixed and the elements are accessed sequentially.

2) **Linked Lists** :- A linked list is a data structure that stores a collection of data elements that are **connected by links or pointers**. **Each element in the list contains a data field and a pointer to the next element in the list**.

3) **Stacks** :- A stack is a data structure that stores a collection of data elements in **a last in, first out (LIFO) manner**. The elements can be added to or removed from the stack only at one end, known as the **top** of the **stack**.

4) **Queues** :- A queue is a data structure that stores a collection of data elements in a **first in first out (FIFO) manner**. The elements can be

added to the read of the queue and removed from the front of the queues.

Non-linear data structures are those in which the data elements are not arranged in a linear sequence. In contrast to linear data structures, insertion and deletion of elements in non linear data structures are not necessarily performed in a sequential manner. Some common types of non-linear data structures include trees and graphs.

1) **Trees** :- A tree is a non linear data structure that consists of a collection of nodes connected by edges. Each node in a tree contains a data element and pointers to its child nodes. Trees are hierarchical data structures that are used to represent relationships between data elements. The top most node in a tree is called the root node, and each node in the tree has a parent node except for the root node.

2) **Graphs** :- A graph is a non-linear data structure that consists of a collection of vertices or nodes connected by edges. Each edge in a graph connects two vertices, and each vertex in a graph contains a data element. Graphs are used to represent relationships between data elements that are not hierarchical.

Homogeneous and non-homogeneous data structures

Homogeneous data structures is one in which all the data elements are of the same type. An array is a common example of homogeneous data structure, where all elements are of the same data types and the size of the array is fixed.

Non-homogeneous data structures is one in which data elements may not be of the same type. A structure in the C programming language is an example of a non-homogeneous data structure, where data elements of different data types can be grouped together. Non-homogeneous data structures are useful when we need to store different types of data elements together.

Static and Dynamic data structures

A **static data structure** is one whose size and structure associated memory location is fixed at compile time. An array is an example of a static data structure, where the size of the array is fixed at a compile time. Static data structures are useful when the size of the data is known in

advance and the data elements are accessed sequentially.

Dynamic data structures is one that can expand or shrink as required during program execution, and their associated memory locations change accordingly. A linked list is an example of a dynamic data structure, where new elements can be added or removed from the list during program execution. Dynamic data structures are useful when the size of the data is not known in advance or when the data elements are accessed randomly.

Data structure Operations

Algorithms are a set of operations that manipulate the data represented by different data structures using various operations. These operations play a significant role in the processing of data and are as follows :-

Creating :- Creating a data structure is the first operation, where the data structure is declared, initialized, and memory locations are reserved for data elements. This operation prepares the data structures for future operations.

Inserting :- Inserting is the operation of adding a new data element to the data structure. It involves finding the appropriate location in the data structure and adding the element in that location.

Updating :- This operation involves changing the value of data elements within the data structure. For example, when updating an element in an array, the value of the element at a specific index is changed.

Deleting :- This operation involves removing a data element from the data structure. For example, when deleting an element from a queue, the element at the front of the queue is removed.

Traversing :- This operation involves accessing data elements within a data structure. This is the most frequently used operation associated with data structures for example, when traversing an array, each element in the array is accessed sequentially.

Searching :- This operation involves finding the location of a data element with a given value or finding the locations of all elements that satisfy one or more conditions in the data structure. For example, when searching for an element in a binary search tree, the tree is traversed to find the element with the given value.

Sorting :- This operation involves arranging data elements in some logical order, such as ascending order or descending order of student names. For example, when sorting an array, the elements are rearranged in a specific order based on a comparison function.

Merging :- This operation involves combining the data elements in two different sorted sets into a single sorted set. For example, when merging two sorted arrays, the elements are combined and rearranged in a specific order.

Destroying :- This operation involves deallocating memory associated with the data structure when it is no longer needed. For example, when destroying a linked list, each node in the list is deallocated, and the memory is freed.

Abstract Data Type (ADT)

Abstract Data Types (ADTs) are a way of organizing and defining data structures in computer science. An ADT consists of two main components: the data to be stored and the operations that can be performed on that data. It provides a high-level specification of the data and operations, without specifying the implementation details or the programming language used.

ADTs focus on the essential properties of the data and the operations, abstracting away unnecessary details. This allows programmers to work with the data and perform operations on it without having to worry about the underlying implementation. It promotes modular design and encapsulation, as the ADT provides a well-defined interface for interacting with the data.

For example, let's consider the ADT "employees of an organization". The relevant attributes of an employee may include their name, ID, sex, age,

salary, department, and address. These attributes define the data that can be stored in the ADT.

The ADT also specifies the operations that can be performed on the employee data, such as hiring, firing, retiring, updating employee information, and retrieving employee details. These operations define how the data can be manipulated and accessed.

The ADT abstracts away non-relevant attributes, such as weight, color, and height, as they are not necessary for the specific purpose of managing employees in the organization.

It's important to note that the ADT is independent of any specific programming language or implementation. It provides a conceptual model for organizing and working with data, and it can be implemented in various programming languages using different data structures and algorithms.

By defining and using ADTs, programmers can create reusable and modular code, as the ADT provides a clear interface for interacting with the data and hides the implementation details. This promotes code maintainability, code reuse, and overall software design.

Abstraction

In the context of algorithms, abstraction refers to the process of simplifying complex details and focusing on the essential aspects of a problem or task. It involves hiding unnecessary details and exposing only the relevant information that is needed to understand and solve the problem.

Abstraction allows us to think about problems and algorithms at a higher level of understanding, without getting caught up in the specific implementation details. It helps in managing the complexity of algorithms and making them more manageable and easier to comprehend.

When we abstract an algorithm, we focus on its overall purpose, functionality, and behavior, rather than the specific steps or low-level

implementation details. We define the inputs, outputs, and operations of the algorithm, without getting into the specific algorithms or data structures used.

Abstraction enables us to design algorithms and analyze their efficiency without being tied to a specific programming language or platform. It allows us to generalize and create algorithms that can be applied to different scenarios or problem domains.

By abstracting away unnecessary details, we can create more modular and reusable algorithms. We can think about algorithms as black boxes, where we know what the algorithm does and how to interact with it, without needing to understand the internal workings.

CHAPTER TWO

Algorithm and Algorithm Analysis

An algorithm can be defined as a finite set of instructions or a step-by-step procedure that takes raw data as input and processes it to produce refined data. It is a fundamental tool used to solve well-specified computational problems. Algorithms are utilized in various fields, including computer science, mathematics, engineering, and many other domains.

Properties of an Algorithm:

1. **Input:** Every algorithm receives zero or more data items as input. These inputs can be provided externally, and they serve as the initial information on which the algorithm operates.
2. **Output:** An algorithm should produce at least one output data item as a result of its computations. The output represents the refined or processed data that is obtained after executing the algorithm.
3. **Definiteness:** Each instruction or step within an algorithm must be clear, unambiguous, and have a precise meaning. Ambiguity should be avoided to

ensure that there is no confusion or multiple interpretations regarding the intended operations.

4. **Finiteness:** An algorithm is expected to terminate after a finite number of steps for all possible inputs. In other words, if we were to follow the instructions of an algorithm, it should eventually reach a stopping point, ensuring that the process does not continue indefinitely.
5. **Effectiveness:** Every instruction in an algorithm must be feasible and achievable. It means that each step should be practical and provide a meaningful contribution towards solving the problem at hand.
6. **Sequence:** An algorithm consists of a series of steps, and each step should have a clearly defined preceding and succeeding step. The order in which instructions are executed matters, and the algorithm should specify the correct sequence to achieve the desired outcome. The first step, known as the start step, and the last step, known as the halt step, must be explicitly identified.
7. **Correctness:** An algorithm should compute the correct answer or solution for all possible valid inputs. It is crucial for an algorithm to produce accurate and reliable results, meeting the requirements and expectations defined by the problem statement.
8. **Language Independence:** An algorithm should be independent of any specific programming language. It should be defined in a way that focuses on the logic and operations rather than the syntax of a particular programming language. This property allows the algorithm to be implemented in different programming languages as needed.
9. **Completeness:** An algorithm should provide a complete solution to the given problem. It should consider all relevant

aspects and requirements of the problem domain, ensuring that the solution addresses all necessary components and constraints.

10. **Efficiency:** An algorithm should aim to solve the problem using the least amount of computational resources such as time and space. Efficiency is concerned with optimizing the algorithm's performance and minimizing the resources required for its execution. This involves reducing unnecessary operations, optimizing data structures, and improving time complexity to achieve faster and more economical solutions.

These properties collectively define the characteristics that an algorithm should possess in order to be effective, reliable, and practical in solving computational problems.

Analysis of Algorithm

Algorithm analysis involves determining the computing time and storage space required by different algorithms. Its purpose is to predict the resource requirements of algorithms in a given environment. When solving a problem, there are multiple possible algorithms to choose from, and algorithm analysis helps in selecting the best algorithm using scientific methods.

To classify data structures and algorithms as good or efficient, precise analysis is needed to evaluate their resource requirements. **The main resources considered in algorithm analysis are:**

1. **Running Time:** Running time refers to the amount of time an algorithm takes to execute and provide the output. **It is often treated as the most important resource** since computational time is typically the most valuable resource in problem-solving domains.
2. **Memory Usage:** Memory usage refers to the amount of storage space an algorithm requires to store data and intermediate

results during its execution. Analyzing memory usage helps understand the algorithm's space complexity and its impact on the overall performance and efficiency.

3. **Communication-Bandwidth:** Communication bandwidth refers to the amount of data that needs to be transferred or communicated between different components of an algorithm or between different processes in a distributed system. This resource is particularly relevant in parallel and distributed computing scenarios.

Using actual clock-time as a consistent measure of an algorithm's efficiency can be challenging because it can vary based on various factors, including specific processor speed, current processor load, and specific data used in a particular run of the program. The input size and properties, as well as the operating environment, can also influence the clock-time measurement.

Instead of relying on absolute clock-time, algorithm analysis often focuses on analyzing algorithms based on the number of operations they perform. By quantifying the number of operations, such as comparisons, assignments, or iterations, required to solve a problem, it becomes possible to evaluate the algorithm's efficiency relative to the input size.

This approach allows for a more meaningful understanding of how an algorithm's efficiency changes with different input sizes. It provides insights into how the algorithm scales and whether it exhibits desirable characteristics, such as sub-linear, linear, or polynomial time complexity, as the input size increases.

By analyzing algorithms based on operation counts and studying their time and space complexities, researchers and developers can make informed decisions about algorithm selection and design. This helps in creating efficient solutions, optimizing performance, and understanding the trade-offs between different algorithms when solving computational problems.

Complexity Analysis

Complexity Analysis is a systematic study of the cost of computation, whether measured in time units, operations performed, or storage space required. It aims to provide a meaningful measure that allows for the comparison of algorithms independent of the underlying operating platform.

There are two main aspects to consider in complexity analysis:

1. **Time Complexity:** Time complexity focuses on determining the approximate number of operations or steps required to solve a problem of a given size, typically denoted as "n." It provides an estimate of the computational time or running time required by an algorithm as the input size increases. Time complexity analysis helps understand how the algorithm's performance scales with larger inputs.
2. **Space Complexity:** Space complexity involves determining the approximate amount of memory or storage space required to solve a problem of a given size "n." It measures the memory usage or storage requirements of an algorithm as the input size grows. Space complexity analysis helps evaluate the efficiency and memory usage characteristics of an algorithm.

Complexity analysis generally involves two distinct phases:

1. **Algorithm Analysis:** In the algorithm analysis phase, the algorithm or data structure is analyzed to produce a function $T(n)$. This function describes the algorithm's complexity by quantifying the number of operations or memory accesses performed as a function of the input size. The goal is to derive an equation or formula that captures the algorithm's behavior in terms of its operations or memory usage.

2. **Order of Magnitude Analysis:** In the order of magnitude analysis phase, the function $T(n)$ derived from the algorithm analysis is further analyzed to determine the general complexity category to which it belongs. This analysis involves classifying the function into a complexity class such as constant time ($O(1)$), logarithmic time ($O(\log n)$), linear time ($O(n)$), quadratic time ($O(n^2)$), or other categories. The order of magnitude analysis provides a concise way to describe the algorithm's complexity without focusing on precise constants or lower-order terms.

WHY DO WE NEED ORDER OF MAGNITUDE ANALYSIS

Order of Magnitude Analysis is a crucial step in complexity analysis that aims to classify an algorithm's time or space complexity into a general complexity category. Instead of focusing on precise constants or lower-order terms, this analysis provides a concise representation of an algorithm's complexity.

The need for Order of Magnitude Analysis arises due to the following reasons:

1. **Concise Complexity Description:** Order of Magnitude Analysis allows for a compact and high-level representation of an algorithm's complexity. Rather than specifying exact counts or measurements, it provides a general description that captures the growth rate or behavior of the algorithm as the input size increases.
2. **Comparison of Algorithms:** By classifying algorithms into complexity categories, such as constant time ($O(1)$), logarithmic time ($O(\log n)$), linear time ($O(n)$), quadratic time ($O(n^2)$), and others, it becomes easier to compare and contrast different algorithms. It enables developers and researchers to understand the relative efficiency and performance characteristics of algorithms without getting lost in specific details.

3. **Complexity Classes:** Complexity classes provide a standardized framework for categorizing algorithms based on their growth rates. Each complexity class represents a specific order of magnitude, capturing the rate at which an algorithm's performance changes with respect to the input size. This classification aids in understanding the scalability and efficiency characteristics of algorithms and facilitates algorithm selection based on the problem requirements.

Order of Magnitude Analysis is essential because it allows for a simplified yet meaningful representation of an algorithm's complexity. It enables us to compare algorithms, identify dominant factors, and make informed decisions regarding algorithm selection and optimization strategies. By focusing on the general behavior of an algorithm rather than specific details, this analysis provides a higher-level perspective on the efficiency and performance characteristics of algorithms.

By performing complexity analysis, developers and researchers can gain insights into how algorithms scale with larger inputs and make informed decisions about algorithm selection. It helps identify the most efficient algorithms for specific problem sizes and enables comparisons between different algorithms based on their time and space requirements. Complexity analysis also provides a theoretical foundation for optimizing algorithmic performance and understanding trade-offs between time and space complexities

ANALYSIS RULES

The provided analysis rules outline common guidelines for analyzing the running time of algorithms. These rules help in estimating the time complexity of an algorithm by considering the count of operations involved. Here's a more detailed explanation of each rule:

1. **Arbitrary Time Unit:** The analysis assumes an arbitrary time unit. It provides a relative measure of time without specifying the exact duration. This allows

for a comparative analysis between different algorithms, focusing on the relative number of operations performed.

2. **Operations with Time 1:** The execution of certain basic operations, such as assignment operations, single input/output operations, boolean operations, arithmetic operations, and function returns, is considered to take time 1. This rule simplifies the analysis by treating these operations as equal units of time.
3. **Selection Statements:** The running time of a selection statement (if, switch) is the sum of the time required for the condition evaluation and the maximum running time among the individual clauses within the selection. This rule accounts for the control flow branching and the potential execution of different code paths.
4. **Loops:** The running time of a loop is determined by multiplying the running time of the statements inside the loop by the number of iterations. For nested loops, the analysis is performed from the innermost loop to the outermost loop. It is assumed that each loop executes the maximum number of iterations possible, providing an upper bound estimation.
5. **Function Calls:** Running time of a function call is 1 for setup + the time for any parameter calculations + the time required for the execution of the function body.

Examples

```
1) void func()
{
    int x = 0;
    int i = 1;
    int j = 1;
    cout << "Enter an Integer value:
";
    cin >> n;

    while (i <= n)
    {
```

```

        x++;
        i++;
    }

    while (j < n)
    {
        j++;
    }
}

```

$$T(n) = 1 + 1 + 1 + 1 + 1 + [n+1 + n + n] + [n + n-1]$$

$$T(n) = 5n+5$$

Example :-

```

int sum(int n)
{
    int partial_sum = 0;
    for (int i = 1; i <= n; i++)
        partial_sum = partial_sum + (i * i * i);
    return partial_sum;
}

```

$$T(N) = 1 + [1 + n + 1 + n + 4n] + 1$$

$$T(n) = 6n + 4$$

Example :-

```

int total(int n)
{
    int sum = 0;
    for (int i = 1; i <= n; i++)
        sum = sum + i;
    return sum;
}

```

$$T(n) = 1 + [1 + n + 1 + n + 2n] + 1$$

$$T(n) = 4 + 4n$$

Example :-

```

int count()
{
    int k = 0;
    cout << "Enter an integer: ";
    cin >> n;
}

```

```

    for (i = 1; i <= n; i++)
        k = k + 1;
    return 0;
}

```

$$T(n) = 1 + 1 + 1 + [1 + n + 1 + n + 2n] + 1$$

$$T(n) = 6 + 4n$$

```

int findSumOfPairs()
{
    int sum = 0;
    int n = nums.size();

    for (int i = 0; i < n; i++) {
        for (int j = 1; j < n; j++) {
            sum += nums[i] + nums[j];
        }
    }

    return sum;
}

```

$$T(n) = 1 + 1 + [1 + n + 1 + n] + n[1 + n + n + 3n-1]$$

Example :-

MEASURE OF TIMES

When we talk about algorithms, we want to know how long they will take to run and how efficient they are. To measure this, we use different ways of looking at their performance.

- **Average Case:** This is like looking at how an algorithm performs on average or in typical situations. It considers a range of different inputs and gives us an idea of what to expect most of the time.
- **Worst Case:** This is the longest amount of time an algorithm could possibly take. It looks at the most challenging or difficult inputs and gives us an upper limit on how long the algorithm will run.
- **Best Case:** This is the shortest amount of time an algorithm could take. It looks at

the easiest or simplest inputs and shows us the best possible scenario.

Usually, we are most interested in the worst-case scenario because it tells us how the algorithm will perform no matter what inputs we give it. To describe this, we use something called "Big-O" notation, which helps us understand how the running time of an algorithm grows as the size of the input gets bigger. We use Big-O notation to express the upper limit of the algorithm's time complexity.

So, when we say an algorithm has a Big-O of $O(n^2)$, it means that the running time of the algorithm will not grow faster than the square of the input size.

ASYMPTOTIC ANALYSIS

Imagine you have two algorithms, and you want to compare how fast they are. To do that, you need a way to measure their performance as the input size gets larger and larger.

Asymptotic analysis helps us understand how the running time of an algorithm changes as the input size increases towards infinity. It focuses on the growth rate of the running time, which means how fast the running time increases as the input gets bigger.

To describe this growth rate, we use different notations:

- **Big-Oh Notation (O):** It tells us the upper limit of the algorithm's running time. For example, if an algorithm has a time complexity of $O(n^2)$, it means the running time won't grow faster than the square of the input size. It gives us a worst-case estimate of the running time.
- Imagine you have a race, and you want to know the maximum time it will take for someone to finish. Big-Oh notation tells you the longest possible time it could take. For example, if someone says the race will take $O(n^2)$ time, it means no

one will take longer than n^2 time to finish.

- **Big-Omega Notation (Ω):** It tells us the lower limit of the algorithm's running time. For example, if an algorithm has a time complexity of $\Omega(n)$, it means the running time will grow at least linearly with the input size. It gives us a best-case estimate of the running time.
- Now, let's think about the fastest time someone can finish the race. Big-Omega notation tells you the shortest possible time. If someone says the race will take $\Omega(n)$ time, it means no one will finish in less than n time.

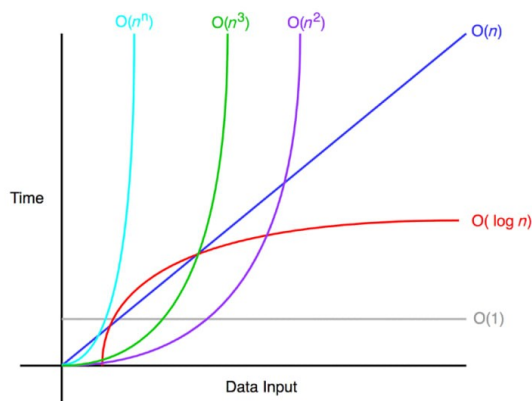
Big-Oh Notation (O) is a way to analyze and describe the upper limit or worst-case running time of an algorithm. It helps us understand how the running time of an algorithm grows as the input size increases. By using Big-O notation, we can estimate the efficiency of an algorithm and compare it with other algorithms.

In Big-O notation, we express the running time of an algorithm as a function of the input size (usually denoted as " n "). The notation $O(f(n))$ represents the upper bound of the running time, meaning the running time will not exceed a certain multiple of the function $f(n)$ as n becomes larger.

Here are some typical orders (growth rates) that are commonly encountered:

- **$O(1)$:** Constant time complexity. The running time of the algorithm remains the same, regardless of the input size. It is considered the best-case scenario.
- **$O(\log n)$:** Logarithmic time complexity. The running time grows logarithmically with the input size. Algorithms with this complexity often divide the problem into smaller subproblems and solve them recursively.

- **$O(n \log n)$** : Log-linear time complexity. The running time grows slightly faster than linearly. Algorithms with this complexity are commonly seen in sorting and searching algorithms like merge sort and quicksort.
- **$O(n)$** : Linear time complexity. The running time grows linearly with the input size. This means that as the input size doubles, the running time also doubles. It represents a proportional relationship between the input size and the running time.
- **$O(n^2)$** : Quadratic time complexity. The running time grows quadratically with the input size. Algorithms with nested loops often have this complexity.
- **$O(2^n)$** : Exponential time complexity. The running time grows exponentially with the input size. Algorithms with this complexity tend to become very slow as the input size increases.



Who is faster $O(n \log n)$ or $O(n)$?

In terms of asymptotic analysis, we compare the growth rates of functions to determine which one grows faster as the input size increases.

In this case, we are comparing $O(n)$ (linear time complexity) and $O(n \log n)$ (log-linear time complexity).

Asymptotically, $O(n \log n)$ grows faster than $O(n)$. This means that as the input size increases, the running time of an algorithm with $O(n \log n)$

complexity will be higher compared to an algorithm with $O(n)$ complexity.

To understand why $O(n \log n)$ is faster, we need to consider the growth rates of these functions. In linear time complexity ($O(n)$), the running time increases linearly with the input size. For example, if the input size doubles, the running time also doubles.

However, in log-linear time complexity ($O(n \log n)$), the running time increases at a slower rate. It grows in a proportional relationship to the input size multiplied by the logarithm of the input size. As the input size grows, the running time increases, but not as quickly as in linear time complexity.

Therefore, for larger input sizes, an algorithm with $O(n \log n)$ complexity will generally be faster than an algorithm with $O(n)$ complexity.

These are just a few examples, and there are many more complexities possible. The goal is to choose an algorithm with the best possible Big-O notation for a given problem to ensure efficiency.

By analyzing the running time using Big-O notation, we can understand how the algorithm performs as the input size grows and make informed decisions about algorithm selection.

ASYMPTOTIC NOTATIONS

Asymptotic notation is a mathematical tool used to describe and analyze the efficiency or performance of an algorithm or function as the input size increases. It helps us understand how the behavior of an algorithm changes when we have more data or a larger problem to solve.

When we talk about the behavior or growth rate of an algorithm, we're interested in how the algorithm's time or space requirements change relative to the size of the input. Asymptotic notation allows us to focus on the most significant factors that affect the algorithm's performance and ignore less significant details.

Three commonly used symbols in asymptotic notation are:

Big O notation (O):- This symbol represents the upper bound or worst-case scenario of an algorithm's time or space complexity. It tells us how the algorithm's performance scales as the input size increases. For example, if we say an algorithm has a time complexity of $O(n^2)$, it means that the algorithm's running time will not exceed a quadratic growth rate as the input size increases. The "O" notation allows us to describe the maximum amount of resources an algorithm may require.

- Represents the upper bound or worst-case scenario of an algorithm's time or space complexity.
- It describes the maximum rate at which an algorithm's performance grows as the input size increases.
- When we say an algorithm has a time complexity of $O(f(n))$, it means the algorithm's running time will not exceed a certain multiple of the function $f(n)$ as the input size grows.
- It provides an upper limit on the growth rate of the algorithm.

Big Omega notation (Ω):- This symbol represents the lower bound or best-case scenario of an algorithm's time or space complexity. It gives us an idea of the minimum amount of resources an algorithm will require to solve a problem as the input size increases. For example, if an algorithm has a time complexity of $\Omega(n)$, it means the algorithm's running time will grow at least linearly with the input size. The " Ω " notation provides information about the minimum efficiency of an algorithm.

- Represents the lower bound or best-case scenario of an algorithm's time or space complexity.
- It describes the minimum rate at which an algorithm's performance grows as the input size increases.
- When we say an algorithm has a time complexity of $\Omega(g(n))$, it means the algorithm's running time will grow at

least as fast as the function $g(n)$ as the input size increases.

- It provides a lower limit on the growth rate of the algorithm.

Big Theta notation (Θ):- This symbol represents both the upper and lower bounds, providing a tight range of possible behaviors for an algorithm's time or space complexity. It describes the average-case scenario when the best and worst cases are similar. For example, if we say an algorithm has a time complexity of $\Theta(n)$, it means that the algorithm's running time grows linearly with the input size, neither faster nor slower. The " Θ " notation allows us to describe the exact growth rate or efficiency of an algorithm within a specific range.

- Represents both the upper and lower bounds, providing a tight range of possible growth rates.
- It describes the average-case scenario when the best and worst cases are similar.
- When we say an algorithm has a time complexity of $\Theta(h(n))$, it means the algorithm's running time grows at the same rate as the function $h(n)$ as the input size increases.
- It provides an exact description of the growth rate of the algorithm within a specific range.

BIG – O THEOREMS

The Big-O theorems and properties help us understand the behavior and relationships between different functions in terms of their growth rates. They provide guidelines and rules for analyzing and comparing functions using Big-O notation.

Detailed Explanation:

Theorem 1: The theorem states that a constant, represented by k , is $O(1)$. It means that regardless of the value of k , it is considered to have a constant complexity or growth rate. In Big-O notation, constants can be ignored because

they do not affect the scalability or efficiency of an algorithm.

Theorem 2: This theorem focuses on polynomials and states that the growth rate of a polynomial is determined by the term with the highest power of n . For example, if we have a polynomial $f(n)$ of degree d , then it is $O(n^d)$. The dominant term in the polynomial determines the overall growth rate of the function.

Theorem 3: The theorem states that a constant factor multiplied by a function does not change its Big-O complexity. The constant factor can be ignored when analyzing the growth rate of the function. For example, if we have a function $f(n) = 7n^4 + 3n^2 + 5n + 1000$, it is $O(n^4)$. The constant factor 7 can be disregarded.

Theorem 4 (Transitivity): This theorem states that if function $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$. It means that if one function has a certain growth rate and another function has a higher growth rate, the first function is also guaranteed to have a growth rate no higher than the second function.

This theorem states that if function $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$. In simpler terms, it means that if one function grows no faster than another function, and that other function grows no faster than a third function, then the first function also grows no faster than the third function.

To understand this theorem, let's break it down further:

1. **Function $f(n)$ is $O(g(n))$:** This means that the growth rate of function $f(n)$ is no greater than the growth rate of function $g(n)$. It indicates that as the input size increases, the resources (time or space) required by $f(n)$ will not exceed the resources required by $g(n)$.
2. **Function $g(n)$ is $O(h(n))$:** This means that the growth rate of function $g(n)$ is no greater than the growth rate of function $h(n)$. It indicates that as the input size increases, the resources required by $g(n)$

will not exceed the resources required by $h(n)$.

The conclusion from these two statements is that:

3. **Function $f(n)$ is $O(h(n))$:** Combining the information from statements 1 and 2, we can say that the growth rate of function $f(n)$ is no greater than the growth rate of function $h(n)$. It implies that as the input size increases, the resources required by $f(n)$ will not exceed the resources required by $h(n)$.

This theorem allows us to chain together multiple comparisons of functions' growth rates using Big-O notation. If we have evidence that one function grows no faster than another, and that other function grows no faster than a third function, we can confidently say that the first function also grows no faster than the third function.

The transitivity property is helpful when comparing and analyzing the efficiency of algorithms. It allows us to make logical deductions about the scalability and resource requirements of algorithms based on their growth rates.

Theorem 5: The theorem states that for any base b , logarithm base b of n ($\log_b(n)$) is $O(\log(n))$. It means that logarithmic functions with different bases grow at the same rate. The base of the logarithm does not significantly affect the growth rate.

Theorem 6: This theorem provides a hierarchy of growth rates for various types of functions. It states that each function in the list is O of its successor. Functions such as constant (k), logarithmic ($\log n$), linear (n), linearithmic ($n \log n$), quadratic (n^2), exponential (2^n , 3^n), larger constants to the n th power, factorial ($n!$), and n to higher powers all have different growth rates. As we move up the list, the growth rate increases.

PROPERTIES OF BIG-O

Higher powers grow faster: When we compare functions with different powers of n , the one

with a higher power grows faster. For example, if we have a function n^r , where r is a positive number, and another function n^s , where s is a positive number and greater than r , then the function n^s grows faster as the input size increases.

Example: Consider two functions, $f(n) = n^2$ and $g(n) = n^3$. As the input size increases, the function $g(n)$ grows faster than $f(n)$ because it has a higher power of n . For instance, when $n = 10$, $f(n) = 100$ and $g(n) = 1000$. Thus, $g(n)$ grows faster than $f(n)$.

Fastest growing term dominates a sum: When we have a sum of functions, the one with the fastest growing term determines the overall growth rate. For example, if we have a sum $f(n) + g(n)$, and the growth rate of $g(n)$ is faster than that of $f(n)$, then the sum is dominated by $g(n)$ in terms of its growth rate.

Example: Let's say we have two functions, $f(n) = n^2$ and $g(n) = n^3 + 1000$. In this case, the fastest growing term in $g(n)$ is n^3 , which dominates the overall growth rate. Even though $f(n)$ has a lower power term and an additional constant, as the input size increases, the term with the highest power, n^3 , will dominate the sum. Thus, the overall growth rate is determined by the fastest growing term.

Exponential functions grow faster than powers: Exponential functions, which are represented as b^n , where b is greater than 1, grow faster than functions with powers of n . This means that as the input size increases, the growth rate of an exponential function outpaces the growth rate of a function with a power of n .

Example: Consider two functions, $f(n) = 2^n$ and $g(n) = n^3$. As the input size increases, the function $f(n)$ with an exponential growth rate grows significantly faster than $g(n)$ with a polynomial growth rate. For example, when $n = 10$, $f(n) = 1024$ and $g(n) = 1000$. The exponential growth of $f(n)$ surpasses the polynomial growth of $g(n)$ as the input size increases.

Logarithms grow more slowly than powers: Logarithmic functions, represented as $\log_b(n)$,

where b is greater than 1, grow more slowly than functions with powers of n . As the input size increases, the growth rate of logarithmic functions is lower compared to functions with powers of n .

Example: Let's compare two functions, $f(n) = \log_2(n)$ and $g(n) = n^2$. As the input size increases, the function $f(n)$ with a logarithmic growth rate grows much more slowly than $g(n)$ with a quadratic growth rate. For instance, when $n = 100$, $f(n) = 6.64$ (approximately) and $g(n) = 10,000$. The logarithmic growth of $f(n)$ is significantly lower compared to the quadratic growth of $g(n)$ as the input size increases.

These properties help us understand and compare the growth rates of functions using Big-O notation. They provide insights into how different types of functions behave as the input size increases. By analyzing and comparing functions based on their growth rates, we can make predictions about the efficiency and scalability of algorithms.

FORMAL DEFINITION OF BIG-O NOTATION

Big O notation is a way to compare and analyze algorithms based on their efficiency and performance. It helps us understand how the runtime or resource requirements of an algorithm change as the input size increases. Big O notation focuses on the largest or most significant term in the algorithm's expression, as it becomes the dominant factor for larger input sizes.

Detailed Explanation: Big O notation provides a standardized way to express the complexity or growth rate of an algorithm. It allows us to make comparisons and predictions about how the algorithm will perform for larger values of input.

Formal Definition: In Big O notation, we say that $f(n) = O(g(n))$ if there exist positive constants " c " and " k " such that for all values of n greater than or equal to " k ", the function $f(n)$ is less than or equal to " c " multiplied by the function $g(n)$. This definition helps us establish an upper bound on the growth rate of $f(n)$ relative to $g(n)$.

Examples: The following points are facts that you can use for Big-Oh problems:

- $1 \leq n$ for all $n \geq 1$
- $n \leq n^2$ for all $n \geq 1$
- $2^n \leq n!$ for all $n \geq 4$
- $\log_2 n \leq n$ for all $n \geq 2$
- $n \leq n \log_2 n$ for all $n \geq 2$

Describing the above

- For any value of n greater than or equal to 1, we can say that 1 is less than or equal to n . This is true for all positive integers.
- Similarly, for any value of n greater than or equal to 1, we can say that n is less than or equal to n^2 . This inequality holds for all positive integers.
- As the value of n increases, the growth rate of 2^n becomes larger than the growth rate of $n!$ (n factorial). This is true for all values of n greater than or equal to 4.
- For values of n greater than or equal to 2, we can observe that the logarithm base 2 of n is less than or equal to n . The logarithmic growth rate is smaller compared to linear growth (n).
- Additionally, for values of n greater than or equal to 2, we can see that n is less than or equal to $n \log_2 n$. The linear growth rate is smaller compared to linearithmic growth ($n \log n$).

Example :- $f(n)=10n+5$ and $g(n)=n$. Show that $f(n)$ is $O(g(n))$.

We want to show that the function $f(n) = 10n + 5$ is "smaller" or "equal to" the function $g(n) = n$ when n gets bigger. We'll do this by finding some numbers that help us compare the two functions.

Step 1: To show that $f(n)$ is $O(g(n))$, we need to find some special numbers called constants c and k . These numbers will help us prove that for any value of n that is greater than or equal to k , the

function $f(n)$ is less than or equal to c multiplied by $g(n)$.

Step 2: Let's try a value for c . We'll choose $c = 15$. Now we need to check if $10n + 5$ is less than or equal to $15n$ for all values of n that are greater than or equal to some number k .

Step 3: To find the value of k , we need to solve the inequality $10n + 5 \leq 15n$. **This inequality helps us determine the smallest value of n for which the inequality is true.** By solving the inequality, we can find the value of k .

$10n + 5 \leq 15n$ Subtracting $10n$ from both sides:
 $5 \leq 5n$ Dividing both sides by 5: $1 \leq n$

From this, we can see that for any value of n that is greater than or equal to 1, the inequality $10n + 5 \leq 15n$ holds true.

So, in this case, we have found that $k = 1$. This means that for all values of n that are 1 or larger ($n \geq 1$), the function $f(n) = 10n + 5$ is less than or equal to 15 multiplied by $g(n) = n$.

Step 4: Since the inequality is true for $n = 1$, we can say that for all values of n that are 1 or larger ($n \geq 1$), the function $f(n) = 10n + 5$ is less than or equal to 15 multiplied by $g(n)$.

Conclusion: So we have found our constants: $c = 15$ and $k = 1$. This means that for all values of n that are 1 or larger, the function $f(n) = 10n + 5$ is "smaller" or "equal to" 15 multiplied by $g(n) = n$.

In simpler terms, we have shown that as n gets bigger, the function $f(n) = 10n + 5$ is not much larger than $g(n) = n$. This comparison helps us understand how the two functions grow and how their values relate to each other.

THE CONSTANTS K AND C

The constants " k " and " c " are used to establish the relationship between two functions when analyzing their growth rates using Big O notation. Here's what each constant represents:

" k ": The constant " k " represents the starting point or threshold value from which the relationship between the functions holds true. It signifies the

input size from which we can say that one function's growth rate is no greater than the other.

"c": The constant "c" represents a scaling factor or a value that helps determine the upper bound of the growth rate. It allows us to establish an upper limit on the resources (time or space) required by an algorithm.

When using Big O notation to compare functions, we want to find values for "k" and "c" that satisfy the condition $f(n) \leq c * g(n)$ for all $n \geq k$. In other words, for any input size n greater than or equal to "k", the function f(n) will be less than or equal to "c" multiplied by the function g(n).

By finding suitable values for "k" and "c" and proving this inequality, we can determine the relationship between the growth rates of the two functions and analyze their efficiency or scalability. These constants help us quantify and compare the performance of algorithms or functions based on their growth rates.

HOW TO FIND THE VALUE OF C ?

To find the value of the constant "c" in Big O notation, you typically need to analyze the behavior of the functions and determine an upper bound on the growth rate. Here are a few general steps to help find the value of "c":

Identify the relevant functions: Determine which functions you want to compare and analyze in terms of their growth rates.

Write down the inequality: Write the inequality $f(n) \leq c * g(n)$ for all $n \geq k$, where f(n) and g(n) represent the functions you are comparing, and k is a threshold value.

Simplify the inequality: Simplify the inequality to isolate the terms involving the functions. **This step usually involves canceling out common factors or rearranging terms to make the inequality easier to work with.**

Determine an upper bound: Look for an upper bound on the growth rate by finding the highest power of n or the term that dominates the

growth of the function. This term will determine the upper limit of the growth rate.

Choose a value for c: Once you have determined the upper bound, you can choose a suitable value for "c" that satisfies the inequality. This value should be greater than or equal to the upper bound to ensure that the inequality holds for all values of n greater than or equal to k.

Test the inequality: Substitute the chosen value of "c" back into the inequality and verify that it holds true for all values of n greater than or equal to k. If the inequality is satisfied, you have found a valid value for "c".

Remember that finding the exact value of "c" is not always necessary in Big O notation. The focus is on establishing an upper bound and showing that a constant "c" exists for which the inequality holds.

Example :- $f(n) = 3n^2 + 4n + 1$. Show that $f(n) = O(n^2)$.

Introduction: We want to show that the function $f(n) = 3n^2 + 4n + 1$ is "smaller" or "equal to" the function $g(n) = n^2$ when n gets bigger. We'll do this by finding some numbers that help us compare the two functions.

Step 1: To show that f(n) is $O(n^2)$, we need to establish that for any value of n greater than or equal to a certain number k, the function f(n) is less than or equal to a constant c multiplied by g(n).

Step 2: Let's break down the inequality and simplify it step by step. We start by comparing the individual terms and inequalities involved.

$4n \leq 4n^2$ for all $n \geq 1$: This step shows that for any value of n greater than or equal to 1, the inequality $4n \leq 4n^2$ holds true. In other words, the term 4n is always smaller or equal to $4n^2$ when n is 1 or larger.

$1 \leq n^2$ for all $n \geq 1$: This step establishes that for any value of n greater than or equal to 1, the inequality $1 \leq n^2$ holds true. This inequality tells us that n^2 is always greater than or equal to 1 when n is 1 or larger.

Step 3: Now that we have these two inequalities, we can combine them to form a new inequality:

$$3n^2 + 4n + 1 \leq 3n^2 + 4n^2 + n^2 \text{ for all } n \geq 1.$$

Simplifying the inequality, we get:

$$3n^2 + 4n + 1 \leq 8n^2 \text{ for all } n \geq 1.$$

This shows that for any value of n greater than or equal to 1, the function $f(n) = 3n^2 + 4n + 1$ is less than or equal to 8 multiplied by $g(n) = n^2$.

Step 4: By choosing $c = 8$ and $k = 1$, we have shown that for all values of n greater than or equal to 1, the function $f(n) = 3n^2 + 4n + 1$ is "smaller" or "equal to" 8 multiplied by $g(n) = n^2$.

In simpler terms, we have demonstrated that as n gets larger, the function $f(n)$ grows no faster than $g(n)$. This comparison helps us understand the growth rates of the two functions and how they relate to each other.

Example :- $F(n) = 5n + 6$ and $g(n) = n$, show that $f(n)$ is $O(g(n))$, to show that $f(n)$ is $O(g(n))$?

Introduction: We want to show that the function $f(n) = 5n + 6$ is "smaller" or "equal to" the function $g(n) = n$ when n gets larger. We'll do this by finding some numbers that help us compare the two functions.

Step 1: To show that $f(n)$ is $O(g(n))$, we need to establish that for any value of n greater than or equal to a certain number k , the function $f(n)$ is less than or equal to a constant c multiplied by $g(n)$.

Step 2: Let's compare the individual terms and inequalities involved.

$5n \leq 5n$ for all $n \geq 1$: This inequality tells us that for any value of n greater than or equal to 1, the term $5n$ is always equal to $5n$. It does not change in relation to the value of n .

$6 \leq n$ for all $n \geq 6$: This inequality states that for any value of n greater than or equal to 6, the term 6 is less than or equal to n . This means

that once n becomes 6 or larger, the value of 6 is always smaller than or equal to n .

Step 3: Combining these inequalities, we can form the following inequality:

$$5n + 6 \leq 5n + n \text{ for all } n \geq 6.$$

Simplifying the inequality, we get:

$$5n + 6 \leq 6n \text{ for all } n \geq 6.$$

This shows that for any value of n greater than or equal to 6, the function $f(n) = 5n + 6$ is less than or equal to 6 multiplied by $g(n) = n$.

Step 4: By choosing $c = 6$ and $k = 6$, we have shown that for all values of n greater than or equal to 6, the function $f(n) = 5n + 6$ is "smaller" or "equal to" 6 multiplied by $g(n) = n$.

In simpler terms, we have demonstrated that as n gets larger, the function $f(n)$ grows no faster than $g(n)$. This comparison helps us understand the growth rates of the two functions and how they relate to each other.

BUT WE CAN ALSO DO THIS

Introduction: We want to show that the function $f(n) = 5n + 6$ is "smaller" or "equal to" the function $g(n) = n$ when n gets larger. In other words, we want to demonstrate that the growth rate of $f(n)$ is no faster than the growth rate of $g(n)$ for sufficiently large values of n .

Detailed Explanation:

Step 1: To show that $f(n)$ is $O(g(n))$, we need to establish that for any value of n greater than or equal to a certain number k , the function $f(n)$ is less than or equal to a constant c multiplied by $g(n)$.

Step 2: Let's start by comparing the individual terms and inequalities involved.

$5n + 6 \leq 11n$ for all $n \geq 1$: This inequality tells us that for any value of n greater than or equal to 1, the term $5n + 6$ is less than or equal to $11n$. We want to find a value of n (which we will call k) from which this inequality holds true.

Step 3: Simplifying the inequality, we get:

$6 \leq 6n$ for all $n \geq 1$.

This inequality indicates that for any value of n greater than or equal to 1, the constant term 6 is less than or equal to $6n$.

Step 4: By examining this inequality, we can see that it is true for $n = 1$. When $n = 1$, we have $6 \leq 6$. Therefore, we can say that for any value of n greater than or equal to 1, the function $f(n) = 5n + 6$ is less than or equal to $11n$.

Step 5: By choosing $c = 11$ and $k = 1$, we have shown that for all values of n greater than or equal to 1, the function $f(n) = 5n + 6$ is "smaller" or "equal to" 11 multiplied by $g(n) = n$.

In simpler terms, we have demonstrated that as n gets larger, the function $f(n)$ grows no faster than $g(n)$. This comparison helps us understand the growth rates of the two functions and how they relate to each other.

Chapter 3

Simple Sorting and Searching Algorithms

Sorting and searching algorithms are fundamental concepts in computer science and are widely studied for several reasons:

Common and Useful Operations: Sorting and searching are pervasive operations in various computer systems and applications. They are essential for managing and organizing data effectively. Sorting algorithms arrange data elements in a specific order, such as ascending or descending, making it easier to search and retrieve information. Searching algorithms help locate specific items or determine their absence in a collection of data. These operations are widely used in applications such as data processing, database management, information retrieval, e-commerce, networking, and more.

Efficient sorting and searching algorithms play a crucial role in optimizing the performance of these systems. They allow for faster access to information, improved data organization, and enhanced user experience.

Time Efficiency: The efficiency of sorting and searching algorithms has a direct impact on the time required to perform these operations. In scenarios involving large datasets or time-critical applications, the speed of sorting and searching becomes crucial. Efficient algorithms significantly reduce the time complexity and execution time, enabling faster and more responsive applications.

As computers continue to handle increasingly large and complex datasets, the need for efficient algorithms becomes even more critical. Optimized sorting and searching algorithms can reduce processing time, improve system throughput, and enhance overall performance.

Algorithmic Problem-Solving: Sorting and searching algorithms serve as fundamental building blocks for algorithmic problem-solving. Many complex problems can be solved by utilizing sorting and searching techniques as subroutines or components. For example, tasks such as finding the maximum or minimum value, identifying duplicate elements, or grouping similar items often involve sorting and searching operations.

By studying and understanding sorting and searching algorithms, individuals develop problem-solving skills and gain insights into algorithmic thinking. These skills can be applied to a wide range of computational problems, enabling the development of efficient solutions and algorithms.

Moreover, sorting and searching algorithms provide a foundation for learning and analyzing more advanced algorithms, data structures, and optimization techniques. They form the basis for exploring topics like graph algorithms, dynamic programming, and algorithmic analysis.

Sorting and searching algorithms are fundamental concepts in computer science due to their widespread use, impact on time efficiency, and contribution to algorithmic problem-solving skills. They are essential tools for organizing data, improving system performance, and developing efficient solutions to computational problems.

Simple Searching Algorithms

a) Linear Search: Sequential search, also known as linear search, is a simple searching algorithm that scans through a list of elements one by one until the desired element is found or the entire list is traversed. It starts from the beginning of the list and compares each element with the target key until a match is found. If a match is found, the algorithm returns the index of the element. If the entire list is traversed without finding a match, the algorithm returns -1 to indicate that the key is not present in the list.

Algorithm Steps:

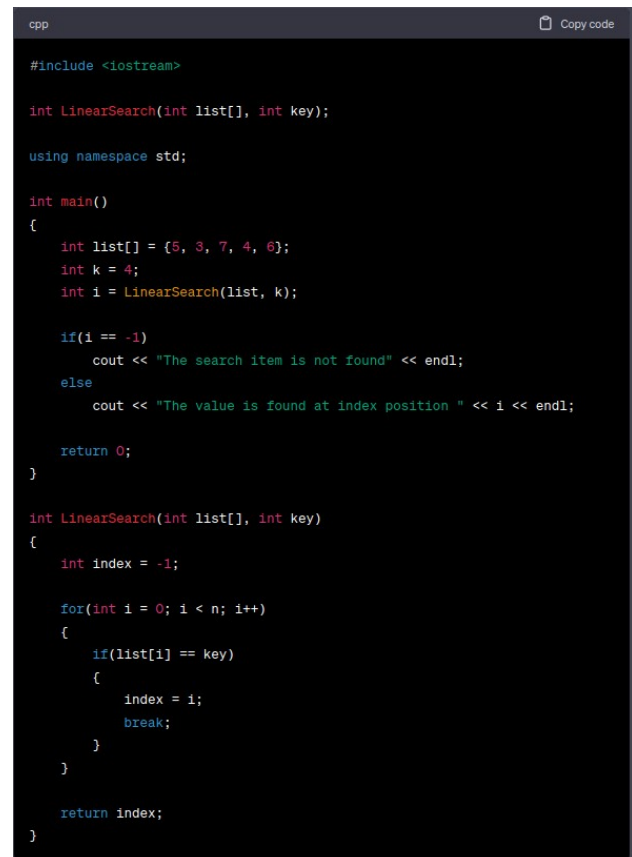
1. Start at the top (beginning) of the list.
2. Compare the element at the current position with the target key.
3. If the current element matches the target key, the search terminates, and the index of the element is returned.
4. If the current element does not match the target key, move to the next element in the list.
5. Repeat steps 2-4 until a match is found or the end of the list is reached.
6. If the entire list has been traversed without finding a match, return -1 to indicate that the key is not present in the list.

Linear search is considered the most natural and straightforward way of searching because it follows a step-by-step approach. It is easy to understand and implement, making it suitable for small lists or unsorted data.

Time Complexity: In the worst-case scenario, when the target element is not present or located at the end of the list, linear search needs to traverse the entire list, resulting in a time complexity of $O(n)$, where n is the number of elements in the list. The time complexity grows linearly with the size of the list.

Linear search is efficient for small lists or when the list is not sorted. However, for larger lists or when the list is sorted, more efficient search algorithms like binary search or hash-based

algorithms should be used to reduce the search time.



```
cpp Copy code
#include <iostream>

int LinearSearch(int list[], int key);

using namespace std;

int main()
{
    int list[] = {5, 3, 7, 4, 6};
    int k = 4;
    int i = LinearSearch(list, k);

    if(i == -1)
        cout << "The search item is not found" << endl;
    else
        cout << "The value is found at index position " << i << endl;

    return 0;
}

int LinearSearch(int list[], int key)
{
    int index = -1;

    for(int i = 0; i < n; i++)
    {
        if(list[i] == key)
        {
            index = i;
            break;
        }
    }

    return index;
}
```

In this code, we have implemented the Linear Search algorithm. Here's how it works:

1. We have an array of numbers called list, and we want to search for a specific value called k (in this case, 4).
2. The main function calls the Linear Search function and passes the list and k as arguments. It stores the returned index in the variable i.
3. The Linear Search function takes the list and the key as inputs and performs the linear search algorithm.
4. Linear search works by checking each element in the list one by one, starting from the first element, until the target value is found or the end of the list is reached.
5. The function uses a loop to iterate through each element in the list.

6. It compares each element with the target value using the condition `list[i] == key`.
7. If a match is found, the index of that element is stored in the variable `index` and the loop is exited using the `break` statement.
8. If the loop completes without finding a match, the value of `index` remains `-1`, indicating that the target value is not found in the list.
9. Finally, the function returns the index value, either the index of the found element or `-1` if the element is not found.
10. In the main function, we check the value of `i` returned by the Linear Search function. If it's `-1`, we print a message indicating that the value is not found. Otherwise, we display the index position where the value is found.

Linear Search is a simple and straightforward search algorithm. It checks each element in the list until it finds the desired value. While it works for any type of list, it may not be the most efficient for large lists. Other search algorithms like Binary Search are more efficient for sorted lists.

b) Binary Search: Binary search is a more efficient searching algorithm applicable to sorted lists. It follows a divide-and-conquer approach. It compares the target key with the middle element of the sorted list and recursively narrows down the search range by half based on whether the target key is smaller or larger than the middle element. This process continues until the target key is found or the search range is exhausted. Binary search is particularly efficient for large lists as it eliminates half of the remaining elements at each step, resulting in a logarithmic time complexity.

These simple searching algorithms serve as the foundation for more advanced and optimized search algorithms. By understanding and analyzing these algorithms, we can gain insights into the basic principles of searching and lay the

groundwork for developing more efficient searching techniques.

Binary search is a searching algorithm that is efficient for searching elements in a sorted list or array. It follows a divide and conquer strategy to repeatedly divide the search space in half and narrow down the search range until the desired element is found or the search space is exhausted.

Algorithm Steps:

1. The binary search algorithm assumes that the input list is sorted in ascending order. If the list is not sorted, a pre-processing step may be required to sort the data.
2. Start by defining the search range, which initially includes the entire sorted list.
3. Compare the target key with the element in the middle of the current search range.
4. If the middle element is equal to the target key, the search terminates, and the index of the element is returned.
5. If the target key is greater than the middle element, eliminate the lower half of the search range and continue searching in the upper half.
6. If the target key is smaller than the middle element, eliminate the upper half of the search range and continue searching in the lower half.
7. Repeat steps 3-6 until either the target element is found or the search range is reduced to a single element.
8. If the search range is reduced to a single element and it does not match the target key, the search terminates, and `-1` is returned to indicate that the key is not present in the list.

Binary search leverages the fact that the list is sorted to efficiently narrow down the search space by eliminating half of the remaining elements in each iteration. This reduces the search time significantly compared to linear search for large datasets.

Time Complexity: Binary search has a time complexity of $O(\log n)$, where n is the number of

elements in the sorted list. Since the search space is halved in each iteration, the time complexity grows logarithmically with the size of the input. This makes binary search highly efficient, particularly for large lists.

Binary search is like finding a word in a dictionary. Imagine you have a sorted dictionary, and you want to find a specific word. In terms of time complexity, it means that the time it takes to find the word grows very slowly as the dictionary size increases. For example, if you have 1000 pages in the dictionary, it would take around 10 steps to find the word. If you have 1,000,000 pages, it would take around 20 steps. The number of steps needed grows logarithmically with the size of the input. With each step, you cut the number of remaining pages in half. For example, if you start with 1000 pages, you roughly eliminate half of them with each step. In the first step, you have 1000 pages.

After one step, you have around 500 pages remaining. After two steps, you have around 250 pages, and so on. As you can see, the number of remaining pages reduces quickly as you keep dividing the search space in half. This is why binary search is efficient. Even with a large dictionary of 1,000,000 pages, it would take around 20 steps to find the word because you are repeatedly cutting the remaining pages in half.

Binary search's time complexity of $O(\log n)$ means that the time it takes to find an element grows very slowly as the size of the input (number of pages or items) increases. With each step, you roughly eliminate half of the remaining possibilities. This makes binary search a highly efficient algorithm for finding elements in sorted lists or arrays, even for large datasets

In this code, we have implemented the Binary Search algorithm. Here's how it works:

```
#include <iostream>

int BinarySearch(int list[], int key);

using namespace std;

int main()
{
    int list[] = {15, 23, 47, 54, 76};
    int k = 54;
    int i = BinarySearch(list, k);

    if(i == -1)
        cout << "The search item is not found" << endl;
    else
        cout << "The value is found at index position " << i << endl;

    return 0;
}

int BinarySearch(int list[], int key)
{
    int found = 0;
    int index = 0;
    int top = n - 1; // n is the number of elements in the list
    int bottom = 0;
    int middle;

    do
    {
        middle = (top + bottom) / 2;

        if(key == list[middle])
            found = 1;
        else
        {
            if(key < list[middle])
                top = middle - 1;
            else
                bottom = middle + 1;
        }
    } while(found == 0 && top >= bottom);

    if(found == 0)
        index = -1;
    else
        index = middle;

    return index;
}
```

1. We have an array of numbers called list, and we want to search for a specific value called k (in this case, 54).
2. The main function calls the BinarySearch function and passes the list and k as arguments. It stores the returned index in the variable i.
3. The BinarySearch function takes the list and the key as inputs and performs the binary search algorithm.
4. Binary search works by repeatedly dividing the sorted list in half and comparing the middle element with the target value. Based on this comparison, it either continues searching in the left or right half of the list.
5. The function uses variables like top and bottom to keep track of the boundaries of the current search range.
6. It calculates the middle index as the average of top and bottom and compares the value at that index with the target value.

7. If the target value is found, the variable found is set to 1. If not, it adjusts the search range by updating top or bottom accordingly.
8. The loop continues until the target value is found or the search range is exhausted (when top becomes less than bottom).
9. Finally, the function returns the index of the target value. If the value is not found, it returns -1.
10. In the main function, we check the value of i returned by the BinarySearch function. If it's -1, we print a message indicating that the value is not found. Otherwise, we display the index position where the value is found.

Binary Search is an efficient search algorithm for sorted lists. It divides the search range in half with each comparison, making it much faster than linear search for large sorted lists.

Sorting Algorithms

Sorting is the process of arranging a list of items or elements in a specific order, either increasing or decreasing. Sorting is a fundamental problem in computer science because it helps us find and access data more efficiently.

Bubble Sorting

Imagine you have a group of numbers that you want to arrange in order from smallest to largest. Bubble Sort helps you do that.

Here's how it works:

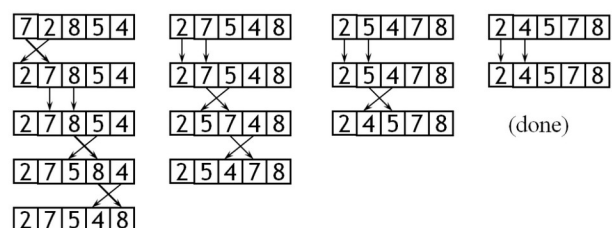
1. Start with an unsorted list of numbers.
2. Compare the first two numbers in the list. If the first number is bigger than the second number, you swap their positions. If not, you leave them as they are.
3. Move to the next pair of numbers and repeat the comparison and swapping

process. Keep doing this until you reach the end of the list.

4. Now, the biggest number in the list will have "bubbled" up to the end. It's like the biggest bubble floating to the top.
5. Repeat steps 2 to 4 for the remaining unsorted numbers, but this time excluding the last number you already sorted (since it's already in the correct position).
6. Keep repeating steps 2 to 5 until the entire list is sorted, meaning there are no more swaps needed.

By repeating the process of comparing and swapping adjacent numbers, Bubble Sort gradually moves the bigger numbers towards the end of the list. This sorting technique is called "Bubble Sort" because the larger numbers bubble up to their correct positions, just like bubbles rising to the surface of water.

Bubble Sort is a simple and easy-to-understand sorting algorithm. However, it may not be the most efficient for large lists because it requires multiple passes and comparisons. There are other sorting algorithms, like Merge Sort or Quick Sort, that are more efficient for bigger sets of numbers. But for smaller lists or learning purposes, Bubble Sort is a good starting point to understand the concept of sorting.



Imagine you have a group of numbers lined up. When we do the Bubble Sort, we look at two numbers at a time. If the first number is bigger than the second number, we swap their positions, just like swapping places in a game. We keep doing this for all the numbers in the line.

Each time we do a swap, it's like the bigger number is floating up to the top of the line, just

like a big bubble rising to the surface. This happens again and again until the biggest number reaches the very end of the line.

So, the idea is that as we go through the numbers and do these swaps, the biggest number keeps moving up, just like bubbles floating to the top. By the end, all the numbers will be sorted from smallest to largest.

Bubble Sort is like playing a game where you compare two numbers, and if they're in the wrong order, you switch their places. This process repeats until all the numbers are sorted, with the biggest one "bubbling" up to the end.

```
cpp
Copy code

#include <iostream>
using namespace std;

void BubbleSort(int list[]);

int list[] = {5, 3, 7, 4, 6};

int main()
{
    cout << "The values before sorting are: \n";
    for(int i = 0; i < 5; i++)
        cout << list[i] << " ";

    BubbleSort(list);

    cout << endl;
    cout << "The values after sorting are: \n";
    for(int i = 0; i < 5; i++)
        cout << list[i] << " ";

    return 0;
}

void BubbleSort(int list[])
{
    int temp;
    int n = 5;

    for (int i = n-2; i >= 0; i--)
    {
        for(int j = 0; j <= i; j++)
        {
            if (list[j] > list[j+1])
            {
                temp = list[j];
                list[j] = list[j+1];
                list[j+1] = temp;
            }
        }
    }
}
```

TIME COMPLEXITY OF BUBBLE SORTING

The time complexity of bubble sort is $O(n^2)$, where n is the number of elements in the array being sorted. In simpler terms, the time complexity represents how the execution time of

the algorithm grows as the size of the input (the array) increases.

Bubble sort works by repeatedly comparing adjacent elements and swapping them if they are in the wrong order. In the worst case scenario, where the input array is in reverse order, bubble sort needs to perform $n-1$ passes to fully sort the array. On each pass, it compares adjacent elements and swaps them if necessary.

To determine the time complexity, we consider the number of comparisons and swaps that occur. On each pass, bubble sort compares $n-1$ pairs of adjacent elements. Therefore, in the worst case, it performs $(n-1) + (n-2) + \dots + 1$ comparisons, which can be simplified to $(n*(n-1))/2$ or approximately $(n^2)/2$ comparisons.

Since the number of comparisons is proportional to n^2 , we can say that the time complexity of bubble sort is $O(n^2)$. This means that as the size of the input array doubles, the number of operations required increases by a factor of four.

Bubble sort is generally considered inefficient for large arrays or datasets, as its time complexity grows significantly with the input size. Other sorting algorithms, such as merge sort or quicksort, offer better time complexity and performance in most cases.

Selection Sorting

Selection sort is a simple sorting algorithm that works by repeatedly finding the minimum element from the unsorted part of the list and placing it at the beginning of the sorted part. It divides the list into two subarrays: the sorted subarray and the unsorted subarray.

To understand selection sort, let's consider a list of numbers that we want to sort in ascending order :-

[5, 2, 9, 1, 3]

Here's how the algorithm works step by step :-

1. Start with the initial list [5, 2, 9, 1, 3].
2. Find the minimum element in the unsorted subarray (2 in this case).

3. Swap the minimum element with the first element of the unsorted subarray, resulting in [2, 5, 9, 1, 3]. Now, the first element is part of the sorted subarray, and the rest are in the unsorted subarray.
4. Move to the next position of the unsorted subarray and repeat steps 2 and 3.
5. Find the minimum element in the remaining unsorted subarray (1 in this case).
6. Swap the minimum element with the first element of the unsorted subarray, resulting in [1, 5, 9, 2, 3]. Now, the first two elements are part of the sorted subarray.
7. Repeat steps 4-6 until the entire list is sorted.

Selection sort works by continuously finding the minimum element from the unsorted subarray and placing it at the beginning of the sorted subarray. This process continues until the entire list is sorted.

Selection sort has a time complexity of $O(n^2)$, where n is the number of elements in the list. It requires $n-1$ passes through the list, each time finding the minimum element from the remaining unsorted sub array. However, unlike other sorting algorithms, selection sort always performs the same number of comparisons regardless of the initial order of the elements.

While selection sort is not the most efficient sorting algorithm for large lists, it has the advantage of simplicity and ease of implementation. It is suitable for small lists or situations where the number of elements is limited.

Defining it simply

Imagine you have a group of numbers that you want to arrange in order from smallest to largest. Selection Sort helps you do that.

Here's how it works:

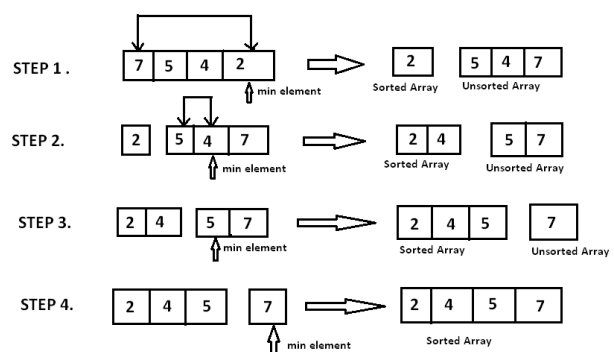
1. Start with an unsorted list of numbers.
2. Find the smallest number in the list and put it in the first position. This is like

finding the tiniest toy in a collection and putting it at the beginning.

3. Now, look for the second smallest number from the remaining unsorted numbers and put it in the second position. It's like finding the next tiniest toy and placing it right after the first one.
4. Keep repeating this process, finding the next smallest number and putting it in its correct position, until the entire list is sorted.
5. By the end, the numbers will be arranged in order from smallest to largest, just like a line of toys sorted from tiniest to largest.

Selection Sort works by repeatedly finding the smallest number from the unsorted part of the list and swapping it with the element in the correct position. It selects the smallest number and puts it in its rightful place, hence the name "Selection Sort."

Selection Sort is a simple sorting algorithm that is easy to understand. However, it may not be the most efficient for large lists because it requires multiple comparisons and swaps. There are other sorting algorithms, like Merge Sort or Quick Sort, that are more efficient for larger sets of numbers. But for smaller lists or learning purposes, Selection Sort is a good starting point to understand the concept of sorting.



Insertion sorting

Imagine you have a bunch of playing cards that are all mixed up, and you want to put them in order from smallest to largest. Insertion Sort helps you do that.

Here's how it works:

1. Start with an empty hand and spread out the cards facing down on the table.
2. Pick up one card with your empty hand. This card represents the first element of the sorted part of the cards.
3. Look at the next card on the table. If it's smaller than the card in your hand, place it to the left of that card. If it's bigger, put it to the right.
4. Now, pick up the third card and compare it with the two cards in your hand. If it's smaller than the left card, move the left card to the right, creating space for the new card. If it's between the two cards, insert it in between. If it's larger than both cards, put it to the right.
5. Keep doing this process for each new card, comparing it with the cards in your hand and finding its correct spot. If a card is smaller than the left card, keep moving the left card to the right until you find the right position.
6. Repeat this step for all the remaining cards on the table, one by one.
7. By the end, you will have sorted all the cards from smallest to largest, just like a neat deck of cards.

Insertion Sort works by building a sorted part of the cards gradually. You start with one card and compare each new card with the sorted cards in your hand. By finding the right spot for each new card, you create a sorted portion of the cards.

```
cpp Copy code

#include <iostream>
using namespace std;

void selectionSort(int list[]);

int list[] = {5, 3, 7, 4, 6};

int main()
{
    cout << "The values before sorting are: \n";
    for(int i = 0; i < 5; i++)
        cout << list[i] << " ";

    selectionSort(list);

    cout << endl;
    cout << "The values after sorting are: \n";
    for(int i = 0; i < 5; i++)
        cout << list[i] << " ";

    return 0;
}

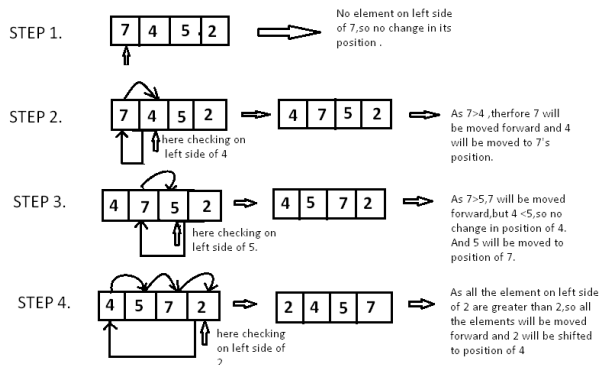
void selectionSort(int list[])
{
    int n = 5;
    int minIndex, temp;

    for (int i = 0; i <= n - 2; i++)
    {
        minIndex = i;

        for (int j = i + 1; j <= n - 1; j++)
        {
            if (list[j] < list[minIndex])
            {
                minIndex = j;
            }
        }

        if (minIndex != i)
        {
            temp = list[i];
            list[i] = list[minIndex];
            list[minIndex] = temp;
        }
    }
}
```

Insertion Sort is like organizing cards on a table, moving them around and finding the correct place for each card until they are all in order. It's a simple and intuitive sorting algorithm, but it might take more time for larger sets of cards.



```

cpp
Copy code

#include <iostream>
using namespace std;

void InsertionSort(int list[]);

int list[] = {5, 3, 7, 4, 6};

int main()
{
    cout << "The values before sorting are: \n";
    for(int i = 0; i < 5; i++)
        cout << list[i] << " ";

    InsertionSort(list);

    cout << endl;
    cout << "The values after sorting are: \n";
    for(int i = 0; i < 5; i++)
        cout << list[i] << " ";

    return 0;
}

void InsertionSort(int list[])
{
    int n = 5;

    for (int i = 1; i <= n - 1; i++)
    {
        for(int j = i; j >= 1; j--)
        {
            if (list[j-1] > list[j])
            {
                int temp = list[j];
                list[j] = list[j-1];
                list[j-1] = temp;
            }
            else
            {
                break;
            }
        }
    }
}

```

CHAPTER 4

LINKED LISTS

A linked list is a data structure used in algorithms to store and organize data. It consists of a sequence of nodes, where each node contains two components: **a data element and a reference (or link) to the next node in the sequence.**

Unlike arrays, linked lists do not require contiguous memory allocation. Each node can be allocated separately, and the references between nodes allow us to traverse the list efficiently.

Here are some key points about linked lists:

1. **Structure:** Each node in a linked list contains two parts: the data element, which stores the actual value or information, and a reference (or link) to the next node in the sequence.
2. **Head:** The first node in the linked list is called the head. It serves as the starting point for accessing the list.
3. **Tail:** The last node in the linked list is called the tail. It marks the end of the list and typically contains a reference to null or a special value indicating the end.

4. **Connections:** Nodes in a linked list are connected through their references. The link in each node points to the next node in the sequence, forming a chain-like structure.
5. **Dynamic Size:** Linked lists can grow or shrink dynamically by adding or removing nodes. Unlike arrays, which have a fixed size, linked lists can be modified more easily.
6. **Types of Linked Lists:** There are different types of linked lists, such as singly linked lists, doubly linked lists, and circular linked lists. Each type has its own variations in terms of node connections and traversal.
7. **Advantages and Disadvantages:** Linked lists offer advantages such as efficient insertion and deletion at any position, flexibility in size, and dynamic memory allocation. However, they have slower access times compared to arrays and require additional memory for storing the links.

In algorithms, linked lists are commonly used in situations where frequent insertions or deletions are required, or when the size of the data is unknown or can change over time. They provide a flexible and efficient way to store and manipulate data elements in a sequence.

TYPES OF LINKED LIST

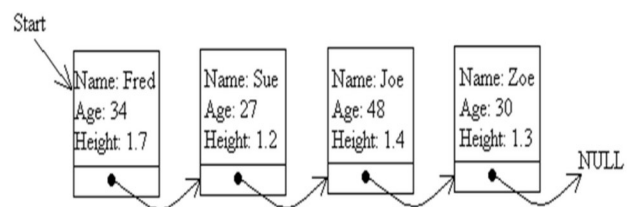
A singly linked list is a type of linked list where each node contains a data element and a reference (or link) to the next node in the sequence. It is called "singly" because it only allows traversal in one direction, from the head to the tail.

Key Points about Singly Linked List:

- **Structure:** Each node in a singly linked list has two components: the data element, which holds the value or

information, and a reference to the next node in the list.

- **Head:** The first node in the linked list is called the head. It serves as the starting point for accessing the list.
- **Tail:** The last node in the linked list is called the tail. It marks the end of the list and typically contains a reference to null or a special value indicating the end.
- **Connections:** Nodes in a singly linked list are connected through their next references. The next reference in each node points to the next node in the sequence.
- **Traversal:** To traverse a singly linked list, we start at the head and follow the next references until we reach the end (tail) or the reference becomes null.
- **Dynamic Size:** Singly linked lists can grow or shrink dynamically by adding or removing nodes. They do not require contiguous memory allocation like arrays, allowing for efficient insertion and deletion at any position.



Operation of single linked list

1) Adding a node to the end of a singly linked list:

- Start from the head of the linked list and traverse through the nodes until you reach the last node (the node with the next reference as NULL).
- Create a new node with the desired data.
- Set the next reference of the last node to point to the newly created node.
- Update the tail pointer to point to the newly added node if necessary.

2. Adding a node to the left of a specific data in a singly linked list:

- Start from the head of the linked list and traverse through the nodes until you find the node with the desired data.
- Create a new node with the desired data.
- Set the next reference of the newly created node to point to the node found in the previous step.
- Update the next reference of the node before the found node to point to the newly created node.

3. Adding a node to the right of a specific data in a singly linked list:

- Start from the head of the linked list and traverse through the nodes until you find the node with the desired data.
- Create a new node with the desired data.
- Set the next reference of the newly created node to point to the next node of the found node.
- Set the next reference of the found node to point to the newly created node.

4. Deleting a node from the end of a singly linked list:

- Start from the head of the linked list and traverse through the nodes until you reach the second-to-last node.
- Update the next reference of the second-to-last node to NULL, effectively removing the last node from the list.
- Update the tail pointer to point to the new last node if necessary.

5. Deleting a node from the front of a singly linked list:

- Update the head pointer to point to the second node, effectively removing the first node from the list.

6. Deleting any node using the search data from a singly linked list:

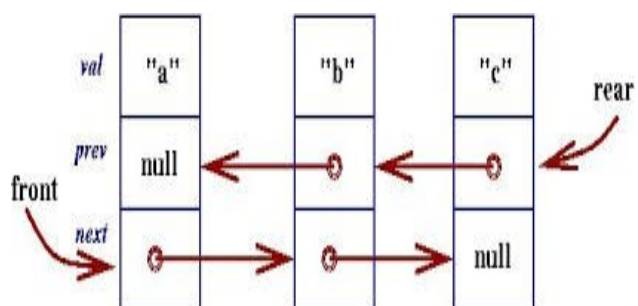
- Start from the head of the linked list and traverse through the nodes until you find the node with the desired data.
- Update the next reference of the previous node to point to the next node of the found node, effectively removing the found node from the list.

7. Displaying the nodes from the singly linked list in a forward manner:

- Start from the head of the linked list and traverse through the nodes one by one.
- Print or process the data of each node as you go.

These operations allow you to manipulate and work with the elements of a singly linked list, such as adding or removing nodes at different positions or displaying the list in a forward manner. Each operation involves updating the next references of the nodes to maintain the integrity and connectivity of the linked list.

Double Linked List :- A doubly linked list is a type of linked list where each node contains not only a reference to the next node (successor), but also a reference to the previous node (predecessor). This additional reference allows for easy traversal in both forward and backward directions.



Here's a brief description of the operations you mentioned for a doubly linked list:

1. Adding a node to the end of a doubly linked list:

- Start from the head of the linked list and traverse through the nodes until you reach the last node.

- Create a new node with the desired data.
- Set the next reference of the last node to point to the newly created node.
- Set the previous reference of the newly created node to point to the last node.
- Update the tail pointer to point to the newly added node if necessary.

2. Adding a node to the front of a doubly linked list:

- Create a new node with the desired data.
- Set the next reference of the new node to point to the current head node.
- Set the previous reference of the new node to NULL.
- Set the previous reference of the current head node to point to the new node.
- Update the head pointer to point to the new node.

3. Adding a node to the left of a specific data in a doubly linked list:

- Start from the head of the linked list and traverse through the nodes until you find the node with the desired data.
- Create a new node with the desired data.
- Set the next reference of the new node to point to the found node.
- Set the previous reference of the new node to point to the previous node of the found node.
- Set the next reference of the previous node to point to the new node.
- Set the previous reference of the found node to point to the new node.

4. Adding a node to the right of a specific data in a doubly linked list:

- Start from the head of the linked list and traverse through the nodes until you find the node with the desired data.
- Create a new node with the desired data.
- Set the next reference of the new node to point to the next node of the found node.
- Set the previous reference of the new node to point to the found node.
- Set the next reference of the found node to point to the new node.
- Set the previous reference of the next node to point to the new node.

5. Deleting a node from the end of a doubly linked list:

- Start from the tail of the linked list and traverse through the nodes until you reach the second-to-last node.
- Set the next reference of the second-to-last node to NULL, effectively removing the last node from the list.
- Update the tail pointer to point to the new last node if necessary.

6. Deleting a node from the front of a doubly linked list:

- Update the head pointer to point to the second node, effectively removing the first node from the list.
- Set the previous reference of the new head node to NULL.

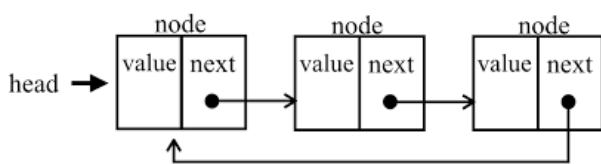
7. Deleting any node using the search data from a doubly linked list:

- Start from the head of the linked list and traverse through the nodes until you find the node with the desired data.
- Set the next reference of the previous node to point to the next

node of the found node, effectively removing the found node from the list.

- Set the previous reference of the next node to point to the previous node of the found node.

Circular Linked Lists :- A circular singly linked list is a type of linked list in which the last node of the list points back to the first node, forming a circular loop. This allows for continuous traversal from any node to any other node in the list.



Operations of a Circular Singly Linked List:

1. Adding a node to the end of a Circular singly linked list:

- Start from the head of the linked list and traverse through the nodes until you reach the last node.
- Create a new node with the desired data.
- Set the next reference of the last node to point to the newly created node.
- Set the next reference of the newly created node to point to the head node, completing the circular loop.

2. Adding a node to the left of a specific data in a Circular singly linked list:

- Start from the head of the linked list and traverse through the nodes until you find the node with the desired data.
- Create a new node with the desired data.
- Set the next reference of the new node to point to the found node.

- Set the next reference of the previous node of the found node to point to the new node.

3. Adding a node to the right of a specific data in a Circular singly linked list:

- Start from the head of the linked list and traverse through the nodes until you find the node with the desired data.
- Create a new node with the desired data.
- Set the next reference of the new node to point to the next node of the found node.
- Set the next reference of the found node to point to the new node.

4. Deleting a node from the end of a Circular singly linked list:

- Start from the head of the linked list and traverse through the nodes until you reach the second-to-last node.
- Set the next reference of the second-to-last node to point to the head node, effectively removing the last node from the list.

5. Deleting a node from the front of a Circular singly linked list:

- Update the head pointer to point to the second node, effectively removing the first node from the list.
- Set the next reference of the last node to point to the new head node.

6. Deleting any node using the search data from a Circular singly linked list:

- Start from the head of the linked list and traverse through the nodes until you find the node with the desired data.
- Set the next reference of the previous node to point to the next node of the found node,

effectively removing the found node from the list.

7. Displaying the nodes from the Circular singly linked list in a forward manner:

- Start from the head node and iterate through the nodes, printing or processing the data of each node until you reach the head node again.

The circular nature of the linked list ensures that traversal can continue indefinitely in a loop, allowing efficient access to any node in the list.

CHAPTER 5

STACKS AND QUEUES

A stack is a linear data structure that follows the Last-In, First-Out (LIFO) principle. It can be visualized as a stack of plates, where new plates are added to the top and removed from the top. In a stack, all insertions and deletions occur at one end, known as the top of the stack.

Basic Stack Operations:

1. **Push:** The push operation adds an element to the top of the stack. It places the new element on top of the existing elements, becoming the new top of the stack.
2. **Pop:** The pop operation removes and returns the element from the top of the stack. It retrieves the topmost element and removes it from the stack, exposing the element beneath it as the new top.
3. **Peek:** The peek operation retrieves the top element from the stack without removing it. It provides access to the topmost element without modifying the stack.

Push Operation: The push operation adds an item to the top of the stack. It follows these steps:

- Check if there is room in the stack to add a new item.

- If there is room, place the new item on top of the stack.
- If there is no room (the stack is full), give an error message to indicate that the stack is already at its maximum capacity.

In simple terms, pushing an item onto the stack is like adding a new book to the top of a stack of books. You can only add a book if there is space available. If the stack is full, you can't add any more books, and an error message is displayed.

Pop Operation: The pop operation removes the item from the top of the stack and returns its value. It follows these steps:

- Check if the stack is not empty (contains at least one item).
- If the stack is not empty, remove the item from the top of the stack.
- Return the value of the removed item.
- If the stack is empty, give an error message to indicate that there are no items in the stack.

In simpler terms, popping an item from the stack is like taking the top book from a stack of books. You can only remove a book if there are books in the stack. If the stack is empty, there are no books to remove, and an error message is displayed.

The push operation adds an item to the stack, while the pop operation removes and returns the top item. Both operations ensure that the stack remains in the Last-In, First-Out (LIFO) order.

Imagine you have a stack of books, where each book represents an element in the stack. The topmost book is the one you can easily see and access. Now, let's understand the peek operation:

Peek: The peek operation allows you to take a look at the topmost book in the stack without removing it. It gives you access to the element at the top of the stack.

Think of it like this:

- You have a stack of books placed one on top of the other.

- When you perform the peek operation, you can take a quick glance at the book on the top of the stack.
- You can read its title, check its details, or simply see what it is without taking it out from the stack.
- However, the book remains in its place, and the stack itself remains unchanged.

In programming terms, the peek operation retrieves the value of the top element of the stack without actually removing it. It allows you to access the data stored in the top element temporarily. This can be useful when you need to examine the topmost element or check its value without altering the stack's overall structure.

These operations are the basic building blocks of a stack and allow you to add and remove items in a specific order. They provide a way to manage data in a stack efficiently and effectively.

The key characteristics of a stack ADT are:

1. **Elements are stored in a specific order:** Stacks store elements based on the order of insertion, from the bottom to the top. The last element inserted is always at the top of the stack.
2. **Elements are added to the top:** New elements are added to the top of the stack, becoming the new top element.
3. **Only the top element can be accessed or removed:** Stacks allow access and removal of only the top element. Other elements in the stack remain inaccessible until the top element is removed.

What is the whole point of a stack

The main purpose of a stack is to provide a simple and efficient way to manage data based on the Last-In, First-Out (LIFO) principle. Stacks are used to solve various problems and are fundamental in computer science and programming. **Here are some key points about the purpose and benefits of using a stack:**

Undo/Redo Operations: Stacks can be used to implement undo and redo functionality in

applications. Each operation performed by the user can be stored on a stack, allowing them to go back to previous states by undoing operations or redoing them if needed.

Time complexity of a stack : The goal of a stack is to perform all operations in constant time, denoted as $O(1)$. This means that regardless of the number of elements in the stack, operations like push, pop, and peek should execute in constant time, making stacks efficient and straightforward to implement.

The goal of a stack is to make sure that all the operations you can do on a stack, like adding elements (push), removing elements (pop), and looking at the top element (peek), take the same amount of time no matter how many elements are in the stack. **This is called constant time.**

Imagine you have a stack of books. When you want to add a new book to the stack, you simply place it on top. This operation is called push. It doesn't matter if there are 10 books or 100 books in the stack; adding a new book will always take the same amount of time.

When you want to remove a book from the top of the stack, you take the top book and remove it. This operation is called pop. Again, whether you're removing the top book from a small stack or a large stack, it will always take the same amount of time.

Finally, when you want to peek at the top book without removing it, you just take a quick look. This operation is called peek. It also takes the same amount of time, regardless of how many books are in the stack.

The important thing to understand is that regardless of the size of the stack (the number of books), the time it takes to perform these operations remains the same.

This constant time makes stacks efficient and straightforward to work with because you don't have to worry about the size of the stack affecting the speed of the operations. It's a consistent and reliable way to manage data.

Array Implementation of a stack

In array implementation of a stack, we use a one-dimensional array to represent the stack. The array has a fixed size and each element of the array represents an item in the stack.

Here are the key components and operations involved in the array implementation of a stack:

1. **Array:** We create an array with a fixed size to hold the elements of the stack. The size of the array determines the maximum capacity of the stack.
2. **Top:** We use a variable called "top" to keep track of the index of the topmost element in the stack. Initially, when the stack is empty, the top is set to -1.
3. **Push Operation:** To add an element to the stack (push operation), we increment the value of the top variable and then assign the new element to the array at the index specified by the top.
4. **Pop Operation:** To remove an element from the stack (pop operation), we first check if the stack is empty by verifying if the top is -1. If the stack is not empty, we retrieve the element from the array at the index specified by the top, then decrement the value of the top variable.
5. **Peek Operation:** To get the top element of the stack without removing it (peek operation), we simply return the element from the array at the index specified by the top.
6. **Overflow Condition:** If we attempt to push an element into a full stack (i.e., the array is already at its maximum capacity), it results in an overflow condition, indicating that the stack is full and cannot accept more elements.
7. **Underflow Condition:** If we attempt to pop an element from an empty stack (i.e., the top is -1), it results in an underflow condition, indicating that the stack is empty and there are no elements to remove.

The array implementation of a stack provides a simple and efficient way to manage elements in a Last-In, First-Out (LIFO) manner. However, it has a fixed capacity, which means it can run into overflow or underflow conditions if not managed properly.

Application of stacks

Evaluation of Algebraic Expressions ?

Infix, prefix, and postfix are different notations or ways of representing mathematical expressions. They determine the order of operations and the placement of operators and operands within the expression.

Infix Notation: This is the most common way of writing mathematical expressions, where operators are placed between the operands. For example, " $2 + 3$ " is an infix expression. Infix notation can sometimes be ambiguous and requires the use of parentheses to indicate the desired order of operations.

Prefix Notation: In prefix notation, also known as Polish notation, operators are placed before their operands. For example, " $+ 2 3$ " is a prefix expression. This notation eliminates the need for parentheses as the order of operations is determined solely by the position of the operators.

Postfix Notation: In postfix notation, also known as Reverse Polish notation (RPN), operators are placed after their operands. For example, " $2 3 +$ " is a postfix expression. Similar to prefix notation, postfix notation eliminates the need for parentheses as the order of operations is determined by the position of the operators.

The conversion from infix to postfix notation is done to simplify the evaluation of mathematical expressions. Postfix notation has several advantages over infix notation:

1. **Elimination of Ambiguity:** Postfix notation removes the need for parentheses to indicate the order of operations. The position of the operators inherently

determines the order, making the expression unambiguous.

2. **Easy Evaluation:** Postfix notation lends itself well to evaluation using a stack or other data structures. The operands are pushed onto the stack, and when an operator is encountered, the necessary operands are popped from the stack and the result is pushed back. This simplifies the evaluation process.
3. **Operator Precedence:** In postfix notation, the precedence of operators is evident from their position. There is no need to rely on operator precedence rules as required in infix notation.

By converting infix expressions to postfix notation, we can easily and efficiently evaluate the expressions, simplify the computation, and remove ambiguity. It provides a more structured and unambiguous representation of mathematical expressions, facilitating their interpretation and calculation.

Conversion from infix to postfix

conversion. In infix notation, the operators are placed between the operands, while in postfix notation, the operators come after the operands. Converting an infix expression to postfix notation helps in evaluating the expression in a more organized way.

Here's how the conversion works:

1. We start with an empty stack and an empty postfix notation.
2. We scan the infix expression from left to right, one character at a time.
3. If we encounter an operand (such as A, B, C), we directly add it to the postfix notation.
4. If we encounter an operator, we follow certain rules to decide whether to push it onto the stack or add it to the postfix notation:
 - If the stack is empty or the top operator has lower precedence

than the current operator, we push the current operator onto the stack.

- If the top operator has higher or equal precedence to the current operator, we pop the top operator from the stack and add it to the postfix notation. Then we repeat this step until the top operator has lower precedence or the stack becomes empty. After that, we push the current operator onto the stack.
5. If we encounter a left parenthesis '(', we push it onto the stack.
 6. If we encounter a right parenthesis ')', we repeatedly pop operators from the stack and add them to the postfix notation until we encounter the matching left parenthesis '(' on top of the stack. Then we pop and discard the left parenthesis.
 7. After scanning the entire infix expression, we pop any remaining operators from the stack and add them to the postfix notation.

The resulting postfix notation is the converted form of the original infix expression. It eliminates the need for parentheses to indicate the order of operations, as the postfix notation inherently represents the order.

Exempl 1: $A + B * C - D / E$

Infix	Stack(Bottom->)	Postfix
$A + B * C - D / E$	Empty	Empty
$+ B * C - D / E$	Empty	A
$B * C - D / E$	+	A
$* C - D / E$	+	AB
$C - D / E$	+	AB
$- D / E$	+	ABC
$- D / E$	+	ABC*
$- D / E$	Empty	ABC*+
D / E	-	ABC*+
$/ E$	-	ABC*+D
E	- /	ABC*+D
Empty	- /	ABC*+DE
Empty	-	ABC*+DE/
Empty	Empty	ABC*+DE/-

Example :-

Exempl 2: $A * B - (C + D) + E$

Infix	Stack(Bottom->)	Postfix
$A * B - (C + D) + E$	Empty	Empty
$* B - (C + D) + E$	Empty	A
$B - (C + D) + E$	*	A
$- (C + D) + E$	*	AB
$- (C + D) + E$	Empty	AB*
$(C + D) + E$	-	AB*
$C + D) + E$	-(AB*
$+ D) + E$	-(AB*C
$D) + E$	-(+	AB*C
$) + E$	-(+	AB*CD
$+ E$	-	AB*CD+
E	--+	AB*CD+
Empty	--+	AB*CD+E
Empty	-	AB*CD+E+
Empty	Empty	AB*CD+E+-

Example :-

Exempl 3: $a + b * c + (d * e + f) * g$

Infix	Stack(Bottom->)	Postfix
$a + b * c + (d * e + f) * g$	Empty	Empty
$+ b * c + (d * e + f) * g$	Empty	a
$b * c + (d * e + f) * g$	+	a
$* c + (d * e + f) * g$	+	ab
$c + (d * e + f) * g$	+	ab
$+ (d * e + f) * g$	+	abc
$(d * e + f) * g$	+	abc*
$(d * e + f) * g$	Empty	abc*+
$(d * e + f) * g$	+	abc*+
$d * e + f) * g$	+(abc*+
$* e + f) * g$	+(abc*+d
$e + f) * g$	+(*	abc*+d
$+ f) * g$	+(*	abc*+de
$f) * g$	+(+	abc*+de*
$) * g$	+(+	abc*+de*f
$* g$	+	abc*+de*f+
g	+	abc*+de*f+
Empty	+	abc*+de*f+g
Empty	Empty	abc*+de*f+g+

Queue Abstract Data Type

A queue is a data structure that follows the First-In, First-Out (FIFO) principle. It behaves like a line of people waiting for a bank teller or customers waiting in a queue. In a queue, elements are stored in the order they were inserted, and the first element inserted is the first one to be removed.

Basic Properties of a Queue:

- Ordering:** Elements are stored in the order they were added, with new elements being added to the end (back) of the queue.
- Operations:** The key operations performed on a queue are:
 - Offer or Enqueue:** This operation adds an element to the back of the queue. It inserts a new element at the end, following the FIFO order.
 - Remove or Dequeue:** This operation removes and returns the element at the front of the queue. It retrieves the element that has been waiting in the queue the longest.
 - Peek (just for displaying purpose):** This operation returns the element at the front of the queue without removing it. It allows you to examine the element that will be removed next.
 - Other Operations:** Additional operations commonly available for queues include checking if the queue is empty, getting the size of the queue, and clearing all elements from the queue.

Goal: The goal of a queue is to ensure that every operation performed on the queue executes in constant time, denoted as $O(1)$. This means that regardless of the number of elements in the queue, operations like offer, remove, peek, isEmpty, size, and clear should be efficient and have consistent performance.

Queue Features:

- Ordering:** A queue maintains the order in which elements were added. New elements are added to the end of the queue, while removal happens from the front.

2. Operations Efficiency: All the fundamental operations of a queue, such as offer, remove, and peek, have a constant time complexity of $O(1)$. This makes queues efficient for managing and processing elements in a specific order.

The Queue Operations: When we think about a queue, we can imagine a line of people waiting for a bank teller, where the first person to arrive is the first to be served. The queue has two main reference points:

- **Front:** The front of the queue represents the element that has been waiting in the queue the longest. It is the element that will be removed next.
- **Rear:** The rear of the queue represents the end of the queue, where new elements are added.

Array Implementation Of A Queue

In the array implementation of a queue, a fixed-size array is used to store the elements of the queue. The front and rear indices keep track of the position of the front and rear elements, respectively.

Example: Let's consider an example of a queue implemented using an array with a fixed size of 5.

Initially, the queue is empty:

Front = -1 (no elements in the queue) Rear = -1 (no elements in the queue)

Enqueue Operation (Adding elements): When an element is enqueued (added) to the queue, it is inserted at the rear position and the rear index is incremented.

Let's enqueue the elements 10, 20, 30, and 40 in sequence:

Enqueue(10):

- Front = 0, Rear = 0
- Queue: [10, _, _, _, _]

Enqueue(20):

- Front = 0, Rear = 1
- Queue: [10, 20, _, _, _]

Enqueue(30):

- Front = 0, Rear = 2
- Queue: [10, 20, 30, _, _]

Enqueue(40):

- Front = 0, Rear = 3
- Queue: [10, 20, 30, 40, _]

Note that the front index remains at -1 since there are no elements removed yet.

Dequeue Operation (Removing elements): When an element is dequeued (removed) from the queue, it is done from the front position, and the front index is incremented.

Let's dequeue two elements from the queue:

Dequeue():

- Front = 1, Rear = 3
- Removed element: 10
- Queue: [_, 20, 30, 40, _]

Dequeue():

- Front = 2, Rear = 3
- Removed element: 20
- Queue: [_, _, 30, 40, _]

Note that the elements are removed from the front of the queue, and the front index is incremented to reflect the removal.

Enqueue and Dequeue Operations: The enqueue operation adds elements to the rear of the queue, while the dequeue operation removes elements from the front of the queue.

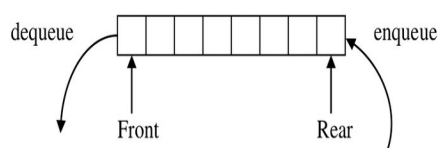
Enqueue Operation:

- When an element is enqueued, it is inserted at the rear position (Rear index is incremented).
- If the queue is full (Rear index reaches the maximum capacity), it means there is no room to add more elements.

Dequeue Operation:

- When an element is dequeued, it is removed from the front position (Front index is incremented).
- If the queue is empty (Front index equals Rear index), it means there are no elements to remove.

In the array implementation of a queue, the front and rear indices are used to keep track of the positions for enqueue and dequeue operations, ensuring that the elements are added to the rear and removed from the front in the correct order.



Example:

Operation	Content of queue
Enqueue(B)	B
Enqueue(C)	B, C
Dequeue()	C
Enqueue(G)	C, G
Enqueue (F)	C, G, F
Dequeue()	G, F
Enqueue(A)	G, F, A
Dequeue()	F, A

Note: This basic implementation of a queue using an array does not make efficient use of space because dequeued spaces are not reused. To overcome this, a more advanced data structure called a circular queue or ring buffer can be used. In a circular queue, the rear index wraps around to the beginning of the array when it reaches the end, effectively reusing the spaces of dequeued elements.

Array Implementation of a Circular Queue

A circular queue or ring buffer is an advanced data structure that overcomes the space inefficiency problem in the basic array implementation of a queue.

In a circular queue, the rear index wraps around to the start of the array when it reaches the end,

effectively reusing the positions of dequeued elements. This makes it a more efficient data structure in terms of space utilization.

Let's illustrate this with an example. Suppose we have a circular queue with a capacity of 5, represented as an array of size 5.

Circular Queue: [_, _, _, _, _]

- front: -1
- rear: -1

Now, let's enqueue three elements 10, 20, and 30.

Enqueue 10:

Circular Queue: [10, _, _, _, _]

- front: 0
- rear: 0

Enqueue 20:

Circular Queue: [10, 20, _, _, _]

- front: 0
- rear: 1

Enqueue 30:

Circular Queue: [10, 20, 30, _, _]

- front: 0
- rear: 2

Enqueue 40:

Circular Queue: [10, 20, 30, 40, _]

- front: 0
- rear: 3

Now, let's dequeue two elements

Dequeue:

Circular Queue: [_, 20, 30, 40, _]

- front: 1

- rear: 3

Dequeue:

Circular Queue: [_, _, 30, 40, _]

- front: 2
- rear: 3

Now, let's enqueue one element 50.

Enqueue 50:

Circular Queue: [_, _, 30, 40, 50]

- front: 2
- rear: 4

As you can see, when rear index reached the end of the array, it wrapped around to the beginning of the array. Now, if we enqueue another element, say 60, it will be placed at the first position, reusing the space of the dequeued elements.

Enqueue 60:

Circular Queue: [60, _, 30, 40, 50]

- front: 2
- rear : 1

It's important to note that in the array implementation, when the rear index reaches the end of the array, the next enqueue operation would overwrite the first elements of the queue if they have been dequeued. This is known as queue overflow. Similarly, if the front and rear indices are equal, any dequeue operation would result in underflow, indicating that the queue is empty.

Queue Overflow and Underflow

In the context of a circular queue or a ring buffer, overflow and underflow conditions are important to manage. They're used to handle scenarios when we try to add an element to a full queue (overflow) or try to remove an element from an empty queue (underflow).

Overflow:

Queue overflow happens when we try to enqueue an element into a queue that is already full. In the context of a circular queue, the queue is considered full when the next increment of rear (when incremented and modulus applied) is equal to front.

If this condition is true, it means that the queue is full and any attempt to add more elements into the queue would lead to queue overflow, as it would overwrite the existing elements.

Let's consider the following example:

Circular Queue: [60, 70, 30, 40, 50]

- front: 2
- rear: 1

If we try to enqueue another element, say 80, in this case, it would lead to an overflow because the queue is already full. The new element would overwrite the current front element (30), which would lead to loss of data and incorrect operation of the queue.

Underflow:

Queue underflow happens when we try to dequeue an element from a queue that is already empty. In the context of a circular queue, the queue is considered empty when front is equal to rear.

If front == rear, it means that the queue is empty and any attempt to remove an element from the queue would lead to queue underflow, as there are no elements to remove.

Let's consider the following example:

Circular Queue: [_, _, _, _, _]

- front: -1
- rear: -1

If we try to dequeue an element in this case, it would lead to an underflow because the queue is already empty.

To avoid these conditions, it's important to check whether the queue is full before performing an enqueue operation and to check whether the queue is empty before performing a dequeue operation.

Example: Consider a queue with MAX_SIZE = 4

Operation	Simple array				Circular array			
	Content of the array	Content of the Queue	QUEUE SIZE	Message	Content of the array	Content of the queue	QUEUE SIZE	Message
Enqueue(B)	B	B	1		B	B	1	
Enqueue(C)	B C	BC	2		B C	BC	2	
Dequeue()	C	C	1		C	C	1	
Enqueue(G)	C G	CG	2		C G	CG	2	
Enqueue(F)	C G F	CGF	3		C G F	CGF	3	
Dequeue()	G F	GF	2		G F	GF	2	
Enqueue(A)	G F	GF	2	Overflow	A G F	GFA	3	
Enqueue(D)	G F	GF	2	Overflow	A D G F	GFAD	4	
Enqueue(C)	G F	GF	2	Overflow	A D G F	GFAD	4	Overflow
Dequeue()	F	F	1		A D	FAD	3	
Enqueue(H)	F	F	1	Overflow	A D H F	FADH	4	
Dequeue()		Empty	0		A D H	ADH	3	
Dequeue()		Empty	0	Underflow	D H	DH	2	
Dequeue()		Empty	0	Underflow	H	H	1	
Dequeue()		Empty	0	Underflow		Empty	0	
Dequeue()		Empty	0	Underflow		Empty	0	Underflow

PRIORITY QUEUE

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with equal priorities are enqueued, they are served according to their ordering in the queue.

Generally, the value of the element itself is considered for assigning the priority. For example, The element with the highest value is considered the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element and the element with the highest value as the lowest priority element. We can also set priorities according to our needs. The priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree.

Example of a priority queue

Real-world example of a priority queue could be a hospital's emergency department. Patients with more severe conditions are prioritized over patients with less severe conditions.

Consider the following priority queue of patients waiting in the emergency department. The

number represents the severity of their condition (with a higher number indicating a more severe condition) :-

Patient 1	Patient 2	Patient 3	Patient 4	Patient 5
3	5	1	4	2

In this priority queue, Patient2 with condition severity 5 will be served first, followed by Patient4 with severity 4, then Patient1 with severity 3, and so on.

Operations on Priority Queue

- 1. Enqueue:** This operation inserts an element into the priority queue.
- 2. Dequeue:** This operation removes the highest priority element from the priority queue.
- 3. Peek or Top:** This operation returns the highest priority element in the priority queue without deleting it from the queue.
- 4. IsEmpty:** This operation checks if the priority queue is empty.
- 5. IsFull:** This operation checks if the priority queue is full.

Example :-

Let's consider a priority queue in the context of a task management system. Each task is assigned a priority: the higher the number, the higher the priority. The tasks with higher priority need to be executed before the tasks with lower priority.

Consider the following initial queue:

Task 1	Task 2	Task 3	Task 4	Task 5
2	5	1	3	4

The number associated with each task represents its priority

Enqueue Operation :- Let's enqueue a new tasks , "Task 6" with a priority of "3".

Enqueue (Task6, 3):

Task 1	Task 2	Task 3	Task 4	Task 5	Task 6
2	5	1	3	4	3

The queue is not yet ordered by priority. Many priority queue implementations ensure that the highest priority element is always at the front of the queue

Sorting the Queue:

Now, let's sort the queue by priority. In a real-world priority queue implementation, this would be taken care of automatically by the data structure (for example, a binary heap):

Task 2	Task 5	Task 4	Task 6	Task 1	Task 3
5	4	3	3	2	1

Dequeue Operation:

The dequeue operation removes the highest priority task from the queue. In case of a tie (like Task4 and Task6 both having a priority of 3), the task that was enqueued first is removed.

Dequeue ()

Task 5	Task 4	Task 6	Task 1	Task 3
4	3	3	2	1

Task2 with priority 5 was removed from the queue. The next dequeue operation would remove Task5 which has the next highest priority 4, and so on.

Merging and Demerging Queues

De-merging queues is a process where a given queue is split into two or more queues based on certain criteria. The criteria can be any condition like splitting odd and even elements into two separate queues or separating elements based on their priority or any other user-defined condition.

Let's take an example where we de-merge a queue into two separate queues: one queue containing the even numbers and the other queue containing the odd numbers.

Suppose we have the following queue:

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

Now, we want to de-merge this queue into two separate queues: one containing the even numbers and the other containing the odd numbers.

We start by initializing two empty queues: EvenQueue and OddQueue.

We then dequeue each element from the original queue:

1. If the element is even, we enqueue it to the EvenQueue.
2. If the element is odd, we enqueue it to the OddQueue.

By following these steps, we get the following queues:

Even Queue

2	4	6	8	10
---	---	---	---	----

Odd Queue

1	3	5	7	9
---	---	---	---	---

That's how de-merging of queues works. The original queue can be de-merged based on any condition, not just even or odd numbers. The condition depends on the particular problem we're trying to solve.

Priority De-Merging:

Priority de-merging is the process of splitting a priority queue into two or more priority queues based on certain criteria. The criteria can be any condition like splitting based on odd/even priorities, or separating elements into separate queues based on priority ranges, etc.

For example, suppose we have a priority queue:

PriorityQueue: [(A, 5), (B, 4), (C, 3), (D, 2), (E, 1)]

We could de-merge this queue into two queues: one for elements with a priority greater than or equal to 3, and one for elements with a priority less than 3:

PriorityQueue1: [(A, 5), (B, 4), (C, 3)]

PriorityQueue2: [(D, 2), (E, 1)]

Again, the specifics of how you merge or de-merge priority queues can depend on the specifics of the situation or the problem you're trying to solve.

Merging Queues

Merging queues is a process where two or more queues are combined into a single queue. The new queue might maintain the relative order of elements from the original queues, but the specifics can depend on the context or use case.

Let's take an example where we merge two queues into one.

Suppose we have the following queues:

Queue 1

1	3	5	7	9
---	---	---	---	---

Queue 2

2	4	6	8	10
---	---	---	---	----

We start by initializing an empty queue: **MergedQueue**.

We then dequeue each element from the original queues and enqueue it to the MergedQueue. We can do this in a variety of ways, depending on the specifics of the situation. For example, we could alternate between the queues, or we could add all elements from one queue before adding all elements from the other.

If we alternate between the queues, we get the following result:-

MergedQueue: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In this case, the MergedQueue maintains the relative order of elements from Queue1 and Queue2.

If we add all elements from Queue1 before adding all elements from Queue2, we get the following result:

MergedQueue: [1, 3, 5, 7, 9, 2, 4, 6, 8, 10]

In this case, the MergedQueue does not maintain the relative order of elements from Queue1 and Queue2.

These are simple examples, and the specifics of how you merge queues can depend on the specifics of the situation. For example, if you're merging priority queues, you might want to maintain the priority order in the merged queue.

Priority Merging :-

Priority merging is the process of combining two or more priority queues into a single priority queue. The elements in the new priority queue will be ordered based on their priorities.

For example, suppose we have two priority queues:

PriorityQueue1: [(A, 3), (B, 2), (C, 1)]

PriorityQueue2: [(D, 4), (E, 2), (F, 1)]

If we merge these priority queues, the resulting queue could be:

MergedPriorityQueue: [(D, 4), (A, 3), (B, 2), (E, 2), (C, 1), (F, 1)]

Note that the merged queue maintains the priority order of the elements.

CHAPTER – 6

TREE AND GRAPH

Tree: A tree is a non-linear data structure that consists of a set of interconnected nodes. It represents a hierarchical structure where each node has a specific relationship with other nodes.

Key Components of a Tree:

1. **Nodes:** Nodes are the fundamental building blocks of a tree. Each node

represents a unique element or entity in the tree. In a tree, nodes are connected to other nodes through edges.

2. **Edges:** Edges are the connections or links between nodes in a tree. They represent the relationships between the nodes. Each node (except the root) is connected to exactly one parent node and can have zero or more child nodes.

Rooted Tree: A rooted tree is a type of tree in which one node is designated as the root. The root is the topmost node in the tree and serves as the starting point for traversing the tree. Every other node in the tree has a parent-child relationship with the root or other nodes.

Key Properties of a Rooted Tree:

1. **Parent-Child Relationship:** In a rooted tree, each node (except the root) has exactly one parent node. The parent node is the node from which another node is directly connected. Conversely, the child nodes are the nodes connected to a specific parent node.
2. **Unique Path to Each Node:** In a rooted tree, there is a unique path from the root to every other node. This means that starting from the root, we can follow a sequence of edges to reach any node in the tree.
3. **Length of a Path:** The length of a path in a rooted tree refers to the number of edges along that path. It represents the distance or the number of steps required to traverse from one node to another.

Trees vs. Linear Data Structures: Trees are classified as non-linear data structures because the elements (nodes) do not form a sequential or linear order like arrays or linked lists. Instead, they exhibit a hierarchical structure, allowing for more complex relationships and organization.

Examples of Trees: One common example of a tree is a file system structure, where folders (nodes) are connected to subfolders or files (child nodes) through parent-child relationships. Another

example is a family tree, where individuals (nodes) are connected based on parent-child relationships to represent familial connections.

TREE TERMS

Root: A root is a node in a tree that does not have a parent. It serves as the starting point of the tree.

Internal Node: An internal node is a node in a tree that has at least one child. It is not a leaf node.

External (Leaf) Node: An external node, also known as a leaf node, is a node in a tree that does not have any children.

Ancestors of a Node: The ancestors of a node are the parent node, grandparent node, great-grandparent node, and so on, of a particular node. They are the nodes that come before the given node along the path to the root.

Descendants of a Node: The descendants of a node are the children, grandchildren, great-grandchildren, and so on, of a particular node. They are the nodes that come after the given node along the branches of the tree.

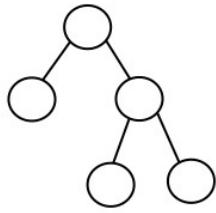
Depth of a Node: The depth of a node is the number of edges or levels between the node and the root. It represents the length of the path from the root to the node.

Height of a Tree: The height of a tree is the depth of the deepest node in the tree. It represents the maximum number of levels in the tree.

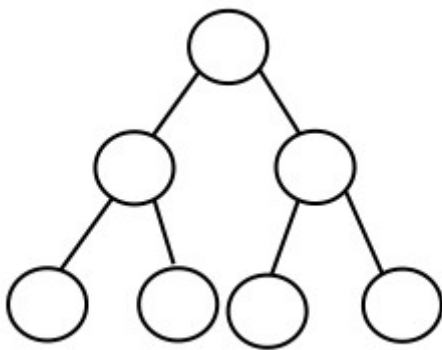
Subtree: A subtree is a smaller tree that is part of a larger tree. It consists of a node and all its descendants.

Binary Tree: A binary tree is a tree in which each node can have at most two children: a left child and a right child.

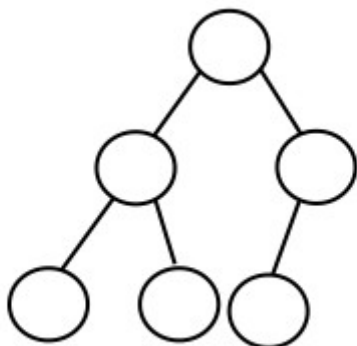
Full Binary Tree: A full binary tree is a binary tree where each node has either 0 or 2 children. In other words, every node in the tree is either a leaf node or has two children.



Balanced Binary Tree :- a binary tree where each node except the leaf nodes has left and right children and all the leaves are at the same level.



Complete Binary Tree: A complete binary tree is a binary tree in which all levels, except possibly the last one, are completely filled with nodes. In the last level, all nodes are filled in from left to right, meaning there are no "gaps" in the sequence of nodes from left to right on that level.



In simpler terms, imagine a binary tree where all levels are filled with nodes, except the last level, which may be partially filled from left to right. This means that as you move down the tree from

the root to the leaves, all levels are filled with nodes until the last level, which may have some gaps on the right side. The nodes are always placed as far left as possible, ensuring the tree is as balanced as possible given the number of nodes.

Binary Search Tree (BST)

A binary search tree, also known as an ordered binary tree, is a type of binary tree data structure that follows specific conditions:

1. Key Property:

- Every node in the BST has a key, which represents a unique element or value.
- No two nodes in the BST have the same key. This ensures that each key in the tree corresponds to a distinct element.

2. Ordering Property:

- For each node in the BST, all keys in its left subtree are smaller than the key in that node.
- Similarly, all keys in its right subtree are larger than the key in that node.
- This ordering property ensures that the elements in the BST are organized in a specific order.
- The left subtree contains elements that are smaller than the current node's key, while the right subtree contains elements that are larger.

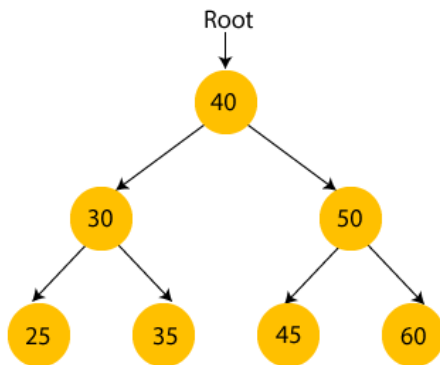
3. Recursive Structure:

- The left and right subtrees of any node in the BST are themselves binary search trees.
- This means that the ordering property holds for every node in the tree, not just at the root level.
- Each node in the BST acts as a root for its respective left and right subtrees, maintaining the

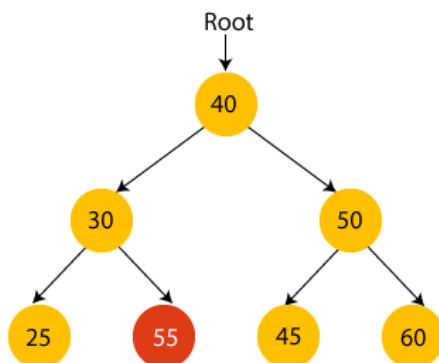
ordering property throughout the entire structure.

- This recursive structure allows for efficient search, insertion, and deletion operations within the BST.

Example :-



Below is not a Binary Search Tree



In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree.

The BST's structure allows for efficient searching, insertion, and deletion operations. By following the ordering property, it is possible to perform binary search operations to quickly locate elements within the tree.

Properties and Operations of a Binary Search Tree:

1. **Searching:** The BST's ordering property enables efficient searching. Starting from the root, comparisons are made between the target key and the keys in each node. Based on the comparison, the search is directed to the left or right subtree, eliminating the need to search the entire tree.
2. **Insertion:** When inserting a new node into a BST, it is placed based on a comparison with the keys of existing nodes. By following the ordering property, the new node is inserted in the appropriate location to maintain the tree's ordered structure.
3. **Deletion:** Removing a node from a BST involves finding the node to be deleted and then reorganizing the tree to maintain the ordering property. Several cases need to be considered, depending on whether the node has no children, one child, or two children.
4. **Traversal:** The BST can be traversed in various ways to visit and process all nodes. Common traversal methods include in-order, pre-order, and post-order traversal, each offering a different order of visiting the nodes.

The efficiency of operations in a BST depends on the height of the tree. A well-balanced BST, where the heights of the left and right subtrees are nearly equal, provides optimal performance. However, an unbalanced BST with a skewed structure can degrade the efficiency of operations.

Binary search trees are widely used in applications that require efficient searching and ordered data storage. They provide a balanced and structured way to organize data elements based on their key values.

Insertion of data In to BST

Insertion of data into a BST involves adding a new node with a key (representing a value) to

the tree while maintaining the ordering property of the BST.

1. Starting at the root node:

- If the tree is empty, the new node becomes the root.
- If the tree is not empty, compare the key of the new node with the key of the current node.
 - If the new node's key is less than the current node's key, move to the left sub tree.
 - If the new node's key is greater than the current node's key, move to the right sub tree.

2. Repeat Step 1 until reaching an appropriate position:

- If the current node has no left (or right) child, insert the new node as its left (or right) child, based on the comparison made in the previous step.
- If the current node has a left (or right) child, move to that child and repeat Step 1.

3. Once the new node is inserted, the ordering property of the BST is preserved:

- All keys in the left subtree of a node are smaller than the key in that node.
- All keys in the right subtree of a node are larger than the key in that node.

By following these steps, new nodes are inserted into the BST in a way that maintains the correct order of elements. This allows for efficient searching and retrieval of data based on their keys.

Traversing (visiting) in to BST

Traversing a binary search tree (BST) refers to the process of visiting each node in the tree in a specific order. It allows us to explore and access the elements of the BST.

There are three common methods of traversing a BST :-

1. Pre-order Traversal:

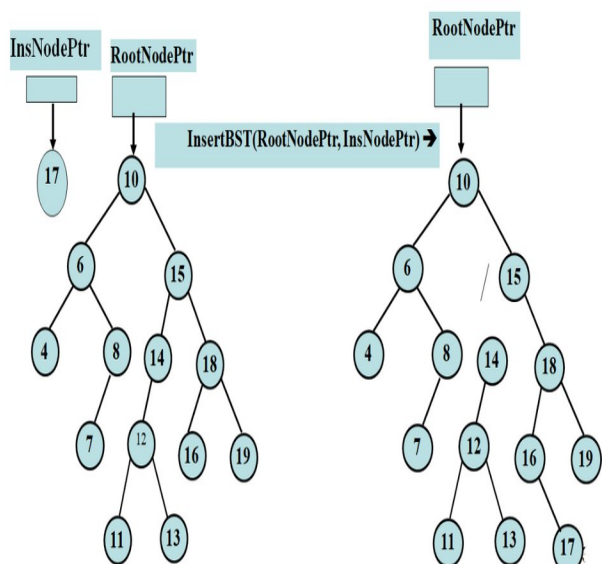
- In this traversal, we visit the nodes in a pre-defined order.
- The process involves visiting the parent node first, followed by recursively traversing the left subtree and then the right subtree.
- This traversal can be useful for creating a copy of the tree or for performing prefix expression evaluations.

2. Inorder Traversal:

- In this traversal, we visit the nodes in ascending order of their keys.
- The process involves recursively traversing the left subtree, visiting the parent node, and then recursively traversing the right sub tree.
- This traversal is often used to retrieve the elements in sorted order from the BST.

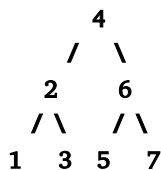
3. Postorder Traversal:

- In this traversal, we visit the nodes in a post-defined order.



- The process involves recursively traversing the left subtree, then the right subtree, and finally visiting the parent node.
- This traversal can be helpful for performing certain mathematical calculations or deleting nodes from the BST.

To understand these traversals, let's consider a simple BST:



In a preorder traversal, we would visit the nodes in the order: 4, 2, 1, 3, 6, 5, 7.

In an inorder traversal, we would visit the nodes in ascending order: 1, 2, 3, 4, 5, 6, 7.

In a postorder traversal, we would visit the nodes in the order: 1, 3, 2, 5, 7, 6, 4.

Traversing a BST allows us to process or display the elements stored in the tree in a particular order. The choice of traversal method depends on the requirements of our specific task.

Searching data in BST

Searching for data in a binary search tree (BST) involves finding a specific element within the tree based on its key value. The searching process utilizes the ordering property of the BST to efficiently locate the desired element.

Here's a brief description of the searching process in simpler terms:

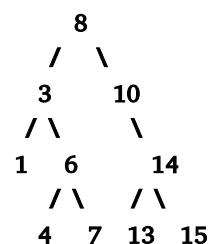
1. Start at the root node of the BST.
2. Compare the key value of the current node with the target key value that you want to search for.
 - If the current node's key matches the target key, you have found

the desired element and the search is successful.

- If the target key is smaller than the current node's key, move to the left subtree, as the desired element should be located in the left subtree if it exists.
 - If the target key is larger than the current node's key, move to the right subtree, as the desired element should be located in the right subtree if it exists.
3. Repeat step 2 until one of the following conditions is met:
 - The target key is found in a node, and the search is successful.
 - The search reaches a leaf node (a node with no children), indicating that the target key does not exist in the BST.

By following this process, you can efficiently search for data within a BST. The key advantage of a BST is that the search can be performed in logarithmic time complexity ($O(\log n)$), where n is the number of elements in the tree. This makes searching in a BST faster compared to searching in an unsorted data structure.

For example, let's consider the following BST:



If we want to search for the key value 6, we start at the root (8) and compare it with the target key (6). Since 6 is smaller, we move to the left subtree. Next, we compare the key value 3 with 6. Again, 6 is larger, so we move to the right subtree of node 3. Finally, we find the key value 6 in the left subtree of node 6, and the search is successful.

Deleting from a BST

Deleting a node from a binary search tree (BST) involves removing a specific element from the tree while maintaining the ordering property of the BST.

The process of deleting a node from a BST depends on the following scenarios:

1. Node has no children (leaf node):

- If the node to be deleted has no children, it can be simply removed from the tree without affecting the other nodes.

2. Node has one child:

- If the node to be deleted has only one child, we can bypass the node by connecting its child directly to its parent.

3. Node has two children:

- If the node to be deleted has two children, we need to find a suitable replacement node to preserve the BST properties. This replacement node is usually the minimum element in the right subtree or the maximum element in the left subtree.
- Once the replacement node is found, we replace the node to be deleted with the replacement node, and then delete the replacement node from its original position (using one of the above scenarios).

The steps for deleting a node from a BST in simpler terms:

1. Start at the root node and search for the node to be deleted by comparing its key with the keys of the nodes in the tree.
 - If the key is smaller, move to the left subtree.
 - If the key is larger, move to the right subtree.

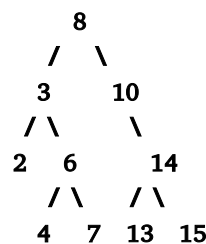
- Repeat this process until the node to be deleted is found or until reaching a leaf node (indicating the node does not exist in the tree).

2. Once the node to be deleted is found:

- If the node has no children, simply remove it from the tree.
- If the node has one child, bypass the node by connecting its child to its parent.
- If the node has two children, find the replacement node (either the minimum element in the right subtree or the maximum element in the left subtree).
- Replace the node to be deleted with the replacement node, and then delete the replacement node from its original position.

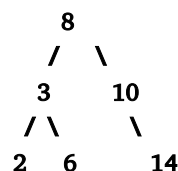
By following these steps, we can safely delete a node from the BST while preserving the ordering property of the tree.

Let's consider a simple binary search tree (BST) as an example :-



Scenario 1: Deleting a leaf node (node with no children)

- Let's delete the node with key 4.
- Since 4 is a leaf node, we can simply remove it from the tree.
- After deletion:



```

      \   /\
      7 13 15

```

Scenario 2: Deleting a node with one child

- Let's delete the node with key 10.
- Node 10 has only one child (right child 14).
- We bypass node 10 by connecting its child (14) directly to its parent (3).
- After deletion:

```

      8
     /\
    3  14
   /\  \
  2 6   15
   /\
  4 7

```

Scenario 3: Deleting a node with two children

- Let's delete the node with key 3.
- Node 3 has two children (2 and 6).
- We find the replacement node, which is the maximum element in its left subtree (2).
- We replace node 3 with the replacement node 2 and delete the original position of 2.
- After deletion:

```

      8
     /\
    2  10
   /\  \
  6 14
 /\  \
4 7 15

```

Scenario 3 another option : Deleting a node with two children

- Let's delete the node with key 3.
- Node 3 has two children (2 and 6).
- We find the replacement node, which is the minimum element in its right subtree (4).

- We replace node 3 with the replacement node 4 and delete the original position of 2.
- After deletion:

```

      8
     /\
    4  10
   /\  \
  2 6   14
   \  /\
   7 13 15

```

GRAPH – NON-LINEAR DATA STRUCTURE

A graph is a mathematical structure that consists of two main components: **vertices** (also called nodes) and **edges**. Vertices represent individual elements, and edges represent connections or relationships between those elements.

Graphs can be classified as **directed** or **undirected** based on whether the edges have a specific direction or not. In a directed graph, the edges have an associated direction, while in an undirected graph, the edges are bidirectional.

Graphs are widely used to model relationships between entities in various domains, such as social networks, transportation networks, and computer networks. They serve as the foundation for many graph algorithms that solve problems related to connectivity, shortest paths, network flows, and more.

Directed and Undirected Graph

A directed graph, also known as a digraph, is a type of graph where the edges have a specific direction associated with them. Each edge connects two vertices, with one vertex being the source or starting point, and the other being the destination or endpoint. The direction of the edges indicates a one-way relationship or flow between the vertices.

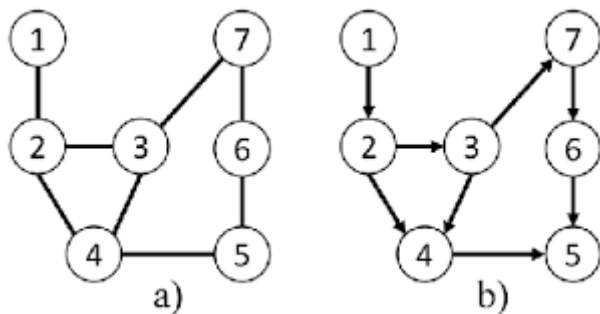
For example, in a directed graph representing a social network, the vertices can represent individuals, and the directed edges can represent

the connections or relationships between them. The direction of the edges would indicate the direction of influence, friendship, or communication between the individuals.

In contrast, an undirected graph is a type of graph where the edges do not have any specific direction. The edges in an undirected graph represent bidirectional relationships between vertices. This means that the relationship between two vertices is symmetric, and there is no distinction between a source and a destination.

For example, in an undirected graph representing a road network, the vertices can represent locations, and the undirected edges can represent the roads connecting the locations. The absence of direction in the edges signifies that the roads allow traffic in both directions.

Directed and undirected graphs have different properties and applications. Directed graphs are useful for modeling systems with one-way relationships, such as flows, dependencies, or directed communication. Undirected graphs are suitable for modeling symmetric relationships, such as connections, friendships, or undirected communication.



Both types of graphs are fundamental in graph theory and are used in various real-world applications, including social networks, transportation networks, computer networks, and many more.

Sparse and Dense Graphs

Sparse Graph: A sparse graph is a type of graph where the number of edges is much smaller compared to the number of vertices. In other

words, it has relatively few connections between its vertices. In a sparse graph, the density of edges is low. The opposite of a sparse graph is a dense graph.

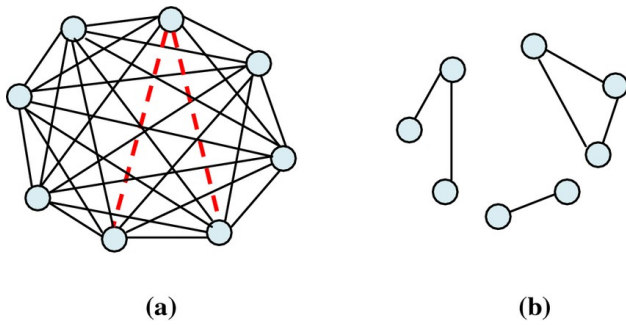
Sparse graphs are often encountered in real-world scenarios where connections between entities are not extensive. For example, in a social network, where each user represents a vertex, a sparse graph would indicate that users have fewer connections or interactions with each other. Similarly, in a transportation network, a sparse graph would imply that there are fewer direct routes between locations.

One of the key advantages of sparse graphs is that they require less memory to store and less computational effort to process, as there are fewer edges to consider. Many graph algorithms and data structures are designed with the assumption of a sparse graph.

Dense Graph: A dense graph is a type of graph where the number of edges is relatively larger compared to the number of vertices. In other words, it has a significant number of connections between its vertices. In a dense graph, the density of edges is high.

Dense graphs are often encountered in scenarios where entities have a high degree of connectivity or interactions. For example, in a complete social network where every user is connected to every other user, the graph would be dense. In a fully connected transportation network, where there are direct routes between every pair of locations, the graph would also be dense.

Dense graphs require more memory to store and more computational effort to process compared to sparse graphs, as there are more edges to consider. Some graph algorithms and data structures may have different performance characteristics when applied to dense graphs.



Determining whether a graph is sparse or dense depends on the ratio of the number of edges to the number of vertices. There is no strict threshold to define the boundary between sparse and dense graphs, as it depends on the specific context and problem being analyzed.

Understanding whether a graph is sparse or dense is important for selecting appropriate algorithms and data structures, as the choice may vary based on the characteristics of the graph.

Weighted and Unweighted Graph

In graph theory, a **weighted graph** is a type of graph where each edge is assigned a numerical value called a weight. The weight represents some kind of quantitative measure or cost associated with the edge. The weights can represent distances, costs, capacities, or any other relevant metric depending on the application.

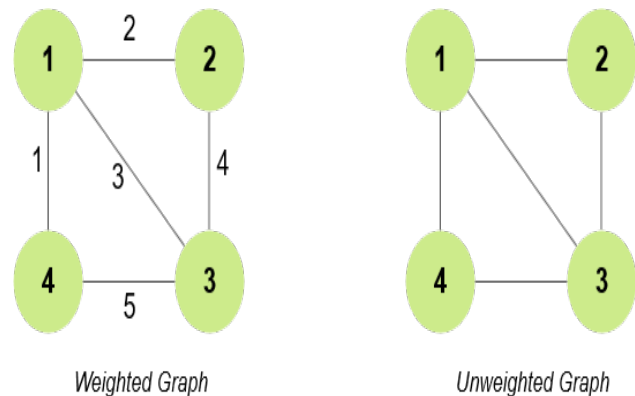
For example, in a weighted graph representing a transportation network, the weights of the edges could represent the distances between locations or the travel times required to move from one location to another. In a social network, the weights could represent the strength of relationships between individuals.

The presence of weights in a graph allows for more nuanced analysis and optimization. It enables the consideration of the costs or distances associated with traversing edges when solving problems such as finding the shortest path or determining the most efficient route.

On the other hand, an **unweighted graph** is a type of graph where all edges are considered to have the same uniform weight or no weight at all. In an unweighted graph, the focus is on the

presence or absence of connections between vertices rather than the quantitative values associated with the edges.

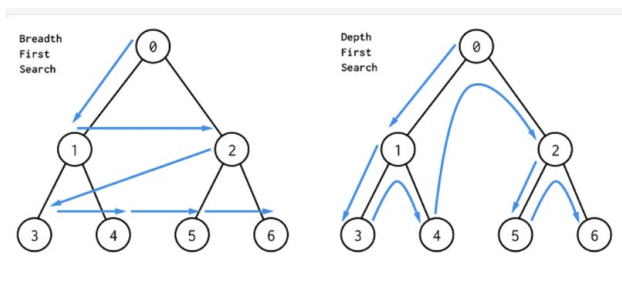
Unweighted graphs are often used in situations where the relationships between vertices are binary, indicating only the presence or absence of a connection. They are simpler to work with and require less computational effort compared to weighted graphs.



The choice between using a weighted or unweighted graph depends on the specific problem being addressed and the available information about the relationships between vertices. Weighted graphs provide more flexibility and precision in modeling real-world scenarios that involve varying costs or measures associated with the connections, while unweighted graphs offer simplicity and efficiency in certain applications where the quantitative values of the connections are not essential.

GRAPH TRAVERSALS

Breadth First Search (BFS) and Depth First Search (DFS) are two commonly used techniques for traversing or exploring a graph to visit all its vertices. They are used to search for specific nodes, find connected components, detect cycles, or find the shortest path between two vertices.



7.3.1 Breadth First Search (BFS):

In BFS, the traversal starts at a given vertex and explores all its neighbors before moving on to the next level of vertices. It follows the "breadth" of the graph, visiting all the vertices at a given level before moving to the next level.

7.3.2 Depth First Search (DFS):

In DFS, the traversal starts at a given vertex and explores as far as possible along each branch before backtracking. It follows the "depth" of the graph, going deeper into each branch before exploring other branches.

