UNIT-I:

Introduction and Overview: Definition, Classification and Operations of Data Structures. Algorithms: Complexity, Time-Space Tradeoff. Arrays: Definition and Classification of Arrays, Representation of Linear Arrays in Memory, Operations on Linear Arrays: Traversing, Inserting, Deleting, Searching, Sorting and erging. Searching: Linear Search and Binary Search, Comparison of Methods. Sorting: Bubble Sort, Selection Sort, and Insertion Sort. TwoDimensional Arrays, Representation of Two-Dimensional Arrays in Memory, Matrices and Sparse Matrices, Multi-Dimensional Arrays.

UNIT-II:

Linked Lists: Definition, Comparison with Arrays, Representation, Types of Linked lists, Traversing, Inserting, Deleting and Searching in Singly Linked List, Doubly Linked List and Circular Linked List. Applications of Linked Lists: Addition of Polynomials. Hashing and Collision: Hashing, Hash Tables, Types of Hash Functions, Collision, Collision Resolution with Open Addressing and Chaining.

UNIT-III:

Stacks: Definition, Representation of Stacks using Arrays and Linked List, Operations on Stacks using Arrays and Linked List, Application of Stacks: Arithmetic Expressions, Polish Notation, Conversion of Infix Expression to Postfix Expression, Evaluation of Postfix Expression. Recursion: Definition, Recursive Notation, Runtime Stack, Applications of Recursion: Factorial of Number, GCD, Fibonacci Series and Towers of Hanoi. Queues: Definition, Representation of Queues using Array and Linked List, Types of Queues: Simple Queue, Circular Queue, Double-Ended queue, Priority Queue, Operations on Simple Queues and Circular Queues using Array and Linked List, Applications of Queues.

UNIT-IV:

Graphs: Definition, Terminology, Representation, Traversal. Trees: Definition, Terminology, Binary Trees, Traversal of Binary Tree, Binary Search Tree, Inserting, Deleting and Searching in Binary Search Tree, Height Balanced Trees: AVL Trees, Insertion and Deletion in AVL Tree.

Data Structures Using C

MOST IMPORTANT

1. ***Explain Algorithm and its characteristics.***
2. ***Explain Performance analysis of an algorithm with time and space complexity.***
3. ***Explain Asymptotic notations (Big-O, Ω, Θ) with examples.***
4. ***Explain Stack ADT, operations and applications.***
5. ***Explain Infix to Postfix conversion using stack with example.***
6. ***Explain Evaluation of Postfix expression using stack.***
7. ***Explain Array ADT and its operations.***
8. ***Explain Sparse matrix representation (Triplet form) with advantages.***
9. ***Explain Polynomial representation using arrays.***
10. ***Explain Well-formed parenthesis checking using stack.***
11. ***Explain Reversing a string using stack.***

SHORT QUESTIONS

12. ***Characteristics of a good algorithm.***
13. ***Difference between time complexity and space complexity.***
14. ***Best case, average case and worst case complexities.***
15. ***Advantages and disadvantages of recursion.***
16. ***Operations on arrays.***
17. ***Stack operations – Push, Pop, Peek.***
18. ***Static stack vs Dynamic stack.***
19. ***Advantages of postfix expression.***
20. ***Applications of stack***

Meaning of Algorithm

*An algorithm is a **systematic and logical sequence of instructions** created to solve a problem in a definite manner.*
It begins with the given input data, processes the data through a series of clearly ordered steps, and finally produces the required output.
*An algorithm always completes its execution after a **limited number of steps**, ensuring that the solution process is controllable and predictable.*

Characteristics of an Algorithm

1. Finiteness

An algorithm must always terminate after executing a finite number of steps. It should never enter an infinite loop while solving a problem. This property ensures that the solution process is controllable, predictable, and practical. Without finiteness, it would be impossible to

obtain a final result. Therefore, every algorithm must have a clearly defined stopping condition.

2. Clarity (Unambiguity)

Each instruction in an algorithm must be precise, clear, and unambiguous. There should be no confusion or multiple interpretations of any step. Clear instructions help both humans and computers to follow the algorithm correctly. Ambiguous steps may lead to incorrect results or execution errors. Hence, clarity is essential for the correctness and reliability of an algorithm.

3. Input

An algorithm may accept one or more input values before execution. These inputs provide the necessary data required to solve the given problem. In some cases, an algorithm may also work without any input. The nature of the input determines how the algorithm processes information. Proper definition of input is necessary for accurate problem solving.

4. Output

An algorithm must produce at least one output after processing the input. The output represents the final solution or result of the problem. It is generated only after all processing steps are completed. An algorithm without output has no practical usefulness. Therefore, output is a mandatory characteristic of any algorithm.

5. Effectiveness

All steps of an algorithm must be simple, clear, and easy to execute. Each operation should be basic and completed in a finite amount of time. The algorithm should not include unrealistic or overly complex operations. Effectiveness ensures that the algorithm can be implemented as a computer program. This property makes the algorithm practical and usable.

6. Generality

A good algorithm should work for all valid input values, not just for a specific case. It should be capable of solving a class of similar problems. Generality allows the algorithm to be reused in different situations. This makes the algorithm flexible, efficient, and more useful in real-world applications. Hence, generality increases the overall usefulness of an algorithm.

Importance of Algorithms

1. Systematic Problem Solving

Algorithms provide a systematic and step-by-step approach to solving problems by breaking complex tasks into smaller, manageable steps. This reduces confusion during analysis and encourages logical thinking before coding. A clear algorithm helps in understanding the solution process and makes implementation easier and more accurate.

2. Improves Efficiency

Algorithms help in selecting the most efficient method to solve a problem. A well-designed algorithm minimizes execution time and reduces memory usage. Efficiency becomes particularly important when dealing with large volumes of data. By comparing different algorithms, the best and fastest solution can be chosen, thereby improving overall program performance.

3. Language Independence

An algorithm is independent of any specific programming language. The same algorithm can be implemented in different languages such as C, Java, or Python. This allows programmers to concentrate on problem-solving logic rather than syntax. As a result, algorithms are universal and transferable across platforms.

4. Easier Program Development

Algorithms act as a blueprint for program development. They guide programmers during coding and help in organizing the program structure. A well-written algorithm reduces logical errors and makes the program easier to understand, modify, and enhance. This leads to better quality and reliable software.

5. Simplifies Debugging and Maintenance

When errors occur, algorithms help in easily identifying the faulty step because the logic is clearly defined. Debugging becomes faster and more systematic. Well-structured algorithms also make program maintenance simpler. Future updates or changes can be made without disturbing the entire program, saving time and effort.

6. Foundation of Computer Science

Algorithms form the core concept of computer science and are used in data structures, operating systems, and databases to improve logical and analytical thinking and to design efficient software systems, making them fundamental to all computing applications.

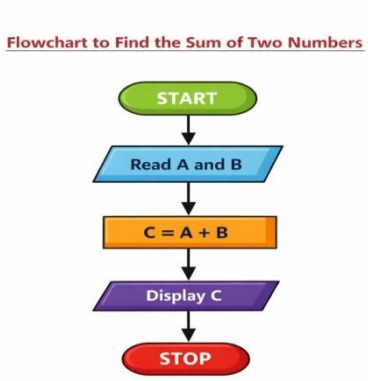
Algorithm – Explanation with Example

An algorithm is a logical and systematic sequence of steps used to solve a problem, which clearly defines the order of operations, is written before coding to understand the solution, is

independent of programming languages, and consists of a start, input, processing, output, and stop.

Example: Algorithm to Find the Sum of Two Numbers

Algorithm Steps



1. *Start*
2. *Read two numbers A and B*
3. *Compute C = A + B*
4. *Display the value of C*
5. *Stop*

Explanation of the Diagram

*The flowchart begins with the **Start** symbol.*
It then reads two input values A and B.
The processing step adds the two values and stores the result in C.
The output step displays the result.
*Finally, the flowchart ends with the **Stop** symbol.*

Program – Explanation with Example

A program is a set of instructions written in a specific programming language to perform a particular task on a computer.It is the practical implementation of an algorithm. A program follows the syntax and rules of the chosen programming language. When executed, it interacts with the computer system to produce output. Programs are used to solve real-world problems by converting logic into executable form.

```
#include <stdio.h>
int main() {
    int A, B, C;
    printf("Enter two numbers: ");
    scanf("%d %d", &A, &B);
    C = A + B;
    printf("Sum = %d", C);
    return 0;
}
```

Explanation of the Program

*In this program, the variables **A** and **B** are used to store the input values entered by the user. The `scanf` statement is used to read two numbers from the keyboard and store them in these variables. The statement **C = A + B** performs the addition of the two input values. The result of the addition is stored in the variable **C**. Finally, the `printf` statement displays the calculated sum on the screen..*

Aspect	Algorithm	Program
Definition	*An algorithm is a step-by-step logical procedure used to solve a problem.*	*A program is a set of instructions written in a programming language to execute a solution.*
Nature	*Conceptual and logical in nature.*	*Practical and executable in nature.*
Language Dependency	*Independent of any programming language.*	*Dependent on a specific programming language.*
Representation	*Written using plain English, pseudocode, or flowcharts.*	*Written using programming languages such as C, Java, or Python.*
Execution	*Cannot be executed directly by a computer.*	*Can be executed by a computer after compilation or interpretation.*
Purpose	*Focuses on problem analysis and solution design.*	*Focuses on implementing and running the solution.*
Flexibility	*Same algorithm can be used across different platforms.*	*Program must be rewritten for different languages or platforms.*
Stage	*Exists at the design and planning stage.*	*Exists at the implementation stage.*

Algorithm Representation Methods

An algorithm can be represented in different forms to make problem solving simple and understandable.
*The commonly used algorithm representation methods are **Natural Language**, **Pseudocode**, and **Flowchart**.*

1. Natural Language Representation

Natural language representation describes an algorithm using simple English sentences.
Each step is written in a sequential and readable form.
This method is easy to understand, especially for beginners.
However, it may sometimes lead to ambiguity if steps are not clearly written.
It is mainly used for explaining the logic of a problem.

Example (Finding the sum of two numbers)

1. *Start the process*
2. *Read two numbers*

3. *Add the two numbers*
4. *Display the result*
5. *Stop*

2. Pseudocode

Pseudocode *is a structured way of writing an algorithm using programming-like statements. It does not follow the syntax of any specific programming language. Pseudocode clearly shows the logical steps involved in solving a problem. It is easy to understand and simple to convert into an actual program. This method is widely used in algorithm design and examinations.*

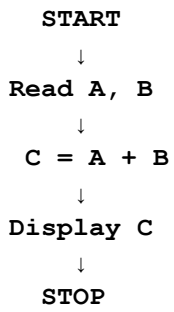
Example (Finding the sum of two numbers)

```
BEGIN
  READ A, B
  C ← A + B
  PRINT C
END
```

3. Flowchart

A flowchart is a graphical representation of an algorithm. It uses standard symbols such as ovals for start and stop, rectangles for processing, and diamonds for decision making. Flowcharts show the flow of control from one step to another in a clear and systematic manner. They make algorithms easy to visualize, understand, and communicate. Flowcharts are very useful for explanation, analysis, and debugging of programs.

Example (Finding the sum of two numbers)



Advantages of Pseudocode

1. Easy to Understand

Pseudo code uses simple English statements to describe program logic. It is easy to read and understand even for beginners. No prior knowledge of any programming language is

required to follow pseudo code. Because of this simplicity, it is ideal for learning, teaching and explaining algorithms.

2. Language Independent

Pseudo code is not tied to any specific programming language. The same pseudo code can be easily converted into languages such as C, Java, or Python. This allows programmers to focus on the logic of the problem rather than worrying about syntax rules. As a result, pseudo code makes problem solving flexible and universal.

3. Simplifies Program Development

Pseudocode acts as a blueprint before writing the actual code. It helps in organizing the program logic in a clear, step-by-step manner. Once the pseudocode is prepared, coding becomes easier and faster. This also reduces logical mistakes during programming.

4. Easy to Convert into Program

Each step in pseudocode closely resembles programming statements. Therefore, it can be easily and directly translated into a program with minimal effort. This saves time during coding. Hence, pseudocode bridges the gap between program logic and actual code.

5. Helps in Error Detection

Logical errors can be identified easily at the pseudocode stage itself. Mistakes can be corrected before writing the actual program. This reduces debugging time later. As a result, the quality of the final program is improved.

6. Useful for Documentation and Communication

Pseudocode clearly documents the logic of a program. It helps programmers explain solutions to others in a simple manner. Team members can understand the approach without seeing the actual code. Thus, it improves communication and collaboration.

Recursive Algorithms

Recursive Algorithm

A recursive algorithm is an algorithm that solves a problem by calling itself repeatedly with a smaller input. Instead of solving the entire problem at once, recursion breaks the problem into simpler sub-problems. Each recursive call works on a reduced version of the original problem, and the process continues until a stopping condition is reached. Recursion is widely used in problems that exhibit repetitive or self-similar structure.

1. Concept of Recursion

Recursion is a technique in which a function or algorithm calls itself to solve a problem. With each call, the size of the problem is reduced. The final solution is obtained by combining the results of smaller sub-problems. Recursion provides a natural and simple way to express complex problems. Many mathematical computations and data structure operations effectively use recursion.

2. Base Case and Recursive Case

Base Case

The base case is the condition at which the recursion stops. It provides a direct solution without making further recursive calls. Without a base case, the recursion would continue indefinitely, leading to infinite execution. Therefore, the base case is the most important part of a recursive algorithm. Every recursive algorithm must have at least one base case.

Recursive Case

The recursive case defines how the problem is reduced into smaller sub-problems. It involves calling the same function with a smaller or simpler input. Each recursive call moves the solution closer to the base case. This part performs repeated computation until the base case is reached. Together, the base case and recursive case ensure the correctness of the algorithm.

3. Direct Recursion

Direct recursion is a type of recursion in which a function calls itself directly within its own definition. The recursive call is made using the same function name. Each call reduces the problem size and progresses toward a stopping condition. Direct recursion is simple and easy to understand. It is commonly used in basic problems such as factorial calculation, Fibonacci series, and sum of numbers.

Example: Factorial of a Number

Algorithm / Logic:

- ***If n = 0, return 1 (base case)***
- ***Otherwise, return n × factorial (n−1)***

C Program Example:

```
int factorial(int n)
{
```

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

In this example, the function `factorial()` calls itself directly until the base case is reached.

Indirect Recursion (with Example)

Indirect recursion is a type of recursion in which a function does not call itself directly. Instead, it calls another function, and that function in turn calls the original function, forming a cycle of function calls between two or more functions. Each function call works on a reduced or modified input so that the recursion eventually reaches a stopping condition. Indirect recursion is more complex than direct recursion and is generally used in advanced problem-solving situations.

Example of Indirect Recursion (C Program)

```
#include <stdio.h>

void functionA(int n);
void functionB(int n);

void functionA(int n) {
    if (n > 0) {
        printf("%d ", n);
        functionB(n - 1);
    }
}

void functionB(int n) {
    if (n > 0) {
        printf("%d ", n);
        functionA(n - 1);
    }
}
```

In this example, `functionA()` calls `functionB()`, and `functionB()` calls `functionA()`, forming indirect recursion.

Advantages of Recursion

1. *Simplifies the solution of complex problems.*
2. *Makes the program shorter and more readable.*

3. *Closely matches mathematical definitions.*
4. *Very useful for solving tree and graph problems.*
5. *Improves clarity in divide-and-conquer algorithms.*

Disadvantages of Recursion

1. *Consumes more memory due to multiple function calls.*
2. *Each recursive call uses additional stack space.*
3. *May be slower than iterative solutions.*
4. *Incorrect base cases can cause infinite recursion.*
5. *Debugging recursive programs is difficult.*

Comparison: Direct Recursion vs Indirect Recursion

Aspect	*Direct Recursion*	*Indirect Recursion*
Definition	*A function calls itself directly within its own definition.*	*A function calls another function, which in turn calls the original function.*
Function Calls	*Only one function is involved.*	*Two or more functions are involved.*
Calling Method	*The recursive call is made using the same function name.*	*The recursive calls occur through a chain of functions.*
Complexity	*Simple and easy to understand.*	*More complex and harder to trace.*
Control Flow	*Flow of execution is straightforward.*	*Flow of execution forms a cycle between functions.*
Examples	*Factorial, Fibonacci series.*	*Two functions calling each other alternately.*
Usage	*Used in basic and common recursive problems.*	*Used in advanced or specialized problem solving.*

Explain Performance Analysis of an Algorithm with Time and Space Complexity

Introduction

*Performance analysis of an algorithm is the process of evaluating how efficiently an algorithm solves a given problem. It mainly focuses on measuring the **time taken** and **memory used** by an algorithm. This analysis is carried out theoretically and does not depend on a particular computer system or programming language. It helps in comparing different algorithms that solve the same problem. Thus, performance analysis plays a crucial role in designing efficient and effective programs.*

Need for Performance Analysis

Performance analysis is necessary to determine the efficiency of an algorithm. It helps in predicting how an algorithm behaves when the input size becomes large. By analyzing

performance, we can choose the most suitable algorithm among many alternatives. It ensures optimal utilization of system resources such as time and memory. Therefore, performance analysis helps in developing fast, reliable, and scalable programs.

Performance Analysis of an Algorithm

Performance analysis studies the behavior of an algorithm as the size of the input increases. It enables programmers to estimate the efficiency of an algorithm even before it is implemented. The performance of an algorithm is evaluated using two main parameters:

1. ***Time Complexity***
2. ***Space Complexity***

Time Complexity

Definition

Time complexity is a measure of the amount of time required by an algorithm to execute as a function of the input size. It is determined by counting the number of basic operations performed by the algorithm. Time complexity helps estimate execution time without actually running the program. It is usually expressed using asymptotic notations. Time complexity focuses on the efficiency of the algorithm rather than the actual execution time on a machine.

Cases of Time Complexity

1. Best Case

*The best case represents the **minimum time** taken by an algorithm to execute. It occurs when the input is in the most favorable condition. This case gives an optimistic measure of algorithm performance. However, it rarely occurs in real-life situations. Best case analysis is mainly useful for theoretical understanding.*

2. Average Case

*The average case represents the **expected time** taken by an algorithm. It considers all possible inputs and their probabilities of occurrence. This case provides a more realistic measure of performance compared to the best case. However, average case analysis is often difficult to compute. Its value lies between best case and worst case performance.*

3. Worst Case

*The worst case represents the **maximum time** taken by an algorithm to execute. It occurs when the input is in the least favorable condition. Worst case analysis provides an upper bound on the execution time of an algorithm. It is very important for ensuring reliability,*

especially in critical applications. Therefore, worst case analysis is the most commonly used method in algorithm performance analysis.

Space Complexity

Definition

Space complexity is the measure of the total memory required by an algorithm during its execution. It includes memory used for variables, constants, arrays, data structures, and recursion calls. Space complexity helps in determining the memory efficiency of an algorithm. It is especially important in systems where memory resources are limited. Like time complexity, space complexity depends on the size of the input.

Components of Space Complexity

1. Fixed Space

Fixed space refers to the memory required for constants, program instructions, and simple variables. This memory requirement does not depend on the input size and remains constant throughout the execution of the algorithm. Examples include loop counters, fixed variables, and constants. Fixed space is usually small and does not significantly affect overall space complexity.

2. Variable Space

Variable space depends on the size of the input. It includes memory used for arrays, dynamic data structures, and recursion stacks. As the input size increases, the variable space requirement also increases. Recursive algorithms generally consume more variable space due to multiple function calls. This component has a major impact on the total space complexity of an algorithm.

Explain Asymptotic Notations (Big-O, Ω, Θ) with Examples

Introduction

*Asymptotic notations are mathematical tools used to analyze the performance of algorithms. They describe how the **time complexity** or **space complexity** of an algorithm grows as the input size increases. These notations help compare algorithms independently of hardware, programming language, or implementation details. Asymptotic analysis focuses on the **growth rate** of complexity rather than the exact execution time. The commonly used asymptotic notations are **Big-O**, **Omega (Ω)**, and **Theta (Θ)**.*

1. Big-O Notation (O)

Definition

*Big-O notation represents the **upper bound** on the time or space complexity of an algorithm. It describes the **worst-case performance** of an algorithm. Big-O tells us the maximum time an algorithm may take for a given input size. It is the most widely used notation in algorithm analysis. By providing a guaranteed performance limit, Big-O ensures reliability.*

Example 1: Linear Search

- *In the worst case, the element is found at the last position.*
- *The algorithm checks all **n** elements.*

T(n)=n⇒O(n)T(n) = n \Rightarrow O(n)T(n)=n⇒O(n)

Example 2:

T(n)=3n2+5n+2⇒O(n2)T(n) = 3n^2 + 5n + 2 \Rightarrow O(n^2)T(n)=3n2+5n+2⇒O(n2)

(Only the highest-order term is considered; constants and lower-order terms are ignored.)

2. Omega Notation (Ω)

Definition

*Omega (Ω) notation represents the **lower bound** on the performance of an algorithm. It describes the **best-case scenario**. Omega gives the **minimum time** required by an algorithm to solve a problem. It indicates how fast an algorithm can possibly run. This notation is mainly used for theoretical analysis.*

Example: Linear Search

- *If the element is found at the first position.*
- *Only one comparison is required.*

T(n)=1⇒Ω(1)T(n) = 1 \Rightarrow \Omega(1)T(n)=1⇒Ω(1)

This shows that the best-case time complexity is constant.

3. Theta Notation (Θ)

Definition

*Theta (Θ) notation represents the **tight bound** of an algorithm. It describes both the **upper and lower bounds** simultaneously. Theta gives the **exact order of growth** of an algorithm. It*

provides a more precise performance measure. Theta is used when the best-case and worst-case complexities grow at the same rate.

Example 1:

*A loop that runs exactly **n** times:*

T(n)=2n+3⇒Θ(n)T(n) = 2n + 3 \Rightarrow \Theta(n)T(n)=2n+3⇒Θ(n)

Example 2:

T(n)=n2+n⇒Θ(n2)T(n) = n^2 + n \Rightarrow \Theta(n^2)T(n)=n2+n⇒Θ(n2)

In these cases, the algorithm always grows at the same rate for all inputs.

Comparison Table: Asymptotic Notations

Aspect	Big-O Notation (O)	Omega Notation (Ω)	Theta Notation (Θ)
Meaning	Represents the upper bound of an algorithm's performance.	Represents the lower bound of an algorithm's performance.	Represents the exact (tight) bound of an algorithm.
Type of Bound	Upper bound	Lower bound	Both upper and lower bound
Case Represented	Worst case	Best case	Average / exact case
Growth Rate	Shows maximum growth rate.	Shows minimum growth rate.	Shows precise growth rate.
Usage	Used to guarantee performance limits.	Used for theoretical minimum analysis.	Used when growth rate is fixed.
Accuracy	Less precise than Θ.	Less precise than Θ.	Most precise among all.
Example	T(n)=3n2+2n+1→O(n2)T(n)=3n^2+2n+1 → O(n^2)T(n)=3n2+2n+1→O(n2)	T(n)=n→Ω(n)T(n)=n → Ω(n)T(n)=n→Ω(n)	T(n)=2n+4→Θ(n)T(n)=2n+4 → Θ(n)T(n)=2n+4→Θ(n)
Common Use	Most widely used in practice.	Used mainly in theory.	Used for exact analysis.

Arrays – Abstract Data Type (ADT)

Introduction

An array is one of the most basic and important data structures in computer science.
When defined as an Abstract Data Type (ADT), an array focuses on **what operations can be performed**, rather than how it is implemented.
Array ADT provides a logical view of array operations independent of programming language.
It helps in organizing and storing similar types of data efficiently.
Array ADT is widely used in algorithm design and problem solving.

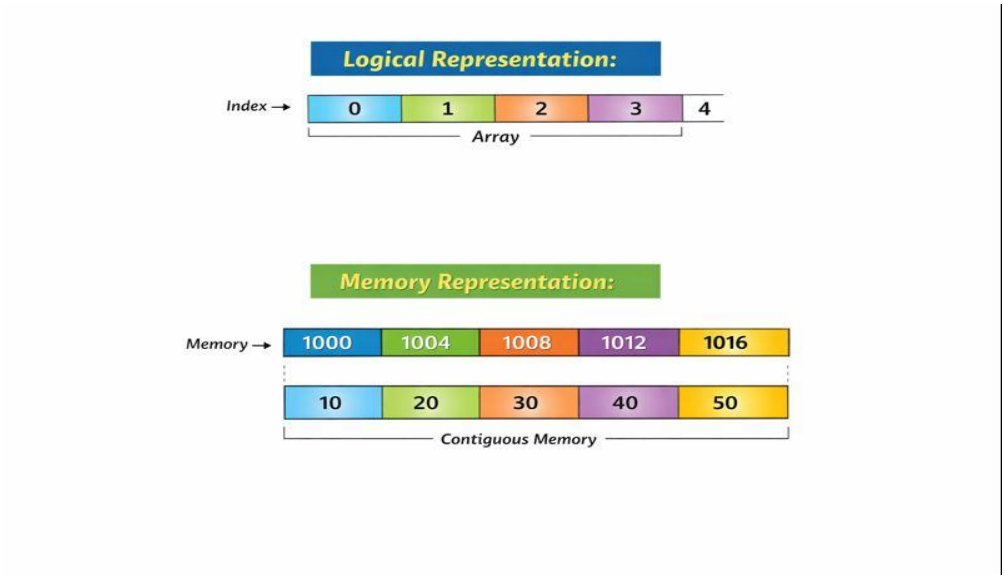
Definition of Array ADT

An **Array Abstract Data Type (ADT)** is a collection of elements of the **same data type**, stored in **contiguous memory locations**, and accessed using a common name with an index value.
It defines a set of operations that can be performed on the array.
The internal representation is hidden from the user.
Only the behavior of operations is specified.
This makes Array ADT independent of implementation details.

Characteristics of Array ADT

- All elements are of the **same data type**
- Elements are stored in **contiguous memory locations**
- Each element can be accessed using an **index**
- The size of the array is usually **fixed**
- Fast access is possible using index values

Neat Diagram of Array ADT



Operations on Array ADT

1. Traversal

Traversal is the process of accessing each element of the array one by one.
It is used to display or process array elements.
Traversal takes linear time.
All elements are visited sequentially.
It is the most basic array operation.

---

2. Insertion

Insertion is the process of adding a new element at a specific position.
Existing elements are shifted to make space.
Insertion at the beginning or middle is costly.
Insertion at the end is faster if space is available.
It increases the size of the array logically.

---

3. Deletion

Deletion is the process of removing an element from a given position.
Remaining elements are shifted to fill the gap.
Deletion reduces the logical size of the array.
It may take more time if elements need to be shifted.
Care must be taken to avoid index errors.

4. Searching

Searching is used to find the position of a given element.
Common methods include linear search and binary search.
Linear search checks elements one by one.
Binary search works only on sorted arrays.
Searching helps in data retrieval.

---

5. Updating

Updating is the process of modifying an existing element.
The value at a given index is replaced with a new value.
This operation is very fast.
It requires only direct access using index.
No shifting of elements is required.

Advantages of Array ADT

- Simple and easy to use
- Fast access using index
- Efficient memory usage for fixed-size data
- Suitable for storing homogeneous data
- Useful in implementing other data structures

Limitations of Array ADT

- Fixed size leads to memory wastage or overflow
- Insertion and deletion are costly
- Cannot store different data types
- Not suitable for dynamic data
- Resizing is difficult

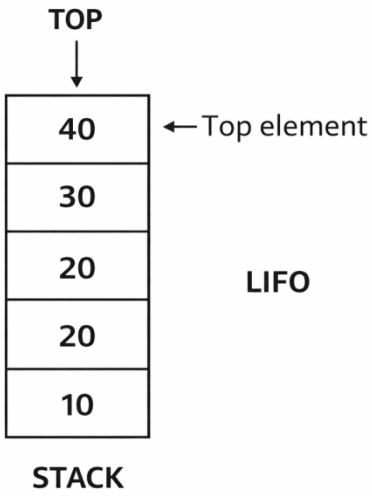
Explain Stack ADT, Operations and Applications

Introduction

A stack is one of the fundamental linear data structures used in computer science.
It follows a specific order of data access which makes it suitable for many practical problems.
When defined as an Abstract Data Type (ADT), a stack specifies **what operations are allowed**, not how they are implemented.
Stack ADT plays an important role in algorithm design and program execution.
It is widely used in expression evaluation, recursion, and memory management.

A **Stack Abstract Data Type (ADT)** is a linear data structure in which insertion and deletion of elements are performed **only at one end**, called the **TOP**.
It follows the **LIFO (Last In, First Out)** principle.
The last element inserted is the first element removed.
Stack ADT specifies the allowed operations, not the implementation details.
It is widely used in program execution and expression processing.

Neat Diagram of Stack ADT



Basic Operations of Stack ADT

- **Push** – Insert an element at the top
- **Pop** – Remove the top element
- **Peek** – View the top element
- **IsEmpty** – Check whether stack is empty
- **IsFull** – Check whether stack is full

Operations on Stack ADT

1. Push Operation

Push is the operation of **inserting an element** onto the top of the stack.
Before insertion, the stack is checked for overflow.
If the stack is full, insertion is not possible.
Push increases the stack size by one.
It always occurs at the top position.

---

2. Pop Operation

Pop is the operation of **removing the top element** from the stack.
Before deletion, the stack is checked for underflow.
If the stack is empty, deletion is not possible.
Pop decreases the stack size by one.
Only the top element can be removed.

---

3. Peek (Top) Operation

Peek returns the value of the **top element** without removing it.
It helps in viewing the current top of the stack.
This operation does not modify the stack.
It is useful in expression evaluation.
Peek improves program efficiency.

---

4. IsEmpty Operation

This operation checks whether the stack is empty.
It returns true if there are no elements in the stack.
It helps prevent underflow errors.
This check is performed before pop operation.
It ensures safe stack usage.

---

5. IsFull Operation

This operation checks whether the stack is full.
It is mainly used in array-based stack implementation.
It helps prevent overflow errors.
This check is done before push operation.
It ensures memory safety.

Applications of Stack ADT

1. Expression Evaluation

Stacks are used to evaluate postfix and prefix expressions.
Operators and operands are processed using stack operations.
Stacks eliminate the need for parentheses.
This application is widely used in compilers.
It simplifies expression computation.

---

2. Infix to Postfix Conversion

Stacks are used to convert infix expressions into postfix form.
Operator precedence is managed using stack operations.
This conversion makes expression evaluation easier.
It is an important application in syntax analysis.
Stacks help handle operators efficiently.

---

3. Function Calls and Recursion

Stacks store function call information such as local variables and return addresses.
Each recursive call is pushed onto the stack.

After completion, it is popped from the stack.
This mechanism is known as the call stack.
It supports program execution control.

---

4. Checking Well-Formed Parentheses

Stacks are used to check whether parentheses are balanced.
Opening brackets are pushed onto the stack.
Closing brackets pop matching elements.
An empty stack indicates correct expression.
This is useful in syntax checking.

---

5. Reversing a String

Characters of a string are pushed onto the stack.
They are popped one by one to reverse the string.
This method is simple and efficient.
It demonstrates LIFO principle clearly.
Stacks make string reversal easy.

Advantages of Stack ADT

- Simple and easy to implement
- Efficient access to data
- Useful in many algorithms
- Supports recursion naturally

Limitations of Stack ADT

- Limited access to elements
- Only top element can be accessed
- Fixed size in static stacks
- Overflow and underflow problems

Explain Infix to Postfix Conversion Using Stack with Example

Introduction

In computer science, expressions can be written in infix, postfix, or prefix forms.
In **infix notation**, operators are written between operands (for example, A + B).
Computers find infix expressions difficult to evaluate because of operator precedence and parentheses.
To simplify evaluation, infix expressions are converted into **postfix expressions**.
A **stack** is used to perform this conversion efficiently.

Infix and Postfix Expressions

Infix Expression

Example 3: Infix to Postfix Conversion

Infix Expression

```
A * ( B + C ) / D
```

Step-by-Step Conversion

Symbol	Stack	Postfix
A	—	A
*	*	A
(	* (	A
B	* (	AB
+	* (+	AB
C	* (+	ABC
)	*	ABC+
/	/	ABC+*
D	/	ABC+*D
End	—	ABC+*D/

Postfix Expression

```
ABC+*D/
```

Explanation

- Operand **A** is added directly to postfix.
- `*` is pushed onto the stack.
- Parentheses control evaluation of `B + C`.
- Inside parentheses, operands are added and `+` is popped after `)`.
- `*` has equal precedence with `/`, so it is popped before pushing `/`.
- Operand **D** is added.
- Remaining operator `/` is popped at the end.

Thus, the final postfix expression is **ABC+*D/**.

Explain Evaluation of Postfix Expression Using Stack

Introduction

Postfix expression evaluation is one of the most important applications of stack.
In postfix notation, the operator appears **after the operands**.
Unlike infix expressions, postfix expressions **do not require parentheses** or operator precedence rules.
Because of this, postfix expressions are easier for computers to evaluate.
A **stack** is used to perform postfix expression evaluation efficiently.

Postfix Expression

A postfix expression is an expression in which the operator comes **after** the operands.

Example:
Infix: `A + B`
Postfix: `AB+`

Need for Stack in Postfix Evaluation

A stack is used to store operands temporarily.
When an operator is encountered, operands are popped from the stack.
The operation is performed and the result is pushed back onto the stack.
This process continues until the entire expression is evaluated.
Thus, stack plays a key role in postfix expression evaluation.

Algorithm: Evaluation of Postfix Expression

1. Initialize an empty stack.
2. Scan the postfix expression from left to right.
3. If the symbol is an **operand**, push it onto the stack.
4. If the symbol is an **operator**:
    - Pop the top two operands from the stack.
    - Perform the operation.
    - Push the result back onto the stack.
5. After scanning the entire expression, the top of the stack contains the final result.

Example: Evaluation of Postfix Expression

Postfix Expression

```
23*54*+9-
```

Step-by-Step Evaluation

Symbol	Stack	Action
2	2	Push operand
3	2, 3	Push operand
*	6	2 × 3 = 6
5	6, 5	Push operand
4	6, 5, 4	Push operand
*	6, 20	5 × 4 = 20
+	26	6 + 20 = 26
9	26, 9	Push operand

Symbol	Stack	Action
-	17	26 − 9 = 17

Final Result

```
17
```

Explanation of Example

Operands are pushed onto the stack as they appear.
When an operator is encountered, the top two operands are popped.
The operation is performed in correct order.
The result is pushed back onto the stack.
After processing all symbols, the final value remains on the stack.

Advantages of Postfix Expression Evaluation

- No need for parentheses
- No operator precedence rules required
- Simple and fast evaluation
- Easy implementation using stack
- Widely used in compilers and calculators

Explain Sparse Matrix Representation (Triplet Form) with Advantages

Introduction

A matrix in which most of the elements are zero is called a **sparse matrix**.
Storing all elements of such a matrix leads to wastage of memory.
To overcome this problem, special storage techniques are used.
One such efficient method is the **triplet representation**.
This method stores only the non-zero elements of the matrix.

Sparse Matrix

Definition

A **sparse matrix** is a matrix in which the number of zero elements is much greater than the number of non-zero elements.
Storing zero values is unnecessary and inefficient.
Sparse matrices are common in scientific and engineering applications.
Hence, compact storage methods are required.

Need for Sparse Matrix Representation

- Reduces memory wastage caused by storing zeros
- Improves storage efficiency
- Makes matrix operations faster
- Useful when matrix size is large
- Optimizes resource utilization

Triplet Representation of Sparse Matrix

Concept

In triplet representation, only **non-zero elements** of the matrix are stored.
Each non-zero element is represented using **three values**:

1. **Row number**
2. **Column number**
3. **Value**

The first row of the triplet matrix stores:

- Total number of rows
- Total number of columns
- Total number of non-zero elements

Structure of Triplet Form

Row Column Value

r c v

- `r` → Row index of non-zero element
- `c` → Column index of non-zero element
- `v` → Actual value

Example of Sparse Matrix

Original Matrix

```
0 0 3
0 4 0
0 0 5
```

This matrix has:

- 3 rows
- 3 columns
- 3 non-zero elements

Triplet Representation

Row Column Value

3 3 3
0 2 3
1 1 4
2 2 5

- First row indicates matrix size and number of non-zero elements
- Remaining rows store only non-zero values

Advantages of Triplet Representation

1. Memory Efficient

Only non-zero elements are stored.
Zero elements are ignored.
This significantly reduces memory usage.
It is ideal for large sparse matrices.
Thus, storage becomes efficient.

---

2. Saves Processing Time

Operations are performed only on non-zero elements.
Unnecessary computations on zero values are avoided.
This improves performance.
Algorithms run faster.
Hence, processing time is reduced.

---

3. Easy to Implement

Triplet form is simple to understand and implement.
It uses basic row-column-value structure.
No complex data structures are required.
It is easy to store and retrieve values.
Thus, implementation is straightforward.

---

4. Efficient for Matrix Operations

Matrix addition and transpose can be performed efficiently.
Operations work only on stored elements.
This improves computational efficiency.
It is suitable for real-time applications.
Thus, performance is enhanced.

---

5. Reduces Storage Cost

Less memory requirement reduces overall storage cost.
Useful in memory-constrained systems.
Ideal for scientific computing.
Widely used in databases and graphics.
Hence, cost-effective.

Limitations of Triplet Representation

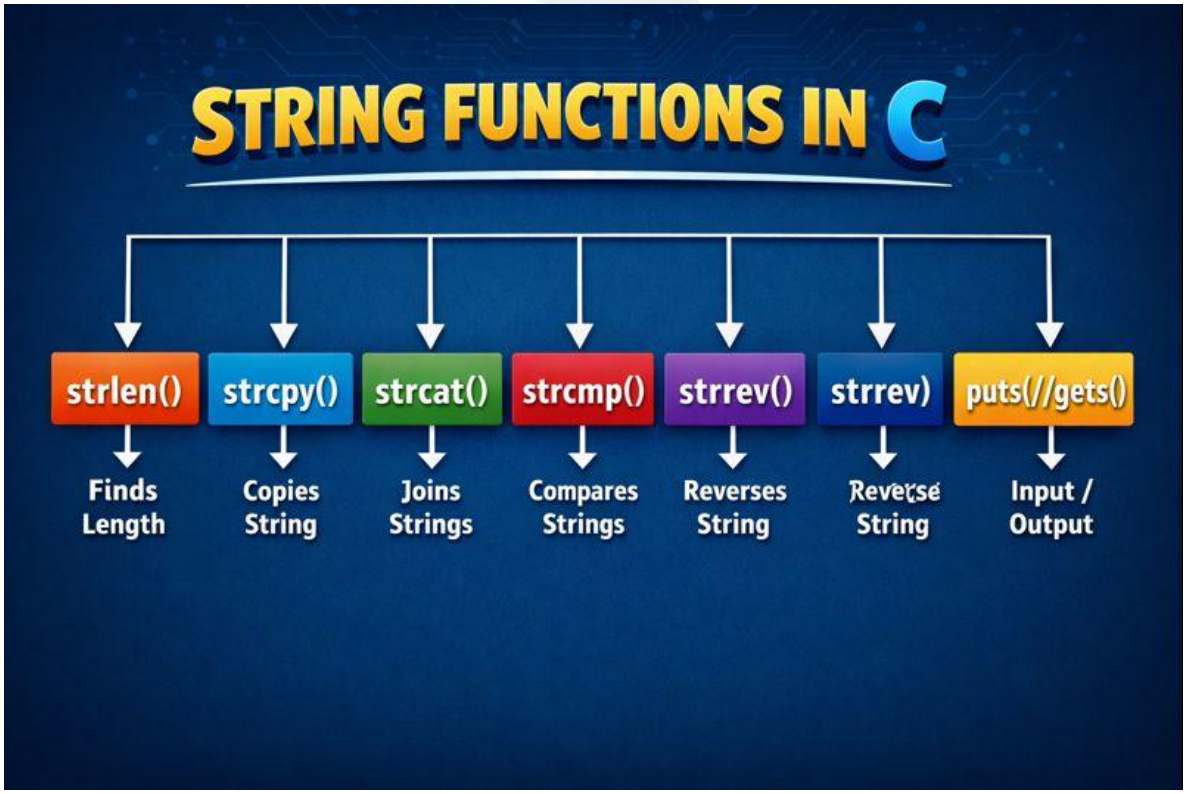
- Accessing elements is slower than normal matrix
- Not suitable for dense matrices
- Requires searching for elements
- Direct indexing is not possible

1. Definition of String ADT

A **String ADT** is an abstract data type that represents a **sequence of characters**.
It defines the **operations that can be performed on strings** without specifying how the string is stored or implemented in memory.

---

2. String Operations



1. Length Operation

The length operation is used to find the number of characters present in a string, excluding the null character (`\0`). This operation helps in validating input size and controlling loops. In C language, the function `strlen()` is used to perform this operation.

Syntax:
```
strlen(string_name);
```

Example:
If the string is `"DATA"`, the length of the string is **4**.

---

2. Copy Operation

The copy operation duplicates the contents of one string into another string. It is useful when the same string data needs to be stored in more than one variable. In C, the `strcpy()` function copies the source string into the destination string.

Syntax:
```
strcpy(destination, source);
```

Example:
If the source string is `"HELLO"`, after copying, the destination string will also contain `"HELLO"`.

---

3. Concatenation Operation

Concatenation is the process of joining two strings into a single string. The second string is appended at the end of the first string. In C language, this operation is performed using the `strcat()` function.

Syntax:
```
strcat(string1, string2);
```

Example:
If `string1 = "DATA"` and `string2 = "BASE"`, the result after concatenation is `"DATABASE"`.

---

4. Comparison Operation

The comparison operation is used to compare two strings to check whether they are equal or not. It compares strings character by character based on their ASCII values. In C, this is done using the `strcmp()` function.

Syntax:
```
strcmp(string1, string2);
```

Example:
If both strings are `"CAT"`, the function returns **0**, indicating that the strings are equal.

1. Text Processing

Strings are widely used for processing text such as words, sentences, and paragraphs. Operations like counting characters, searching a word, replacing text, and reversing words are all done using strings.
Example: Spell checkers, text editors, and document formatting tools.

---

2. Input and Output Operations

User input like names, addresses, passwords, and messages are handled using strings. Output messages displayed on the screen are also strings.
Example:

```
char name[20];
scanf("%s", name);
printf("Welcome %s", name);
```

---

3. File Handling

In file operations, strings are used to read and write text data from files. Each line or word read from a file is treated as a string.
Example: Reading a line from a text file using `fgets()`.

---

4. Database Applications

Strings are used to store and process textual data in databases such as names, email IDs, city names, and queries.
Example: SQL queries sent from C programs are stored as strings.

---

5. Command Line Arguments

Command line inputs provided while running a program are stored as strings.
Example:

```
int main(int argc, char *argv[])
```

Here `argv[]` is an array of strings.

---

6. Web and Internet Applications

Strings are used to handle URLs, form data, user credentials, and messages exchanged between client and server.
Example: Handling HTTP request data.

---

7. Operating System Applications

File names, directory paths, environment variables, and system commands are all strings.
Example:

```
system("dir");
```

---

8. Communication Software

Chat applications, email systems, and messaging apps use strings to send and receive messages.
Example: Text messages in chat applications.

---

9. Game Development

Strings are used for player names, scores, instructions, dialogues, and menu options in games.
Example: Displaying “Game Over” or “Level Completed”.

---

10. Pattern Matching and Searching

Strings are used in applications that require searching patterns such as plagiarism detection, data validation, and compiler design.
Example: Searching keywords in source code.

1. Definition of Pattern Matching

Pattern Matching is the process of **finding the occurrence of a pattern (substring)** within a larger string called **text**.

In pattern matching, the pattern is compared with different parts of the text to check whether it matches completely or not.

📌 **In simple words:**
Pattern matching is used to **search a word or sequence of characters inside another string**.

🔹 **Example:**

- Text: `COMPUTERPROGRAMMING`
- Pattern: `PROGRAM`
- Result: Pattern found starting at position 8

---

2. Naïve Pattern Matching Algorithm

The **Naïve Pattern Matching Algorithm** is the **simplest and most basic method** of pattern matching.
It checks the pattern **at every possible position** in the text.

Working Principle

1. Align the pattern with the beginning of the text.
2. Compare characters one by one.
3. If all characters match → pattern found.
4. If mismatch occurs → shift the pattern by one position.
5. Repeat until the end of the text.

Algorithm (Steps)

Let

- Text = `T` of length `n`
- Pattern = `P` of length `m`

1. For `i = 0` to `n - m`
2. Compare `P[0…m-1]` with `T[i…i+m-1]`
3. If all characters match, report match at position `i`

Example

Text: `AABAACAADAABAABA`
Pattern: `AABA`

Shift	Comparison	Result
0	AABA = AABA	Match
1	ABAA ≠ AABA	No
2	BAAC ≠ AABA	No
9	AABA = AABA	Match

Time Complexity

- **Worst case:** `O(n × m)`
- **Best case:** `O(n)`

📌 This algorithm is easy to understand but inefficient for large texts.

Pattern matching is widely used in computer science and real-world applications where **searching, comparing, and identifying text patterns** is required. The main applications are explained below in **exam-oriented paragraph form**.

3. Applications of Pattern Matching

1. Text Editors

Pattern matching is used in text editors to **find, search, and replace words or phrases** within documents.
Example: Searching a word in MS Word or Notepad using *Find* option.

---

2. Search Engines

Search engines use pattern matching to **locate keywords** entered by users within millions of web pages.
Example: Google searching for a word inside web content.

---

3. Compiler Design

In compiler design, pattern matching is used during **lexical analysis** to identify keywords, operators, identifiers, and symbols from source code.
Example: Recognizing `int`, `if`, `while` in C programs.

---

4. Bioinformatics

Pattern matching is used to find **specific DNA or protein sequences** in biological data.
Example: Matching gene patterns in DNA sequences.

---

5. Plagiarism Detection

Plagiarism detection tools compare documents to **identify copied or similar text patterns**.
Example: Detecting similarity between student assignments.

---

6. Data Validation

Pattern matching is used to **validate input formats** such as email IDs, phone numbers, and passwords.
Example: Checking whether an email follows the correct format.

---

7. Information Retrieval Systems

Used to search required information from **databases and digital libraries**.
Example: Searching records using names or keywords.

---

8. Network Security

Pattern matching helps in **intrusion detection systems** to identify malicious data patterns or viruses.
Example: Detecting virus signatures in network traffic.

---

9. Natural Language Processing (NLP)

Used in language processing tasks such as **word recognition, spell checking, and text analysis**.
Example: Identifying keywords in sentences.

Array

Definition

An **array in C** is a collection of **elements of the same data type**, stored in **contiguous memory locations**, and accessed using a **single variable name with an index**.

📌 **Index starts from 0** in C.

1. Declaration of Array

Points:

- Declares an array and reserves memory
- Specifies data type and size
- No values are assigned at this stage

Syntax:

```
data_type array_name[size];
```

Example:

```
int arr[5];
```

---

2. Initialization of Array

Points:

- Assigns values to array elements
- Values are stored sequentially
- Index starts from 0

Syntax:

```
data_type array_name[size] = {value1, value2, ..., valueN};
```

Example:

```
int arr[5] = {10, 20, 30, 40, 50};
```

3. Array Representation (Index & Memory)

Points:

- Elements stored in contiguous memory
- Each element has a unique index
- Index range is 0 to size-1

Diagram (Representation):

```
Index →    0     1     2     3     4
Array →   10    20    30    40    50
```

4. Accessing Array Elements

Points:

- Accessed using array name and index
- Allows reading or modifying values
- Direct access makes it fast

Syntax:

```
array_name[index];
```

Example:

```
printf("%d", arr[2]);   // Output: 30
```

5. Updating Array Elements

Points:

- Changes value at a specific index
- Uses direct assignment

Syntax:

```
array_name[index] = new_value;
```

Example:

```
arr[1] = 99; // Changes 20 to 99
```

Types of Arrays in C

---

1. One-Dimensional Array

Definition:
A **one-dimensional array** is used to store a **list of elements** of the same data type in a single row.

Points:

- Uses a single index
- Stores elements linearly
- Index starts from 0

Syntax:

```
data_type array_name[size];
```

Example:

```
int a[5] = {1, 2, 3, 4, 5};
```

Neat Diagram:

```
Index →   0   1   2   3   4
Array →   1   2   3   4   5
```

---

2. Two-Dimensional Array

Definition:
A **two-dimensional array** stores data in **rows and columns**, similar to a matrix.

Points:

- Uses two indices (row and column)
- Useful for tables and matrices
- Stored in contiguous memory row-wise

Syntax:

```
data_type array_name[rows][columns];
```

Example:

```
int mat[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

Neat Diagram:

```
Column → 0 1 2 Row 0 1 2 3 Row 1 4 5 6
```

Traversing

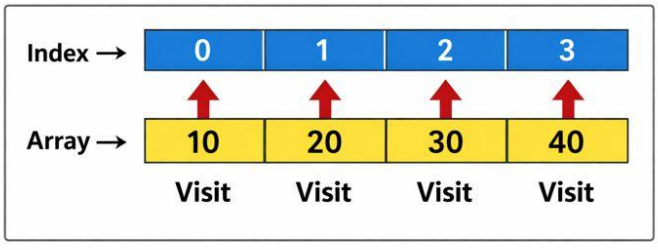
Traversing is the operation of accessing and processing each element of an array exactly once. In this operation, the array elements are visited sequentially starting from the first index (0) up to the last index (n−1). Traversing is mainly used to display the elements of the array or to perform calculations such as finding the sum or average of elements.

Syntax:

```
for(i = 0; i < n; i++)
{
    // process arr[i]
}
```

Example:

```
for(i = 0; i < n; i++) printf("%d ", arr[i]);
```



Insertion

Insertion is the process of adding a new element at a specified position in an array. Since arrays store elements in contiguous memory locations, insertion requires shifting the existing elements one position to the right to create space for the new element. After shifting, the new value is placed at the required index and the size of the array increases by one.

Syntax:

```
for(i = n; i > pos; i--)
    arr[i] = arr[i-1];
arr[pos] = element;
```

Example (Insert 25 at index 2):

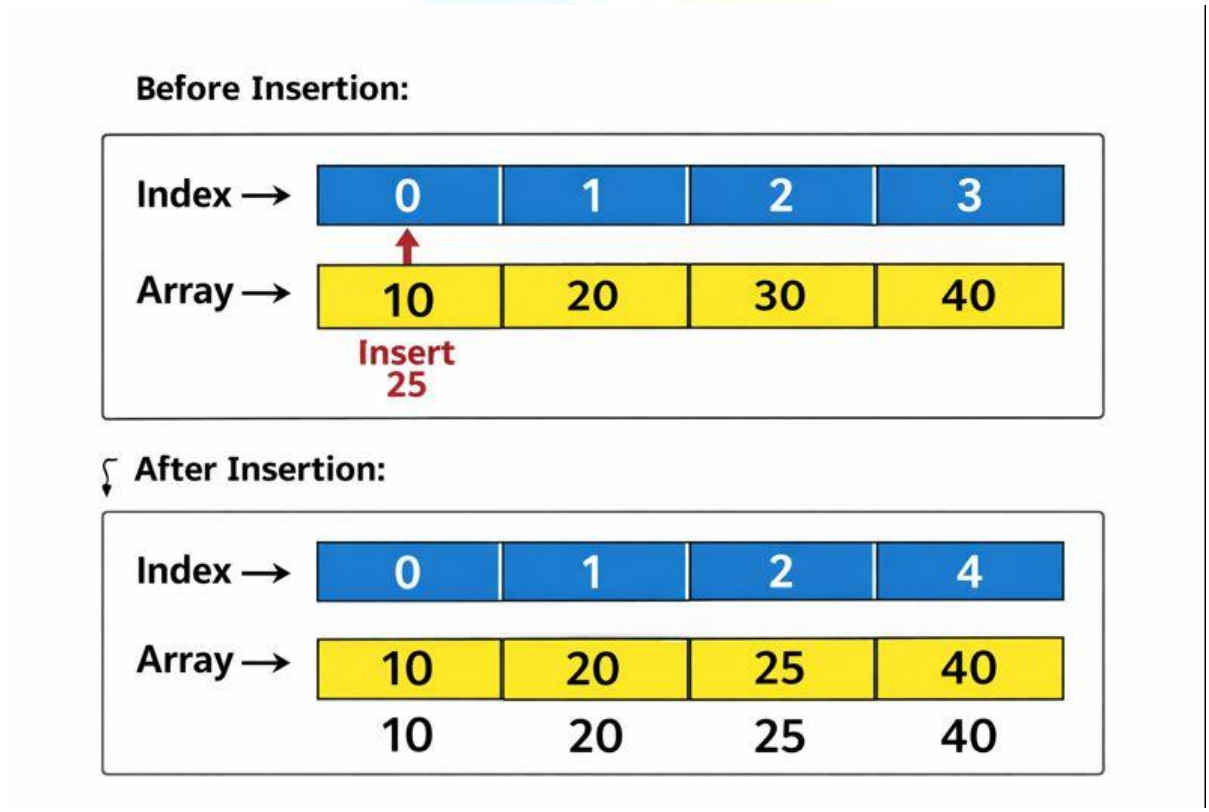
```
for(i = n; i > 2; i--) arr[i] = arr[i-1]; arr[2] = 25;
```

Syntax:

```
for(i = pos; i < n-1; i++)
    arr[i] = arr[i+1];
```

Example (Delete element at index 2):

```
for(i = 2; i < n-1; i++) arr[i] = arr[i+1];
```



Deletion

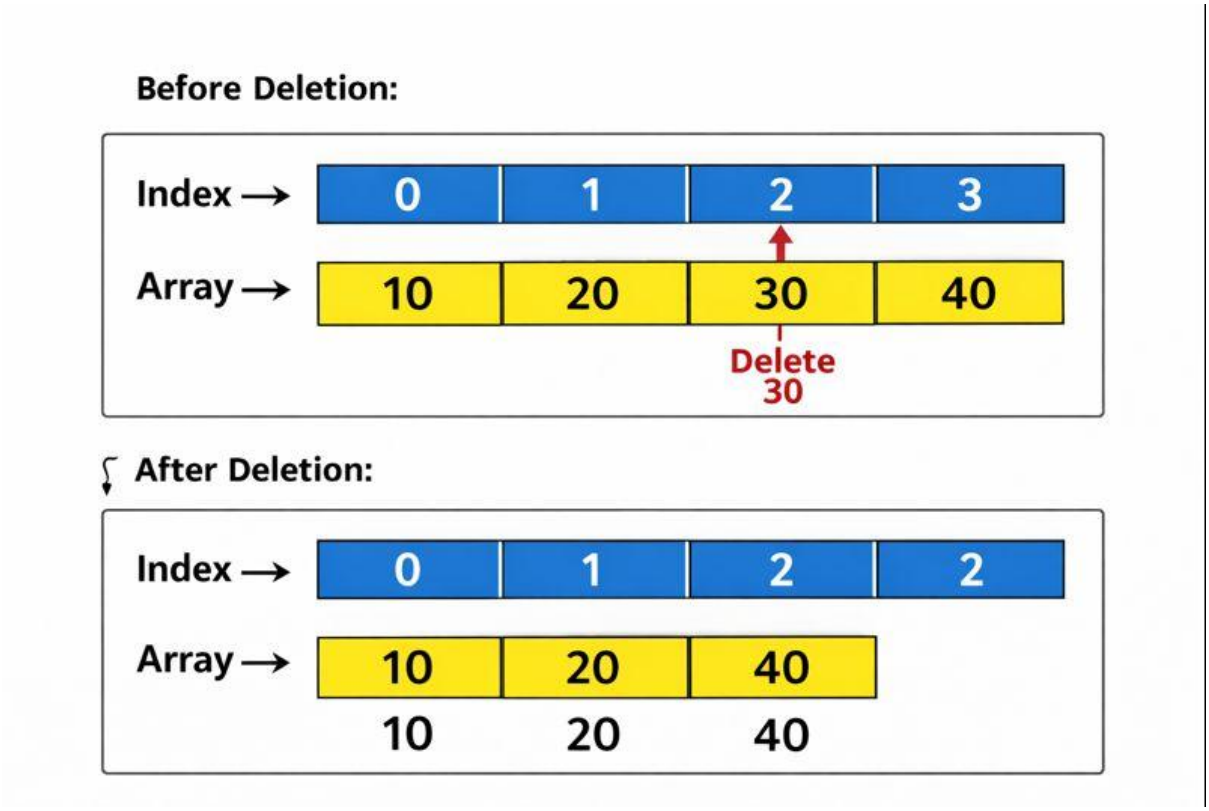
Deletion is the operation of removing an element from a given position in an array. After deleting the element, all the elements following that position are shifted one position to the left to fill the empty space. As a result, the size of the array is reduced by one.

Syntax:

```
for(i = pos; i < n-1; i++)
    arr[i] = arr[i+1];
```

Example (Delete element at index 2):

```
for(i = 2; i < n-1; i++) arr[i] = arr[i+1];}
```



Searching

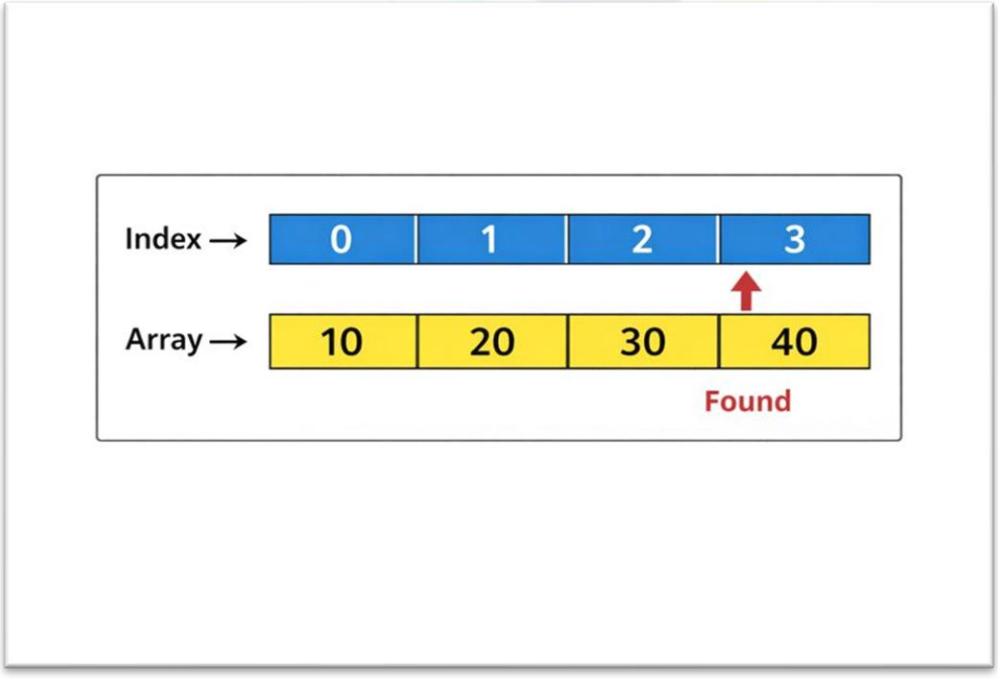
Searching is the operation of finding the location of a particular element in an array. In linear search, each element of the array is compared with the required value one by one until a match is found or all elements are checked. Searching helps in checking the presence of an element and retrieving its position.

for(i = 0; i < n; i++)

```
{
    if(arr[i] == key)
        break;
}
```

Example (Search 40):

```
for(i = 0; i < n; i++)
{
    if(arr[i] == 40)
        printf("Found at index %d", i);
}
```



Updating

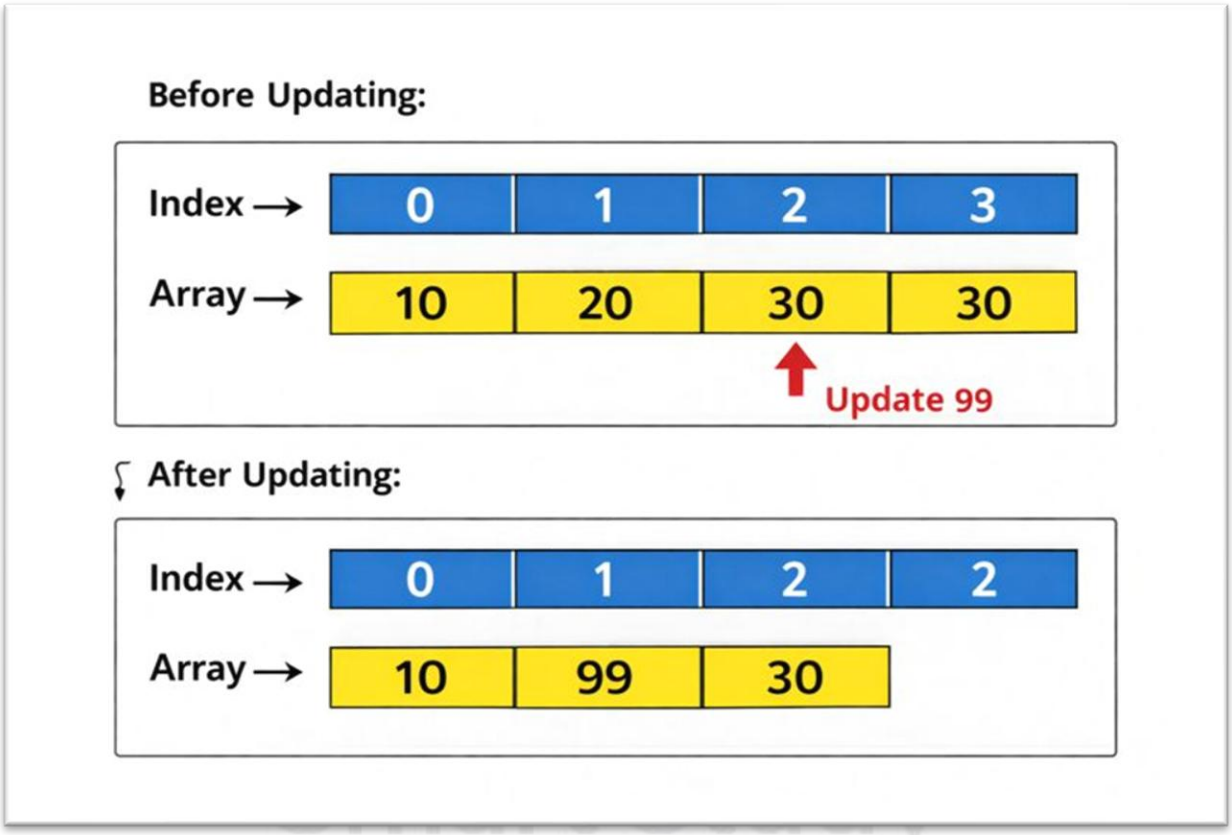
Updating is the operation of modifying the value of an existing element in the array. This is done by directly accessing the element using its index and assigning a new value. Since arrays provide direct access to elements, updating an element is a fast and simple operation.

Syntax:

```
arr[index] = new_value;
```

Example:

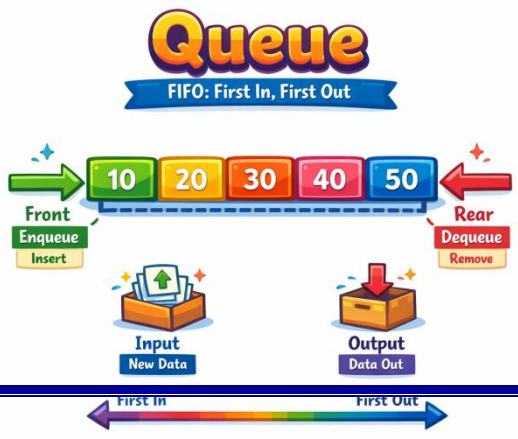
```
arr[1] = 99;
```



UNIT – II : Queues, Linked Lists & Hashing

1.1 Definition of Queue

A **Queue** is a **linear abstract data structure** that stores elements in a sequential manner, where **insertion of elements is performed at one end called the REAR** and **deletion of elements is performed at the**



opposite end called the FRONT.
The order of element processing in a queue follows the **First-In-First-Out (FIFO)** principle, which means that the element inserted earliest is the first one to be removed. This behavior is also referred to as the **Last-In-Last-Out (LILO)** technique.

A queue is described as a **both open-ended data structure** because it allows operations at **two different ends**, unlike a stack where insertion and deletion occur at the same end. This structure ensures fairness and preserves the natural order of data processing.

Queues are particularly useful in applications that require **orderly and sequential handling of data**. They are commonly used in **CPU process scheduling, printer spooling, input/output buffering, network data transmission, and real-world waiting systems** such as queues at service counters. By maintaining the arrival order of elements, queues provide efficient and systematic data management.

Real-World Examples

- Queue at ticket counters
- Bus stops
- Vehicles on a one-way road
- CPU scheduling and printer spooling

Key Features of Queue/ Characteristics of Queue

1. A queue follows the **First-In-First-Out (FIFO)** principle, in which the element inserted first is the first one to be removed from the queue.
2. In a queue, **insertion of elements is always performed at the rear end**, while **deletion of elements is always performed at the front end**.
3. A queue uses **two pointers or indices**, namely **FRONT** and **REAR**, to manage insertion and deletion operations efficiently.
4. The **FRONT pointer** always points to the first element of the queue, whereas the **REAR pointer** points to the last inserted element.
5. A queue is known as a **both open-ended data structure** because operations are carried out at two opposite ends.
6. Unlike stacks, a queue does not allow insertion and deletion at the same end, which helps in preserving the **FIFO order**.
7. A queue supports **overflow and underflow conditions** during its operations.
8. **Overflow** occurs when an insertion is attempted on a full queue, and **underflow** occurs when a deletion is attempted on an empty queue.
9. Queues ensure **systematic and orderly processing of data**, especially where tasks must be handled sequentially.
10. Due to these characteristics, queues are widely used in **operating systems, CPU scheduling, data buffering, printer spooling, and communication systems**.

Representation of Queue

A **Queue** can be represented in **three main ways** in data structures. The method of representation determines how queue elements are stored in memory and how efficiently insertion and deletion operations are carried out. Irrespective of the representation used, all queues operate according to the **FIFO (First-In-First-Out)** principle and are managed using two pointers known as **FRONT** and **REAR**.

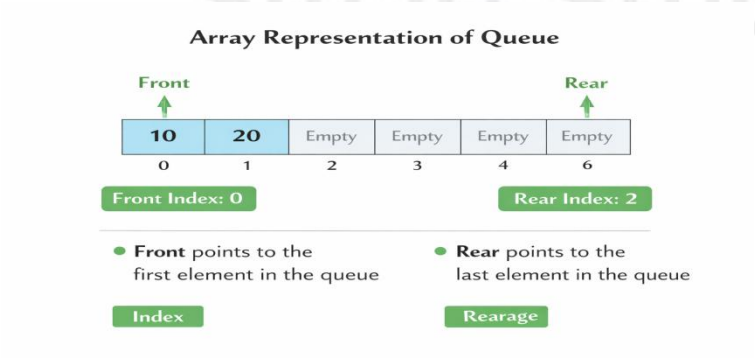
The commonly used representations of a queue are:

1. **Array Representation**
2. **Linked List Representation**
3. **Circular Queue Representation**

1. Array Representation of Queue

In the array representation, the queue is implemented using a **one-dimensional array** of fixed size. The elements of the queue are stored sequentially in the array. Two integer variables, **FRONT** and **REAR**, are used to track the positions of deletion and insertion respectively.

- When the queue is initially empty, both FRONT and REAR are set to **−1**.
- During insertion, the REAR pointer is incremented and the new element is placed at the index indicated by REAR.
- During deletion, the element at the index pointed to by FRONT is removed and FRONT is incremented.



Conditions:

- **Queue Empty:** `FRONT = -1` or `FRONT > REAR`
- **Queue Full:** `REAR = MAXSIZE - 1`

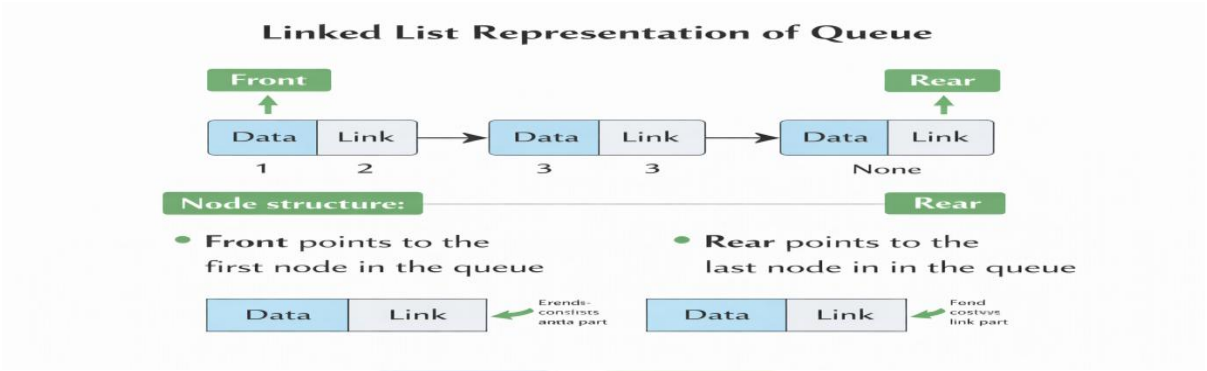
Limitation:
In a simple linear queue, once elements are deleted, the freed memory locations at the

beginning of the array cannot be reused. This leads to **memory wastage**, which is a major drawback of this representation.

2. Linked List Representation of Queue

In the linked list representation, the queue is implemented using a **singly linked list**. Each node of the linked list contains two parts: **data** and a **link** to the next node. Two pointers, **FRONT** and **REAR**, are maintained.

- FRONT points to the first node in the list.
- REAR points to the last node in the list.
- Insertion is performed by creating a new node and linking it at the REAR end.
- Deletion is performed by removing the node pointed to by FRONT.



Advantages:

- The queue size can grow dynamically.
- No memory wastage occurs, as memory is allocated only when required.
- Overflow occurs only when system memory is exhausted.

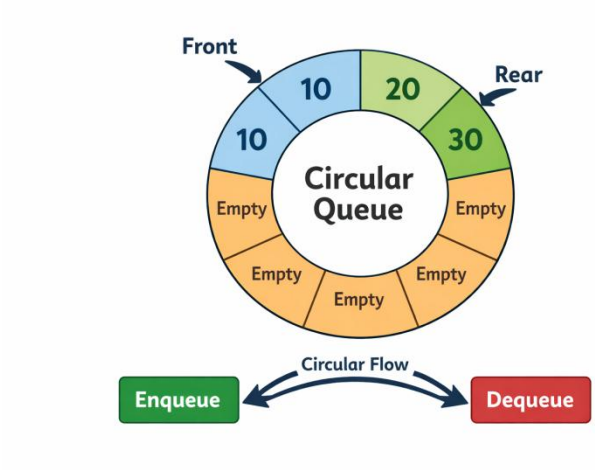
3. Circular Queue Representation

A **Circular Queue** is an improved form of the array-based queue. In this representation, the last position of the array is connected back to the first position, forming a circular structure.

- When the REAR pointer reaches the end of the array, it moves back to the beginning if free space is available.
- This representation allows better utilization of memory.
- Circular queues eliminate the problem of unused spaces that occur in linear queues.

Condition for Full Queue:

```
(REAR + 1) % MAXSIZE = FRONT
```



1.2 Basic Operations on Queue

EnQueue (Insertion)

EnQueue is the operation of adding a new element to the queue. In a queue, insertion is always performed at the **rear end**. Before inserting an element, it is essential to check whether the queue is full in order to avoid the **overflow condition**.

If the queue is not full, the **rear pointer is incremented** and the new element is stored at the position indicated by the rear. If the queue is initially empty, the **front pointer is initialized** before performing insertion. This operation preserves the **FIFO (First-In-First-Out)** order of the queue.

Algorithm for EnQueue Operation

Procedure: EnQueue(data)

```
1. If REAR = MAXSIZE - 1
       Print "Queue Overflow"
       Stop
2. If FRONT = -1
       Set FRONT = 0
3. Set REAR = REAR + 1
4. Set QUEUE[REAR] = data
5. Stop
```

EnQueue Operation

```
int enqueue(int data) {
    if (isFull()) {
        printf("Queue Overflow\n");
        return 0;
    }

    if (front == -1)       // Queue
is initially empty
        front = 0;

    rear++;
    queue[rear] = data;
    return 1;
}
```

DeQueue (Deletion)

DeQueue is the operation of removing an element from the queue. In a queue, deletion is always performed at the **front end**. Before removing an element, it is necessary to check whether the queue is empty to avoid the **underflow condition**.

If the queue is not empty, the element pointed to by the **front pointer** is removed. After deletion, the front pointer is incremented to point to the next element in the queue. If the deleted element was the last element in the queue, both **front and rear pointers are reset**. This operation ensures that the FIFO order of the queue is maintained.

Algorithm for DeQueue Operation

Procedure: DeQueue()

```
1. If FRONT = -1 OR FRONT > REAR
       Print            "Queue
Underflow"
       Stop
2. Set data = QUEUE[FRONT]
3. Set FRONT = FRONT + 1
4. If FRONT > REAR
       Set FRONT = REAR = -1
5. Return data
6. Stop
```

C Code for DeQueue Operation

```
int dequeue() {
    int data;

    if (isEmpty()) {
        printf("Queue Underflow\n");
        return 0;
    }

    data = queue[front];
    front++;

    if (front > rear) {
  // Queue becomes empty after deletion
        front = -1;
        rear = -1;
    }
```

Peek Operation

Peek is the operation of viewing the element present at the **front of the queue** without removing it. This operation is used to identify the next element that will be processed. Before performing the peek operation, it is necessary to check whether the queue is empty to avoid an invalid access.

If the queue is not empty, the element pointed to by the **front pointer** is returned. Since no deletion occurs, the structure of the queue remains unchanged and the FIFO order is preserved.

Algorithm for Peek Operation

Procedure: Peek()

```
1. If FRONT = -1 OR FRONT > REAR
       Print "Queue is Empty"
       Stop
2. Return QUEUE[FRONT]
```

Peek Operation

```
int peek() {
    if (isEmpty()) {
        printf("Queue is
Empty\n");
        return 0;
    }

    return queue[front];
}
```

```
3. Stop
```

isEmpty Operation

isEmpty is the operation used to check whether the queue contains any elements. This operation helps in identifying the **underflow condition** before performing deletion or peek operations.

A queue is considered empty when the **front pointer is equal to −1** or when the **front pointer is greater than the rear pointer**. The isEmpty operation does not modify the queue; it only returns the status of the queue.

Algorithm for isEmpty Operation

Procedure: isEmpty()

```
1. If FRONT = -1 OR FRONT > REAR
       Return TRUE
2. Else
       Return FALSE
3. Stop
```

C Code for isEmpty Operation

```
int isEmpty() {
    if (front == -1 ||
front > rear)
        return 1;
    else
        return 0;
}
```

isFull Operation

isFull is the operation used to check whether the queue has reached its maximum storage capacity. This operation helps in identifying the **overflow condition** before attempting an insertion.

In an array-based queue implementation, the queue is considered full when the **rear pointer reaches the last index of the array**. The isFull operation only checks the status of the queue and does not modify its contents.

Algorithm for isFull Operation

Procedure: isFull()

```
1. If REAR = MAXSIZE - 1
       Return TRUE
2. Else
       Return FALSE
3. Stop
```

C Code for isFull Operation

```
int isFull() {
    if (rear == MAX -
1)
        return 1;
    else
        return 0;
```

Queue ADT Implementation in C all operations using array

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 5

int queue[MAX];
int front = -1;
int rear  = -1;

/* Function Prototypes */
void enqueue(int x);
void dequeue();
void peek();
void display();
int isEmpty();
int isFull();

int main() {
   int choice, item;

   while (1) {
      printf("\n===== QUEUE ADT MENU =====\n");
      printf("1. EnQueue (Insert)\n");
      printf("2. DeQueue (Delete)\n");
      printf("3. Peek\n");
      printf("4. isEmpty\n");
      printf("5. isFull\n");
      printf("6. Display\n");
      printf("7. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
      case 1:
         printf("Enter element to insert: ");
         scanf("%d", &item);
         enqueue(item);
         break;

      case 2:
         dequeue();
         break;

      case 3:
         peek();
         break;
```

```
        case 4:
            if (isEmpty())
                printf("Queue is Empty\n");
            else
                printf("Queue is NOT Empty\n");
            break;

        case 5:
            if (isFull())
                printf("Queue is Full\n");
            else
                printf("Queue is NOT Full\n");
            break;

        case 6:
            display();
            break;

        case 7:
            exit(0);

        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}

/* EnQueue Operation */
void enqueue(int x) {
    if (isFull()) {
        printf("Queue Overflow\n");
        return;
    }

    if (front == -1)
   front = 0;
    rear++;
    queue[rear] = x;
    printf("Inserted %d into queue\n", x);
}

/* DeQueue Operation */
void dequeue() {
    if (isEmpty()) {
        printf("Queue Underflow\n");
        return;
```

```
===== QUEUE ADT MENU =====
1. EnQueue (Insert)
2. DeQueue (Delete)
3. Peek
4. isEmpty
5. isFull
6. Display
7. Exit
Enter your choice: 1
Enter element to insert: 10
Inserted 10 into queue

===== QUEUE ADT MENU =====
Enter your choice: 1
Enter element to insert: 20
Inserted 20 into queue

===== QUEUE ADT MENU =====
Enter your choice: 6
Queue elements: 10 20

===== QUEUE ADT MENU =====
Enter your choice: 3
Front element: 10

===== QUEUE ADT MENU =====
Enter your choice: 2
Deleted element: 10
```

```
    }

    printf("Deleted element: %d\n", queue[front]);
    front++;

    if (front > rear)
        front = rear = -1;
}

/* Peek Operation */
void peek() {
    if (isEmpty()) {
        printf("Queue is Empty\n");
        return;
    }
    printf("Front element: %d\n", queue[front]);
}

/* isEmpty Operation */
int isEmpty() {
    if (front == -1 || front > rear)
        return 1;
    else
        return 0;
}

/* isFull Operation */
int isFull() {
    if (rear == MAX - 1)
        return 1;
    else
        return 0;
}

/* Display Operation */
void display() {
    int i;

    if (isEmpty()) {
        printf("Queue is Empty\n");
        return;
    }

    printf("Queue elements: ");
    for (i = front; i <= rear; i++)
        printf("%d ", queue[i]);
    printf("\n");
}
```

Program: Menu-Driven Queue Using Array with Structure/ Program to Implement a Linear Queue Using Array in C

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 4

struct Queue {
   int q[MAX];
   int front;
   int rear;
};

void initQueue(struct Queue *q) {
   q->front = -1;
   q->rear = -1;
}

void insert_q(struct Queue *q) {
   int item;
   if (q->rear == MAX - 1) {
      printf("Queue Overflow\n");
      return;
   }

   if (q->front == -1)
      q->front = 0;

   q->rear++;
   printf("Enter an item to be inserted: ");
   scanf("%d", &item);
   q->q[q->rear] = item;
   printf("Inserted successfully\n");
}

void delete_q(struct Queue *q) {
   if (q->front == -1 || q->front > q->rear) {
      printf("Queue Underflow (Empty)\n");
      q->front = q->rear = -1;
      return;
   }
```

```
    printf("%d deleted successfully\n", q->q[q->front]);
    q->front++;

    if (q->front > q->rear)
        q->front = q->rear = -1;
}

void display_q(struct Queue *q) {
    if (q->front == -1 || q->front > q->rear) {
        printf("Queue is Empty\n");
        return;
    }

    printf("Queue elements are:\n");
    for (int i = q->front; i <= q->rear; i++)
        printf("| %d | ", q->q[i]);
    printf("\n");
}

int main() {
    struct Queue q;
    int choice;

    initQueue(&q);

    while (1) {
        printf("\n***** MENU *****\n");
        printf("1. INSERT\n2. DELETE\n3. DISPLAY\n4. EXIT\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            insert_q(&q);
            break;
        case 2:
            delete_q(&q);
            break;
        case 3:
            display_q(&q);
            break;
        case 4:
            exit(0);
```

```
        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}
```

Types of Queues

In data structures, queues are classified into different types based on the way elements are inserted, deleted, and managed in memory. The commonly used types of queues are:

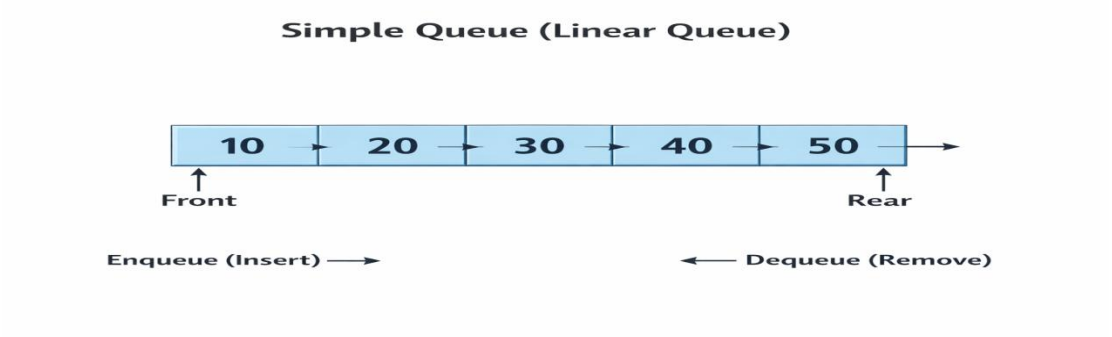
1. Simple Queue (Ordinary / Linear Queue)

A **Simple Queue**, also known as an **Ordinary Queue** or **Linear Queue**, is the most basic type of queue data structure. It follows the **FIFO (First-In-First-Out)** principle, which means that the element inserted first into the queue is the first one to be removed.

In a simple queue, **insertion of elements (EnQueue)** is performed at the **rear end**, while **deletion of elements (DeQueue)** is performed at the **front end**. The queue is commonly implemented using a **one-dimensional array or a linked list** and is managed using two pointers called **FRONT** and **REAR**.

Initially, both **FRONT** and **REAR** are set to **−1**, indicating that the queue is empty. When an element is inserted, the **REAR pointer is incremented** and the element is placed at the rear position. When an element is deleted, the element at the front is removed and the **FRONT pointer is incremented**.

A major drawback of the simple queue when implemented using an array is **memory wastage**. After deleting elements from the front, the empty spaces created at the beginning of the array cannot be reused efficiently, even though free memory is available.



Characteristics of Simple Queue

- A simple queue follows the First In First Out (FIFO) order.
- Insertion of elements is performed at the rear end of the queue.
- Deletion of elements is performed from the front end of the queue.
- Only two pointers, front and rear, are used to manage the queue.
- Once an element is deleted, the space created cannot be reused easily.
- Simple queues may lead to memory wastage after several deletions.
- A simple queue can be implemented using arrays or linked lists.

Example

An example of a simple queue is a **queue at a movie ticket counter**. People stand in a line to buy tickets. The person who arrives first stands at the **front of the queue** and is served first, while new people join the line at the **rear end**. As each person is served, they leave from the front, and the remaining people move forward. This process exactly follows the **FIFO principle** of a simple queue.

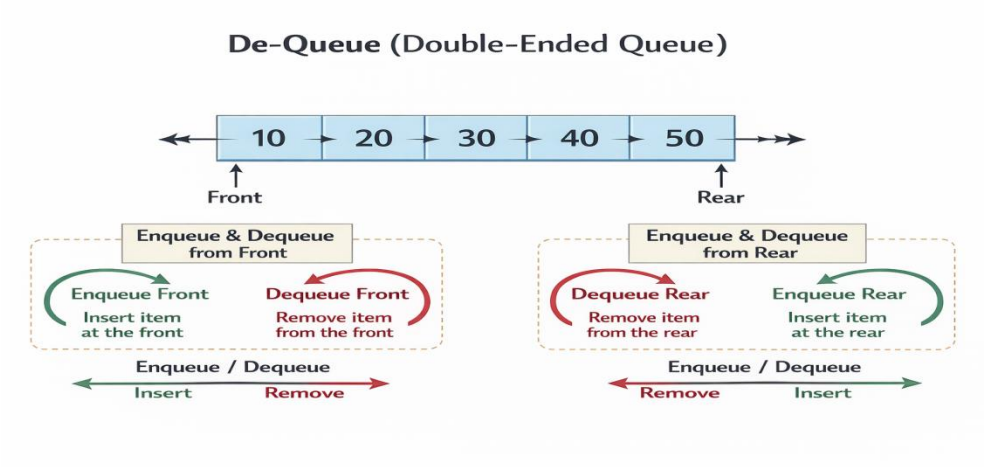
De-Queue (Double-Ended Queue)

A **De-Queue**, also known as a **Double-Ended Queue**, is a special type of queue in which **insertion and deletion of elements can be performed at both ends**, that is, at the **front end** as well as the **rear end**. Unlike a simple queue, which restricts insertion to one end and deletion to the other, a de-queue provides greater flexibility in data handling.

In a de-queue, elements can be added or removed from either end depending on the requirement. Because of this flexibility, a de-queue does not always strictly follow the FIFO principle. It can also behave like a stack in certain situations. De-queues are commonly implemented using arrays or linked lists.

There are two main types of de-queues:

- **Input-restricted de-queue**, where insertion is allowed only at one end but deletion is allowed at both ends.
- **Output-restricted de-queue**, where deletion is allowed only at one end but insertion is allowed at both ends.



Characteristics of De-Queue

- A De-Queue allows insertion of elements at both the front and rear ends.
- It also allows deletion of elements from both the front and rear ends.
- It is more flexible than a simple queue.
- De-Queue does not strictly follow the FIFO principle.
- It can be implemented using arrays or linked lists.
- De-Queue is useful when elements need to be added or removed from both ends.

Example (De-Queue)

Example **of a de-queue** is a **line of people entering and exiting a cinema hall through two doors**.People may **enter the hall from either the front door or the back door**, and similarly, people may **exit from either end** depending on the situation. This ability to add or remove people from **both ends** is similar to how a **de-queue** operates.

C Program: DEQUE Using Array

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 5

int deque[MAX];
int front = -1;
int rear  = -1;

/* Function Prototypes */
void insertFront(int x);
void insertRear(int x);
void deleteFront();
void deleteRear();
void display();

int isFull();
int isEmpty();
```

```
int main() {
    int choice, item;

    while (1) {
        printf("\n===== DEQUE MENU =====\n");
        printf("1. Insert Front\n");
        printf("2. Insert Rear\n");
        printf("3. Delete Front\n");
        printf("4. Delete Rear\n");
        printf("5. Display\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            printf("Enter element: ");
            scanf("%d", &item);
            insertFront(item);
            break;

        case 2:
            printf("Enter element: ");
            scanf("%d", &item);
            insertRear(item);
            break;

        case 3:
            deleteFront();
            break;

        case 4:
            deleteRear();
            break;

        case 5:
            display();
            break;

        case 6:
            exit(0);

        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}

/* Insert at Front */
void insertFront(int x) {
    if (isFull()) {
        printf("DEQUE Overflow\n");
        return;
    }

    if (front == -1)
        front = rear = 0;
    else
        front--;
```

```
    deque[front] = x;
    printf("Inserted %d at front\n", x);
}

/* Insert at Rear */
void insertRear(int x) {
    if (isFull()) {
        printf("DEQUE Overflow\n");
        return;
    }

    if (rear == -1)
        front = rear = 0;
    else
        rear++;

    deque[rear] = x;
    printf("Inserted %d at rear\n", x);
}

/* Delete from Front */
void deleteFront() {
    if (isEmpty()) {
        printf("DEQUE Underflow\n");
        return;
    }

    printf("Deleted %d from front\n", deque[front]);

    if (front == rear)
        front = rear = -1;
    else
        front++;
}

/* Delete from Rear */
void deleteRear() {
    if (isEmpty()) {
        printf("DEQUE Underflow\n");
        return;
    }

    printf("Deleted %d from rear\n", deque[rear]);

    if (front == rear)
        front = rear = -1;
    else
        rear--;
}

/* Check Empty */
int isEmpty() {
    return (front == -1);
}

/* Check Full */
int isFull() {
    return (front == 0 && rear == MAX - 1);
}

/* Display DEQUE */
```

```
void display() {
    int i;

    if (isEmpty()) {
        printf("DEQUE is Empty\n");
        return;
    }

    printf("DEQUE elements: ");
    for (i = front; i <= rear; i++)
        printf("%d ", deque[i]);
    printf("\n");
}
```

Output :

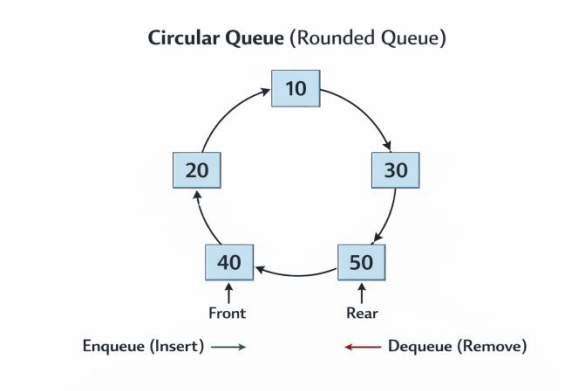
```
Insert Rear → 10
Insert Rear → 20
Insert Front → 5
Display → 5 10 20
Delete Front → 5
Delete Rear → 20
Display → 10
```

Circular Queue (Rounded Queue)

A **Circular Queue**, also known as a **Rounded Queue**, is a modified form of the simple queue in which the **last position of the queue is connected to the first position**, forming a circular structure. This arrangement allows the queue to reuse empty spaces created after deletion.

In a circular queue, when the **REAR pointer reaches the end of the array**, it moves back to the beginning of the array if space is available. This helps in **efficient memory utilization** and overcomes the memory wastage problem of the linear queue.

Insertion of elements is performed at the **rear end**, and deletion of elements is performed at the **front end**, while still following the **FIFO principle**. Circular queues are commonly implemented using arrays and are widely used in applications that require continuous and efficient data handling.



Characteristics of Circular Queue

- A circular queue connects the last position of the queue back to the first position.
- It follows the First In First Out (FIFO) principle.
- The front and rear pointers move in a circular manner.
- When the rear reaches the end, it wraps around to the beginning if space is available.
- Circular queues make efficient use of memory by reusing empty spaces.
- They eliminate the problem of memory wastage found in linear queues.
- Insertion of elements is done at the rear end of the queue.
- Deletion of elements is performed from the front end of the queue.
- Circular queues can be implemented using arrays or linked lists.

Advantages

1. A circular queue ensures efficient use of memory by reusing empty spaces created after deletion.
2. It eliminates the problem of false overflow that occurs in linear queue implementation.
3. The circular structure allows the rear pointer to move back to the beginning when it reaches the end of the array.
4. It avoids unnecessary shifting of elements, making the operations faster and more efficient.
5. Enqueue and Dequeue operations are performed in constant time, improving performance.
6. It is suitable for real-time applications such as buffering and scheduling systems.
7. The circular queue provides better space management compared to a simple linear queue.

Example :

A **circular queue** can be compared to a **wall clock**. After the number **12**, the count starts again from **1** and continues in a circular manner. There is no end point; the sequence keeps repeating in a loop.

This behavior is similar to a circular queue, where after the last position, the next position is again the first.

Program: Circular Queue Using Array in C

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 5

int cqueue[MAX];
int front = -1;
int rear = -1;

/* Function Prototypes */
void enqueue(int x);
void dequeue();
void display();
int isFull();
int isEmpty();

int main() {
    int choice, item;

    while (1) {
        printf("\n===== CIRCULAR QUEUE MENU =====\n");
        printf("1. EnQueue\n");
        printf("2. DeQueue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            printf("Enter element: ");
            scanf("%d", &item);
            enqueue(item);
            break;

        case 2:
            dequeue();
            break;

        case 3:
            display();
            break;

        case 4:
            exit(0);

        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}

/* EnQueue Operation */
```

```
void enqueue(int x) {
    if (isFull()) {
        printf("Circular Queue Overflow\n");
        return;
    }

    if (isEmpty()) {
        front = rear = 0;
    } else {
        rear = (rear + 1) % MAX;
    }

    cqueue[rear] = x;
    printf("Inserted %d\n", x);
}

/* DeQueue Operation */
void dequeue() {
    if (isEmpty()) {
        printf("Circular Queue Underflow\n");
        return;
    }

    printf("Deleted %d\n", cqueue[front]);

    if (front == rear) {
        front = rear = -1;
    } else {
        front = (front + 1) % MAX;
    }
}

/* Display Operation */
void display() {
    int i;

    if (isEmpty()) {
        printf("Circular Queue is Empty\n");
        return;
    }

    printf("Circular Queue elements: ");

    i = front;
    while (1) {
        printf("%d ", cqueue[i]);
        if (i == rear)
            break;
        i = (i + 1) % MAX;
    }
    printf("\n");
}

/* Check Empty */
int isEmpty() {
    return (front == -1);
}

/* Check Full */
int isFull() {
    return ((rear + 1) % MAX == front);
```

```
}
```

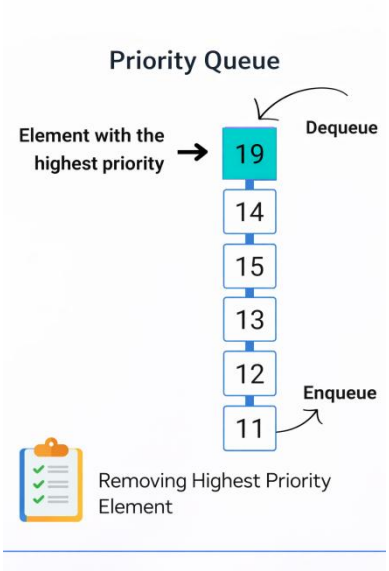
Output

```
===== CIRCULAR QUEUE MENU =====
1. EnQueue
2. DeQueue
3. Display
4. Exit
Enter your choice: 1
Enter element: 10
Inserted 10

Enter your choice: 1
Enter element: 20
Inserted 20

Enter your choice: 1
Enter element: 30
Inserted 30

Enter your choice: 3
Circular Queue elements: 10 20 30

Enter your choice: 2
Deleted 10

Enter your choice: 3
Circular Queue elements: 20 30
```

Priority Queue

A **Priority Queue** is a special type of queue in which **each element is assigned a priority**. Elements are processed based on their priority rather than the order in which they arrive. An element with a **higher priority is removed before** an element with a lower priority. If two elements have the same priority, they are processed according to the **FIFO (First-In-First-Out)** principle.

Priority queues are commonly used in situations where some tasks are more important and need to be handled before others.



Types of Priority Queue

1. **Ascending Priority Queue**
   – Lower priority number ⇒ higher priority
2. **Descending Priority Queue**
   – Higher priority number ⇒ higher priority

Characteristics of Priority Queue

- Elements are arranged in any order internally.
- Only the element with the **highest or lowest priority** is allowed to be deleted at a time.
- Priority queues can be implemented using **arrays** or **linked lists**.
- The **heap data structure** is commonly used to implement priority queues efficiently.

Simple Example

Consider a **hospital emergency room**. Patients are not treated based only on their arrival time. Patients with **serious conditions** are treated first, even if they arrive later, while patients with **minor issues** wait longer. If two patients have the same level of seriousness, they are treated in the order in which they arrived.

This behavior clearly shows how a **priority queue** works.

Priority Queue Implementation Using Array

Program

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 5

struct PriorityQueue {
   int data[MAX];
   int priority[MAX];
   int rear;
};

/* Function Prototypes */
void create(struct PriorityQueue *pq);
void insert(struct PriorityQueue *pq, int item, int pr);
void delete(struct PriorityQueue *pq);
void peek(struct PriorityQueue *pq);
void display(struct PriorityQueue *pq);
int isEmpty(struct PriorityQueue *pq);
int isFull(struct PriorityQueue *pq);

int main() {
   struct PriorityQueue pq;
   int choice, item, pr;

   create(&pq);

   while (1) {
      printf("\n===== PRIORITY QUEUE ADT MENU =====\n");
      printf("1. Insert\n");
      printf("2. Delete\n");
      printf("3. Peek\n");
      printf("4. Display\n");
      printf("5. Exit\n");
      printf("Enter choice: ");
      scanf("%d", &choice);

      switch (choice) {
      case 1:
         printf("Enter element: ");
         scanf("%d", &item);
         printf("Enter priority: ");
         scanf("%d", &pr);
         insert(&pq, item, pr);
         break;

      case 2:
         delete(&pq);
         break;

      case 3:
```

```
            peek(&pq);
            break;

        case 4:
            display(&pq);
            break;

        case 5:
            exit(0);

        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}

/* Create Operation */
void create(struct PriorityQueue *pq) {
    pq->rear = -1;
}

/* Insert Operation */
void insert(struct PriorityQueue *pq, int item, int pr) {
    int i;

    if (isFull(pq)) {
        printf("Priority Queue Overflow\n");
        return;
    }

    if (pq->rear == -1) {
        pq->rear = 0;
        pq->data[0] = item;
        pq->priority[0] = pr;
    } else {
        for (i = pq->rear; i >= 0 && pq->priority[i] > pr; i--) {
            pq->data[i + 1] = pq->data[i];
            pq->priority[i + 1] = pq->priority[i];
        }
        pq->data[i + 1] = item;
        pq->priority[i + 1] = pr;
        pq->rear++;
    }

    printf("Inserted %d with priority %d\n", item, pr);
}

/* Delete Operation */
void delete(struct PriorityQueue *pq) {
```

```
    if (isEmpty(pq)) {
        printf("Priority Queue Underflow\n");
        return;
    }

    printf("Deleted element: %d (Priority %d)\n",
            pq->data[0], pq->priority[0]);

    for (int i = 0; i < pq->rear; i++) {
        pq->data[i] = pq->data[i + 1];
        pq->priority[i] = pq->priority[i + 1];
    }

    pq->rear--;
}

/* Peek Operation */
void peek(struct PriorityQueue *pq) {
    if (isEmpty(pq)) {
        printf("Priority Queue is Empty\n");
        return;
    }
    printf("Highest Priority Element: %d (Priority %d)\n",
            pq->data[0], pq->priority[0]);
}

/* Display Operation */
void display(struct PriorityQueue *pq) {
    if (isEmpty(pq)) {
        printf("Priority Queue is Empty\n");
        return;
    }

    printf("Element\tPriority\n");
    for (int i = 0; i <= pq->rear; i++)
        printf("%d\t%d\n", pq->data[i], pq->priority[i]);
}

/* isEmpty */
int isEmpty(struct PriorityQueue *pq) {
    return (pq->rear == -1);
}

/* isFull */
int isFull(struct PriorityQueue *pq) {
    return (pq->rear == MAX - 1);
}
```

Sample Execution

Insert → (10, 2)
Insert → (20, 1)
Insert → (30, 3)
Display:
20 1
10 2
30 3
Delete → 20

Applications of Queues

- **CPU Scheduling:**
  Queues are widely used in operating systems to manage processes waiting for CPU execution. Processes are placed in a ready queue and are executed in the order decided by the scheduling algorithm, ensuring systematic and fair use of the CPU.
- **Printer Spooling:**
  In printer systems, multiple print requests are handled using a queue. Print jobs are stored in the queue and processed one by one in the order they are received, preventing data loss and ensuring proper job execution.
- **Data Buffering:**
  Queues are used in buffering data during input and output operations. Examples include keyboard input buffers and network data packets, where data is temporarily stored and processed sequentially.
- **Breadth-First Search (BFS):**
  Queues are essential in graph and tree traversal algorithms such as BFS. They help in visiting nodes level by level, ensuring all adjacent nodes are explored before moving to the next level.
- **Traffic Management Systems:**
  Queues are used to control traffic at signals and toll booths. Vehicles are served in the order they arrive, maintaining fairness and reducing congestion.
- **Server Request Handling:**
  In web servers and database servers, incoming client requests are stored in a queue. The server processes these requests one at a time, helping to manage heavy traffic efficiently.
- **Simulation Systems:**
  Queues are used in simulation models such as banking systems, call centers, and ticket booking systems to represent waiting lines and analyze service efficiency.
- **Inter-Process Communication:**
  Queues help in communication between processes by temporarily storing data until the receiving process is ready, ensuring synchronization and smooth data transfer.

Singly Linked List

A **singly linked list** is a linear and dynamic data structure in which data elements are stored in the form of nodes. Each node consists of two parts: a **data field**, which stores the actual information, and a **link field**, which stores the address of the next node in the list. The nodes are connected sequentially through pointers, forming a chain-like structure.

In a singly linked list, each node points only to its immediate successor, which means traversal of the list is possible in only one direction, from the first node to the last node. The address of the first node is stored in a pointer called the **head**. The link field of the last node contains a **NULL** value, indicating the end of the list.

Unlike arrays, singly linked lists do not require contiguous memory locations. Memory for each node is allocated dynamically at runtime, allowing the list to grow or shrink as needed. This makes singly linked lists flexible and efficient when frequent insertion and deletion operations are required.

However, direct access to any element is not possible, and traversal must always begin from the head node. Despite this limitation, singly linked lists are widely used in applications such as dynamic memory management, implementation of stacks and queues, and representation of polynomial expressions.

A **singly linked list** is a collection of nodes in which each node contains:

- **Data**: stores the value
- **Next**: stores the address of the next node in the list

Traversal in a singly linked list is possible **only in one direction**.

Types of Linked list

- Singly Linked List
- Doubly Linked List
- Circular Linked List

Characteristics of Singly Linked List

- A singly linked list is a linear data structure made up of nodes.
- Each node contains two parts: a data field and a link field.
- The link field stores the address of the next node in the list.
- The first node is accessed using a pointer called the head.
- The last node of the list contains a NULL pointer.
- Traversal of the list is possible only in one direction.
- Memory allocation is dynamic and occurs at runtime.
- The list does not require contiguous memory locations.
- Direct access to elements is not possible.
- Extra memory is required to store the link pointer.

Basic Concepts of Singly Linked List

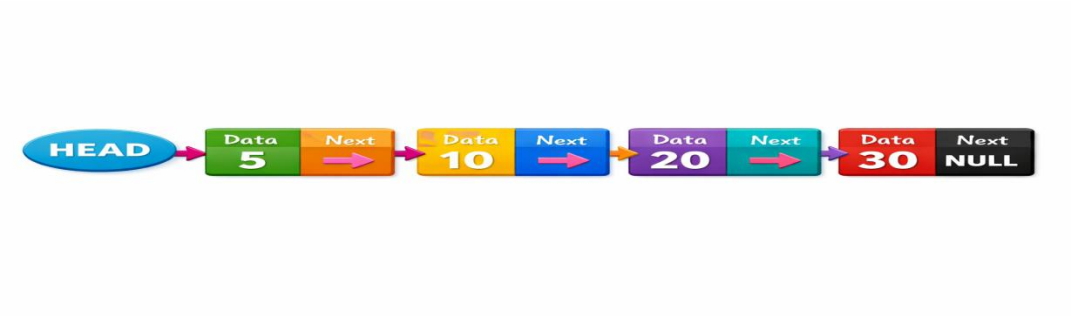
- Each element in a singly linked list is called a **node**.
- Each node contains **two fields**:
    1. **Information field (Data field)** – stores the actual data element.
    2. **Next address field (Link field)** – stores the address of the next node in the list.
- Nodes are stored in **non-contiguous memory locations**.
- The **last node** of the list contains a **NULL pointer**, which indicates the end of the linked list.
- Traversal of a singly linked list is possible **only in one direction**, from the first node to the last node.

IMPLEMENTATION OF SINGLY LINKED LIST

A **Singly Linked List** can be implemented in **TWO main ways**.

1. **Static implementation using arrays(**memory size is known in advance)
2. **Dynamic implementation using pointers (malloc)**

Example of single linked list



Step by step Procedure for inserting values in single linked list

Step 1: Define the Structure of a Node

In a **singly linked list**, each element is called a **node**.
A node is a self-referential structure because it contains a pointer to another node of the same type. Each node consists of **two fields**:

1. Data Field

- The **data field** is used to store the actual value or information.
- It can store any type of data such as integer, float, character, or even a structure.
- In this implementation, the data field stores an integer value.

2. Next Field

- The **next field** is a pointer that stores the **address of the next node** in the linked list.
- This field establishes the connection between nodes.
- For the last node, the next field is set to **NULL**, which indicates the end of the list.

Structure Definition

```
struct node
{
    int data;
    struct node *next;
};
```

Explanation

- `struct node` defines a new data type called `node`.
- `data` holds the value of the node.
- `next` holds the address of the next node.
- This structure is the foundation of the singly linked list.

Step 2: Declare the HEAD Pointer

The **HEAD pointer** is used to store the address of the **first node** in the linked list.

```
struct node *head = NULL;
```

Explanation

- Initially, the linked list is **empty**.
- Since there is no node at the beginning, `head` is assigned **NULL**.
- `head` always points to the first node of the list.
- All operations such as creation, traversal, insertion, and deletion start from `head`.
- If `head` is NULL, it means the list contains no nodes.

Step 3: Create the First Node

```
head = (struct node *)malloc(sizeof(struct
node));
head->data = 5;
head->next = NULL;
```

Explanation

- Memory is dynamically allocated for the first node using `malloc()`.
- The allocated memory is enough to store one node.
- The value `5` is stored in the data field.
- Since this is the first and only node at this moment, its `next` field is set to **NULL**.
- The `head` pointer now stores the address of this node.

Step 4: Create the Second Node

```
struct node *second;
second = (struct node
*)malloc(sizeof(struct node));
second->data = 10;
second->next = NULL;

head->next = second;
```

Explanation

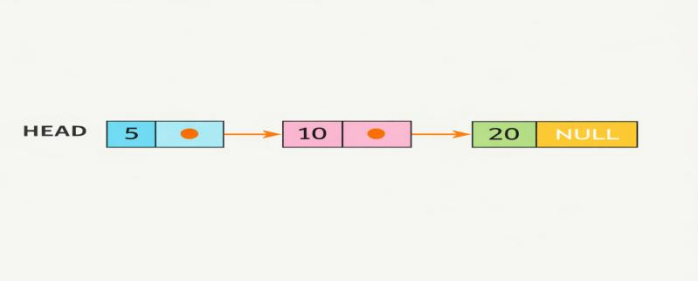
- A new node called `second` is created.
- Memory is allocated dynamically.
- The value `10` is stored in the data field.

- The `next` field is temporarily set to NULL.
- The `next` field of the first node (`head->next`) is updated to store the address of the second node.
- This links the first node to the second node.

Diagram Mapping

Step 5: Create the Third Node

```
struct node *third;
third = (struct node
*)malloc(sizeof(struct
node));
third->data = 20;
third->next = NULL;

second->next = third;
```

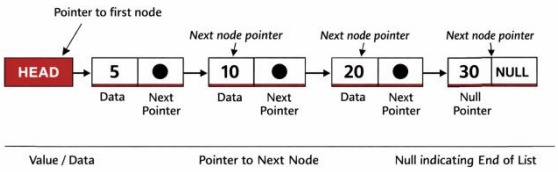


Explanation

- A third node is created dynamically.
- The value `20` is stored in the data field.
- The `next` field is set to NULL.
- The `next` of the second node is updated to point to the third node.
- Now, three nodes are connected sequentially.

Step 6: Create the Fourth (Last) Node

```
struct node *fourth;
fourth = (struct node
*)malloc(sizeof(struct node));
fourth->data = 30;
fourth->next = NULL;

third->next = fourth;
```



Explanation

- The fourth node is created and memory is allocated.
- The value `30` is stored in the data field.
- The `next` field is set to NULL.
- The `next` of the third node is updated to point to the fourth node.
- Since the fourth node's `next` is NULL, it becomes the **last node**.

Step 7: Traversal of the Linked List

Traversal is the process of **visiting each node exactly once**, starting from the head node.

```
struct node *temp = head;

while (temp != NULL)
{
    printf("%d ", temp->data);
    temp = temp->next;
}
```

Explanation

- A temporary pointer `temp` is initialized with `head`.
- The loop continues as long as `temp` is not NULL.
- The data of the current node is printed.
- `temp` moves to the next node using `temp->next`.
- When `temp` becomes NULL, traversal stops.

Output

```
5 10 20 30
```

Singly Linked List Operations

1. **Creation** – Creating nodes and forming the linked list
2. **Traversal** – Visiting and displaying all nodes
3. **Insertion** – Adding a new node
    - At the beginning
    - At the end
    - At a specific position
4. **Deletion** – Removing a node
    - From the beginning
    - From the end
    - From a specific position
5. **Searching** – Finding a particular element in the list
6. **Updating** – Modifying the data of an existing node
7. **Counting** – Finding the total number of nodes

Creation of Singly Linked List

Creation of a singly linked list is done by dynamically creating nodes and linking them sequentially. Each node contains a data field and a link to the next node. The pointer **HEAD** stores the address of the first node, and the last node's link is set to **NULL** to mark the end of the list.

- Initialize the **HEAD** pointer to `NULL`.
- Create a new node using dynamic memory allocation.
- Store the data value in the data field of the node.
- Set the link (next) field of the new node to `NULL`.
- If the list is empty, make **HEAD** point to the new node.
- Otherwise, link the previous node to the new node.
- Repeat the process to create more nodes and form the linked list.

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *next;
};

int main() {
   struct node *head = NULL, *temp, *new
   int n, i;

   printf("Enter number of nodes: ");
   scanf("%d", &n);

   for(i = 0; i < n; i++) {
      newnode = (struct node*)malloc(sizeof(struct node));
      printf("Enter data: ");
      scanf("%d", &newnode->data);
      newnode->next = NULL;

      if(head == NULL) {
         head = newnode;
         temp = head;
      } else {
         temp->next = newnode;
         temp = newnode;
      }
   }

   return 0;
}
```

Traversal

Traversal is the process of visiting each node of a singly linked list starting from the **HEAD** pointer. During traversal, the data in each node is accessed or displayed, and the pointer moves to the next node until it reaches **NULL**, which indicates the end of the list.

```
void traversal(struct node
*head) {
    struct node *temp =
head;

    while (temp != NULL) {
        printf("%d ", temp-
>data);
        temp = temp->next;
    }
}
```

- Start from the **HEAD** of the singly linked list.
- Assign a temporary pointer to **HEAD**.
- Visit the current node and access or display its data.
- Move the pointer to the next node using the link field.
- Repeat the process until the pointer becomes **NULL**.
- Stop traversal when the end of the list is reached.

Types of Insertion in a Singly Linked List

Insertion in a singly linked list can be performed in the following three ways:

1. **Insertion at the Beginning**
2. **Insertion at the End**
3. **Insertion at a Specific Position**

1. Insertion at the Beginning

Insertion at the beginning adds a new node **before the first node** of the linked list. The `next` of the new node is set to the current head, and the head pointer is updated to the new node. This operation does not require traversal.

```
void insert_begin()
{
    struct node *new_node;
    new_node = (struct node
*)malloc(sizeof(struct node));
    scanf("%d", &new_node->data);
    new_node->next = head;
    head = new_node;
}
```

- Create a new node using dynamic memory allocation.
- Store the given data in the new node.
- Set the link (next) of the new node to point to the current **HEAD**.
- Update **HEAD** to point to the new node.
- The new node becomes the first node of the linked list.

2. Insertion at the End

Insertion at the end adds a new node **after the last node** of the linked list. The list is traversed until the last node is reached, and its `next` is updated to point to the new node.

- Create a new node using dynamic memory allocation.

```
void insert_end()
{
    struct node *new_node, *temp;
    new_node = (struct node
*)malloc(sizeof(struct node));
    scanf("%d", &new_node->data);
    new_node->next = NULL;

        head = new_node;
```

- Store the given data in the new node.
- Set the link (next) of the new node to **NULL**.
- If the list is empty, make **HEAD** point to the new node.
- Otherwise, traverse the list until the last node is reached.
- Set the link of the last node to point to the new node.
- The new node becomes the last node of the linked list.

3. Insertion at a Specific Position

Insertion at a specific position inserts a new node **at a given position** in the linked list. The list is traversed up to the required position, and links are adjusted without shifting elements.

- Start from the **HEAD** of the singly linked lis
- Read the position at which the new node is t
- Create a new node using dynamic memory a
- Store the given data in the new node.
- If the position is **1**, insert the node at the beg
- Otherwise, traverse the list until the node jus
- Set the link of the new node to point to the n
- Update the link of the previous node to point
- Stop the process.

```
void insert_position(int pos)
{
    struct node *new_node, *temp;
    int i;
    new_node = (struct node
*)malloc(sizeof(struct node));
    scanf("%d", &new_node->data);

    temp = head;
    for (i = 1; i < pos - 1; i++)
        temp = temp->next;

    new_node->next = temp->next;
    temp->next = new_node;
}
```

Deletion in a Singly Linked List

1. Deletion at the Beginning

Deletion at the beginning removes the **first node** of the singly linked list. The head pointer is updated to point to the next node, and the memory of the deleted node is freed.

- Start from the **HEAD** of the singly linked list.
- If the list is empty, no deletion is possible.
- Store the address of the first node in a tempor
- Move **HEAD** to point to the next node.
- Free the memory of the deleted first node.
- Stop the process.

```
void delete_begin()
{
    struct node *temp;
    if (head == NULL)
        return;

    temp = head;
    head = head->next;
    free(temp);
}
```

2. Deletion at the End

Deletion at the end removes the **last node** of the linked list. The list is traversed to reach the second-last node, whose `next` pointer is set to NULL, and the last node is deleted.

```
void delete_end()
{
    struct node *temp,

    if (head == NULL)
```

- Start from the **HEAD** of the singly linked list.
- If the list is empty, no deletion is possible.
- If the list contains only one node, delete it and set **HEAD** to `NULL`.
- Otherwise, traverse the list to reach the second last node.
- Set the link of the second last node to **NULL**.
- Delete (free) the last node.
- Stop the process.

3. Deletion at a Specific Position

Deletion at a specific position removes a node from a **given position** in the singly linked list. The list is traversed up to the previous node, and links are adjusted to bypass the deleted node.

```
void delete_position(int pos)
{
    struct node *temp, *prev;
    int i;

    if (head == NULL)
        return;

    if (pos == 1)
    {
        temp = head;
        head = head->next;
```

- Start from the **HEAD** of the singly linked list.
- Read the position at which the node is to be deleted.
- If the position is **1**, move **HEAD** to the next node and delete the first node.
- Otherwise, traverse the list until the node just before the given position is reached.
- Change the link of the previous node to point to the node after the one to be deleted.
- Free the memory of the deleted node.
- Stop the process.

Searching – Finding a Particular Element in the List

Searching in a singly linked list is the process of traversing the list from the **HEAD** node and comparing each node's data with the required element. If a match is found, the search is successful; otherwise, traversal continues until **NULL** is reached, indicating that the element is not present in the list.

- Start traversal from the **HEAD** node of the linked list.
- Compare the data of each node with the required element.
- If a match is found, the search is successful and the process stops.
- If no match is found, move to the next node using the link field.
- Continue until the pointer reaches **NULL**.
- If **NULL** is reached without a match, the

```
int search(struct node *head,
int key) {
    struct node *temp = head;

    while (temp != NULL) {
        if (temp->data == key)
            return 1;    //
element found
        temp = temp->next;
    }
    return 0;            //
element not found
}
```

element is not present in the list.

Updating – Modifying an Element in the List

Updating in a singly linked list is the process of changing the data value of a node. The list is traversed from the **HEAD** pointer to find the required element, and once found, the old value is replaced with a new value.

- Start traversal from the **HEAD** of the linked list.
- Compare the data of each node with the element to be updated.
- When the required element is found, replace the old value with the new value.
- If the element is not found after reaching **NULL**, display an appropriate message.
- Stop the process after updating or after completing traversal.

```
int update(struct node *head, int
oldVal, int newVal) {
    struct node *temp = head;

    while (temp != NULL) {
        if (temp->data == oldVal) {
            temp->data = newVal;    //
update value
            return 1;               //
update successful
        }
        temp = temp->next;
    }
    return 0;   // element not found
}
```

Counting – Number of Nodes in the List

Counting in a singly linked list is the process of traversing the list from the **HEAD** pointer and counting each node until **NULL** is reached. The final count represents the total number of nodes in the list.

- Initialize a counter variable and set it to zero.
- Assign a temporary pointer to the **HEAD** of the linked list.
- Traverse the list node by node using the link field.
- Increment the counter each time a node is visited.
- Continue traversal until the pointer reaches **NULL**.

```
int countNodes(struct node *head) {
    int count = 0;
    struct node *temp = head;

    while (temp != NULL) {
        count++;
        temp = temp->next;
    }
    return count;
}
```

The final value of the counter gives the total number of nodes in the list

Dynamic Implementation of Singly Linked List ADT with all operations

```
#include <stdio.h>
#include <stdlib.h>
```

```
/* Node structure */
struct node {
    int data;
    struct node *next;
};

struct node *head = NULL;

/* Create */
void create(int n) {
    struct node *newnode, *temp;
    for (int i = 0; i < n; i++) {
        newnode = (struct node*)malloc(sizeof(struct node));
        printf("Enter data: ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        if (head == NULL) {
            head = newnode;
            temp = head;
        } else {
            temp->next = newnode;
            temp = newnode;
        }
    }
}

/* Display */
void display() {
    struct node *temp = head;
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

/* Search */
void search(int key) {
    struct node *temp = head;
    while (temp != NULL) {
        if (temp->data == key) {
            printf("Element found\n");
            return;
        }
        temp = temp->next;
    }
    printf("Element not found\n");
```

```
}

/* Update */
void update(int oldVal, int newVal) {
    struct node *temp = head;
    while (temp != NULL) {
        if (temp->data == oldVal) {
            temp->data = newVal;
            printf("Element updated\n");
            return;
        }
        temp = temp->next;
    }
    printf("Element not found\n");
}

/* Count */
void count() {
    int c = 0;
    struct node *temp = head;
    while (temp != NULL) {
        c++;
        temp = temp->next;
    }
    printf("Total nodes = %d\n", c);
}

/* Delete */
void deleteNode(int key) {
    struct node *temp = head, *prev = NULL;

    if (head == NULL) {
        printf("List is empty\n");
        return;
    }

    if (head->data == key) {
        head = head->next;
        free(temp);
        printf("Node deleted\n");
        return;
    }

    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Element not found\n");
        return;
    }
```

```
    prev->next = temp->next;
    free(temp);
    printf("Node deleted\n");
}

int main() {
    int choice, n, key, oldVal, newVal;

    do {
        printf("\n--- Singly Linked List ADT ---\n");
        printf("1. Create\n");
        printf("2. Display\n");
        printf("3. Search\n");
        printf("4. Update\n");
        printf("5. Count\n");
        printf("6. Delete\n");
        printf("0. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter number of nodes: ");
                scanf("%d", &n);
                create(n);
                break;
            case 2:
                display();
                break;
            case 3:
                printf("Enter element to search: ");
                scanf("%d", &key);
                search(key);
                break;
            case 4:
                printf("Enter old value and new value: ");
                scanf("%d %d", &oldVal, &newVal);
                update(oldVal, newVal);
                break;
            case 5:
                count();
                break;
            case 6:
                printf("Enter element to delete: ");
                scanf("%d", &key);
                deleteNode(key);
                break;
        }
    } while (choice != 0);

    return 0;
}
--- Singly Linked List ADT ---
```

```
1. Create
2. Display
3. Search
4. Update
5. Count
6. Delete
0. Exit

Enter choice: 1
Enter number of nodes: 3
Enter data: 10
Enter data: 20
Enter data: 30

Enter choice: 2
10 -> 20 -> 30 -> NULL

Enter choice: 3
Enter element to search: 20
Element found

Enter choice: 4
Enter old value and new value: 20 25
Element updated

Enter choice: 2
10 -> 25 -> 30 -> NULL

Enter choice: 5
Total nodes = 3

Enter choice: 6
Enter element to delete: 10
Node deleted

Enter choice: 2
25 -> 30 -> NULL

Enter choice: 0
--------------------------------------
```

Inserting Dynamic Values in a Singly Linked List

```
#include <stdio.h>
#include <stdlib.h>
/* Definition of node structure */
struct node
{
   int data;
   struct node *next;
};
```

```
/* Head pointer declaration */
struct node *head = NULL;

/* Function to create singly linked list */
void create()
{
   struct node *new_node, *temp;
   int n, i;

   printf("Enter number of nodes: ");
   scanf("%d", &n);

   for (i = 1; i <= n; i++)
   {
      new_node = (struct node *)malloc(sizeof(struct node));

      printf("Enter data for node %d: ", i);
      scanf("%d", &new_node->data);

      new_node->next = NULL;

      if (head == NULL)
      {
         head = new_node;
         temp = new_node;
      }
      else
      {
         temp->next = new_node;
         temp = new_node;
      }
   }
}

/* Function to display the linked list */
void display()
{
   struct node *temp;

   if (head == NULL)
   {
      printf("Linked List is empty\n");
      return;
```

```
    }

    temp = head;
    printf("Singly Linked List:\n");

    while (temp != NULL)
    {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

/* Main function */
int main()
{
    create();
    display();
    return 0;
}
```

Output Example

```
Enter total number of nodes: 4
Enter data for first node: 5
Enter data for node 2: 10
Enter data for node 3: 20
Enter data for node 4: 30

Output:
5 10 20 30
```

Stack Implementation Using Linked List

Introduction

A **stack** is a linear data structure in which the element inserted most recently is the first one to be removed, following the **Last In First Out (LIFO)** rule. When a stack is implemented using a **linked list**, each element is stored in a dynamically created node, and all stack operations are carried out at a single end of the list, known as the **TOP**. Since memory is allocated as needed, this method does not impose a fixed size on the stack, thereby providing

greater flexibility and better utilization of memory compared to an array-based implementation..

Basic Principle

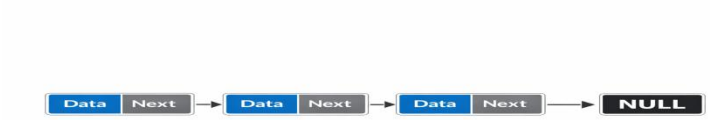
- In this approach, the stack is implemented using a **singly linked list**.
- Each node of the linked list consists of two parts:
  - a **data field** that stores the element, and
  - a **link field** that holds the address of the next node.
- A pointer known as **TOP** is maintained to keep track of the element at the top of the stack.
- When the value of **TOP** is `NULL`, it indicates that the stack contains no elements and is therefore empty.

Node Structure

Each node in the linked list stack has the following form:

```
| Data | Next |
```

Stack Representation



- Insertion and deletion occur **only at TOP**
- This ensures the **LIFO order**

Stack Operations Using Linked List

---

1. PUSH Operation (Insertion)

PUSH is the operation of inserting a new element at the **top of the stack**. A new node is created, its `next` pointer is linked to the current `TOP`, and then `TOP` is updated to point to the new node. This operation takes **O(1)** time.

Steps involved:

1. Allocate memory for a new node
2. Check for memory allocation failure (overflow)
3. Store the data in the new node
4. Make the new node point to the current TOP
5. Update TOP to point to the new node

After this operation, the newly inserted element becomes the **topmost element**.

PUSH Algorithm

```
PUSH(x):
1. Create a new node
2. If memory allocation fails:
      Display "Stack Overflow"
      Exit
3. newNode->data = x
4. newNode->next = TOP
5. TOP = newNode
```

Code

```
void push(int x) {
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));

    if (newNode == NULL) {
        printf("Stack Overflow\n");
        return;
    }

    newNode->data = x;
    newNode->next = top;
    top = newNode;

    printf("Pushed %d into stack\n", x);
}
```

---

2. POP Operation (Deletion)

POP is the operation of removing the element from the **top of the stack**. The `TOP` pointer is moved to the next node and the removed node is deleted.

Steps involved:

1. Check whether the stack is empty (underflow)
2. Store the current TOP node in a temporary pointer
3. Move TOP to the next node
4. Free the memory of the deleted node

This operation maintains the **LIFO behavior**.

POP Algorithm

```
POP():
1. If TOP == NULL:
      Display "Stack Underflow"
      Exit
2. temp = TOP
3. TOP = TOP->next
4. Free(temp)
```

POP Code

```
void pop() {
    struct node *temp;

    if (top == NULL) {
        printf("Stack Underflow\n");
        return;
    }

    temp = top;
    printf("Popped element: %d\n", temp->data);
    top = top->next;
    free(temp);
}
```

---

4. PEEK Operation

PEEK is the operation of viewing the element present at the **top of the stack** without removing it. If the stack is not empty, the value stored in `TOP` is returned**.**

- Used to inspect the stack
- Does not modify stack structure

PEEK Algorithm

```
PEEK():
1. If TOP == NULL:
      Display "Stack is Empty"
2. Else:
      Display TOP->data
```

PEEK Code

```
void peek() {
    if (top == NULL) {
        printf("Stack is Empty\n");
    } else {
```

```
        printf("Top element: %d\n", top->data);
    }
}
```

---

4. isEmpty Operation

The stack is said to be **empty** if the TOP pointer is `NULL`.

isEmpty Code

```
int isEmpty() {
    return (top == NULL);
}
```

5. Display Operation

The **display operation** prints all elements of the stack from **top to bottom**.

- Traversal starts from TOP
- Continues until NULL is reached

Display Code

```
void display() {
    struct node *temp;

    if (top == NULL) {
        printf("Stack is Empty\n");
        return;
    }

    printf("Stack elements:\n");
    temp = top;
    while (temp != NULL) {
        printf("%d\n", temp->data);
        temp = temp->next;
    }
}
```

```
#include <stdio.h>
#include <stdlib.h>

// Node structure
struct node {
    int data;
    struct node *next;
};

struct node *top = NULL;    // Top pointer

// PUSH Operation
void push(int value) {
    struct node *newnode;
```

```
    newnode = (struct node*)malloc(sizeof(struct node));

    if (newnode == NULL) {
        printf("Stack Overflow\n");
        return;
    }

    newnode->data = value;
    newnode->next = top;
    top = newnode;

    printf("%d inserted into stack\n", value);
}

// POP Operation
void pop() {
    struct node *temp;

    if (top == NULL) {
        printf("Stack Underflow\n");
        return;
    }

    temp = top;
    printf("%d deleted from stack\n", top->data);
    top = top->next;
    free(temp);
}

// PEEK Operation
void peek() {
    if (top == NULL) {
        printf("Stack is empty\n");
    } else {
        printf("Top element is %d\n", top->data);
    }
}

// DISPLAY Operation
void display() {
    struct node *temp = top;

    if (temp == NULL) {
        printf("Stack is empty\n");
        return;
    }

    printf("Stack elements are:\n");
    while (temp != NULL) {
        printf("%d\n", temp->data);
        temp = temp->next;
    }
}

// MAIN Function
int main() {
    int choice, value;

    while (1) {
        printf("\n--- STACK USING LINKED LIST ---\n");
        printf("1. Push\n");
```

```
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                push(value);
                break;

            case 2:
                pop();
                break;

            case 3:
                peek();
                break;

            case 4:
                display();
                break;

            case 5:
                printf("Exiting program...\n");
                exit(0);

            default:
                printf("Invalid choice\n");
        }
    }

    return 0;
}
--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to insert: 10
10 inserted into stack

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to insert: 20
20 inserted into stack

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
```

```
4. Display
5. Exit
Enter your choice: 1
Enter value to insert: 30
30 inserted into stack

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
Stack elements are:
30
20
10

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 3
Top element is 30

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 2
30 deleted from stack

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
Stack elements are:
20
10

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 5
Exiting program...
```

Queue Using Linked List –

A **Queue** is a linear data structure that follows the **FIFO (First In, First Out)** principle. This means the element inserted first is removed first, similar to a line of people waiting for a

service. In a queue, insertion takes place at the **rear** end, and deletion takes place at the **front** end.

When a queue is implemented using a **linked list**, dynamic memory allocation is used to create nodes as needed. Each node consists of two parts: a **data field** to store the value and a **link (next pointer)** to connect to the next node. Two pointers are maintained in this implementation:

- **Front** → Points to the first element of the queue.
- **Rear** → Points to the last element of the queue.

Initially, both front and rear are set to NULL, indicating that the queue is empty. The linked list implementation removes the size limitation of array-based queues, as memory is allocated dynamically. Overflow occurs only when the system memory is full, and underflow occurs when deletion is attempted on an empty queue.

Thus, queue operations using a linked list provide efficient insertion and deletion with better memory utilization.

Queue Operations Using Linked List

Enqueue Operation

The **Enqueue** operation is used to insert a new element into the queue. In a linked list implementation, insertion always takes place at the **rear** end. A new node is created dynamically and linked to the last node of the queue. If the queue is empty (both front and rear are NULL), the new node becomes both the front and rear of the queue. Otherwise, the current rear node's next pointer is updated to the new node, and the rear pointer is moved to this new node. This operation takes constant time because no traversal is required.

- **Algorithm – Enqueue**
  1. Create a new node using dynamic memory allocation.
  2. If memory allocation fails, print "Queue Overflow" and stop.
  3. Assign value to newnode->data.
  4. Set newnode->next = NULL.
  5. If rear == NULL
     - Set front = rear = newnode.
  6. Else
     - Set rear->next = newnode.
     - Set rear = newnode.
  7. Stop.

Enqueue

```
void enqueue(int value) {
   struct node *newnode;
   newnode = (struct node*)malloc(sizeof(struct node));

   if (newnode == NULL) {
```

```
        printf("Queue Overflow\n");
        return;
    }

    newnode->data = value;
    newnode->next = NULL;

    if (rear == NULL)
        front = rear = newnode;
    else {
        rear->next = newnode;
        rear = newnode;
    }
}
```

Dequeue Operation

The **Dequeue** operation removes an element from the queue. Since a queue follows the FIFO principle, deletion always occurs at the **front** end. If the queue is empty (front == NULL), the operation cannot be performed and it results in **Queue Underflow**. Otherwise, the node pointed to by front is removed, and the front pointer is updated to the next node. If after deletion the queue becomes empty, both front and rear are set to NULL. This operation also runs in constant time.

- **Algorithm – Dequeue**
  1. If front == NULL
     - Print "Queue Underflow" and stop.
  2. Store front in a temporary variable.
  3. Set front = front->next.
  4. If front == NULL
     - Set rear = NULL.
  5. Free the temporary node.
  6. Stop.

Dequeue

```
void dequeue() {
    struct node *temp;

    if (front == NULL) {
        printf("Queue Underflow\n");
        return;
    }

    temp = front;
    front = front->next;
```

```
    if (front == NULL)
        rear = NULL;

    free(temp);
}
```

Peek Operation

The **Peek** operation is used to view the element present at the front of the queue without removing it. This operation helps in checking the next element that will be deleted. If the queue is empty, a message is displayed. Otherwise, the data stored in the node pointed to by front is shown. Since it directly accesses the front node, it executes in constant time.

Algorithm – Peek

1. If front == NULL
    - Print "Queue is empty".
2. Else
    - Print front->data.
3. Stop.

Peek

```
void peek() {
    if (front == NULL)
        printf("Queue is empty\n");
    else
        printf("Front element = %d\n", front->data);
}
```

Display Operation

The **Display** operation prints all elements of the queue from front to rear. A temporary pointer is used to traverse the linked list starting from front. Each node's data is printed until the pointer reaches NULL. This operation requires visiting every node, so its time complexity is linear.

Algorithm – Display

1. If front == NULL
    - Print "Queue is empty" and stop.
2. Set temp = front.
3. While temp != NULL
    - Print temp->data.
    - Move temp = temp->next.
4. Stop.

Display

```
void display() {
   struct node *temp;

   if (front == NULL) {
      printf("Queue is empty\n");
      return;
   }

   temp = front;
   while (temp != NULL) {
      printf("%d ", temp->data);
      temp = temp->next;
   }
   printf("\n");
}
```

Queue implementation using Linked list

```
#include <stdio.h>
#include <stdlib.h>

/* Node Structure */
struct node {
    int data;
    struct node *next;
};

struct node *front = NULL;
struct node *rear = NULL;

/* Enqueue Operation */
void enqueue(int value) {
    struct node *newnode;

    newnode = (struct node*)malloc(sizeof(struct node));

    if (newnode == NULL) {
        printf("Queue Overflow\n");
        return;
    }

    newnode->data = value;
    newnode->next = NULL;

    if (rear == NULL) {
        front = rear = newnode;
    } else {
        rear->next = newnode;
        rear = newnode;
    }

    printf("%d inserted into queue\n", value);
}

/* Dequeue Operation */
void dequeue() {
```

```
    struct node *temp;

    if (front == NULL) {
        printf("Queue Underflow\n");
        return;
    }

    temp = front;
    printf("%d deleted from queue\n", temp->data);

    front = front->next;

    if (front == NULL)
        rear = NULL;

    free(temp);
}

/* Peek Operation */
void peek() {
    if (front == NULL)
        printf("Queue is empty\n");
    else
        printf("Front element = %d\n", front->data);
}

/* Display Operation */
void display() {
    struct node *temp;

    if (front == NULL) {
        printf("Queue is empty\n");
        return;
    }

    temp = front;
    printf("Queue elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

/* Count Operation */
void count() {
    int c = 0;
    struct node *temp = front;

    while (temp != NULL) {
        c++;
        temp = temp->next;
    }

    printf("Total elements = %d\n", c);
}

/* Main Function */
int main() {
    int choice, value;
```

```
    do {
        printf("\n===== QUEUE USING LINKED LIST =====\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Count\n");
        printf("0. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            printf("Enter value: ");
            scanf("%d", &value);
            enqueue(value);
            break;

        case 2:
            dequeue();
            break;

        case 3:
            peek();
            break;

        case 4:
            display();
            break;

        case 5:
            count();
            break;

        case 0:
            printf("Exiting program...\n");
            break;

        default:
            printf("Invalid choice\n");
        }

    } while (choice != 0);

    return 0;
}
===== QUEUE USING LINKED LIST =====
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Count
0. Exit
Enter your choice: 1
Enter value: 10
10 inserted into queue

===== QUEUE USING LINKED LIST =====
1. Enqueue
2. Dequeue
3. Peek
```

```
4. Display
5. Count
0. Exit
Enter your choice: 1
Enter value: 20
20 inserted into queue

===== QUEUE USING LINKED LIST =====
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Count
0. Exit
Enter your choice: 1
Enter value: 30
30 inserted into queue

===== QUEUE USING LINKED LIST =====
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Count
0. Exit
Enter your choice: 4
Queue elements: 10 20 30

===== QUEUE USING LINKED LIST =====
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Count
0. Exit
Enter your choice: 3
Front element = 10

===== QUEUE USING LINKED LIST =====
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Count
0. Exit
Enter your choice: 5
Total elements = 3

===== QUEUE USING LINKED LIST =====
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Count
0. Exit
Enter your choice: 2
10 deleted from queue

===== QUEUE USING LINKED LIST =====
1. Enqueue
2. Dequeue
```

```
3. Peek
4. Display
5. Count
0. Exit
Enter your choice: 4
Queue elements: 20 30

===== QUEUE USING LINKED LIST =====
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Count
0. Exit
Enter your choice: 0
Exiting program...
```

A **polynomial** is an algebraic expression formed by combining variables and constants using the operations of addition, subtraction, and multiplication. The defining feature of a polynomial is that the variable is raised only to **non-negative integer powers** (0, 1, 2, 3, …). Polynomials are fundamental in mathematics because they are used to represent quantities that change in a regular pattern. In computer science, polynomials are important in areas such as data structures, numerical methods, algorithm analysis, and symbolic computation. They are often stored and manipulated using arrays or linked lists to perform operations like addition, multiplication, and evaluation.

A polynomial is made up of individual parts called **terms**. Each term contains three components:

- A **coefficient**, which is a fixed numerical value
- A **variable**, which represents an unknown or changing quantity
- An **exponent**, which tells how many times the variable is multiplied by itself

For example, in the term $5x^3$:

- 5 is the coefficient
- $x$is the variable
- 3 is the exponent

The exponent must always be a **non-negative integer** (0, 1, 2, 3, …). This condition is what makes an expression a polynomial.

General Form

The general form of a polynomial in one variable is:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Where:

- $a_n, a_{n-1}, \ldots, a_0$are constants
- $n$is a non-negative integer
- $n$represents the **degree** of the polynomial

Important Characteristics

1. The exponent of the variable must be a **whole number (0, 1, 2, 3, ...)**.
2. Polynomials do not contain variables in the denominator.

3. Polynomials do not contain negative or fractional exponents.

Expressions like $\frac{1}{x}$, x^{-2}, or $\sqrt{x}$are **not** polynomials.

Types of Polynomials

1. Based on Number of Terms

- **Monomial** – One term (e.g., $5x^2$)
- **Binomial** – Two terms (e.g., $x + 3$)
- **Trinomial** – Three terms (e.g., $x^2 + 2x + 1$)

2. Based on Degree

- **Linear Polynomial** – Degree 1
- **Quadratic Polynomial** – Degree 2
- **Cubic Polynomial** – Degree 3

Representation of Polynomial in Data Structures

A polynomial is composed of several terms, and each term contains two essential pieces of information: a **coefficient** and an **exponent**. When we work with polynomials in mathematics, we write them symbolically. However, in computer science, we must store them in memory using suitable data structures so that operations such as addition, subtraction, multiplication, and evaluation can be performed efficiently.

The way a polynomial is represented internally affects:

- Memory usage
- Speed of operations
- Simplicity of implementation
- Flexibility for handling different types of polynomials

There are two major approaches to representing polynomials:

1. **Array Representation**
2. **Linked List Representation**

This representation helps in performing operations like:

- Polynomial Addition
- Polynomial Subtraction
- Polynomial Multiplication
- Polynomial Evaluation.

1 Array Representation

In array representation, the polynomial is treated as a sequence of coefficients arranged according to the powers of the variable. The key idea is that the **position (index) of the array corresponds to the exponent**, and the **value stored at that position represents the coefficient** of that exponent.

This method assumes that the polynomial has terms for most exponents between 0 and the highest degree.

If we have:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

We create an array such that:

- array[0] = a_0
- array[1] = a_1
- array[2] = a_2
- ...
- array[n] = a_n

The array directly mirrors the mathematical structure of the polynomial.

◆ **Example**

$$P(x) = 6x^4 + 0x^3 + 3x^2 + 2x + 5$$

This polynomial can be represented as:

Index (Exponent)	**0**	**1**	**2**	**3**	**4**
Coefficient	5	2	3	0	6

Even though the coefficient of $x^3$is 0, we still allocate space for it because arrays are continuous memory structures.

2 Linked List Representation

In linked list representation, a polynomial is viewed as a **collection of independent terms**, rather than a continuous sequence. Each term is stored as a separate node containing:

- Coefficient
- Exponent
- Pointer to the next term

This structure reflects the idea that only meaningful (non-zero) terms need to be stored.
The nodes are usually arranged in either ascending or descending order of exponents to make operations easier.

◆ **Example**

Consider:

$$P(x) = 8x^6 + 4x^2 + 7$$

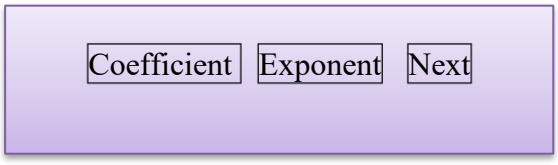
Instead of storing empty positions for exponents 5, 4, 3, and 1, the linked list stores only:

Coefficient	**Exponent**
8	6
4	2
7	0

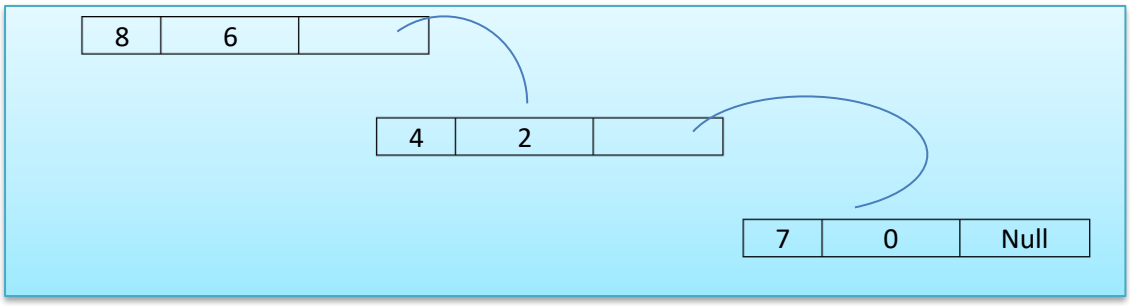
Node Structure Concept

In linked list representation, each term is stored as a node.
Each node contains:

- coefficient
- exponent
- next pointer (link to next term)



Linking the Nodes




```
#include <stdio.h>
#include <stdlib.h>

/* Structure for Polynomial Node */
struct node {
   int coef;
   int exp;
   struct node *next;
};

struct node *head = NULL;

/* Function to create a new node */
struct node* createNode(int c, int e) {
   struct node *newnode;
   newnode = (struct node*)malloc(sizeof(struct node));

   if (newnode == NULL) {
      printf("Memory allocation failed\n");
      exit(1);
   }

   newnode->coef = c;
   newnode->exp = e;
   newnode->next = NULL;

   return newnode;
}

/* Insert in descending order of exponent */
void insert(int c, int e) {
   struct node *newnode = createNode(c, e);
   struct node *temp;

   if (head == NULL || head->exp < e) {
      newnode->next = head;
      head = newnode;
   } else {
      temp = head;
```

```
        while (temp->next != NULL && temp->next->exp > e) {
            temp = temp->next;
        }
        newnode->next = temp->next;
        temp->next = newnode;
    }
}

/* Display polynomial */
void display() {
    struct node *temp = head;

    if (temp == NULL) {
        printf("Polynomial is empty\n");
        return;
    }

    printf("Polynomial: ");
    while (temp != NULL) {
        printf("%dx^%d", temp->coef, temp->exp);
        if (temp->next != NULL)
            printf(" + ");
        temp = temp->next;
    }
    printf("\n");
}

/* Main Function */
int main() {
    int n, coef, exp, i;

    printf("Enter number of terms: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        printf("Enter coefficient and exponent: ");
        scanf("%d %d", &coef, &exp);
        insert(coef, exp);
    }

    display();

    return 0;
}
```

Output

```
Enter number of terms: 3
Enter coefficient and exponent: 8 6
Enter coefficient and exponent: 4 2
```

Enter coefficient and exponent: 7 0
Polynomial: 8x^6 + 4x^2 + 7x^0

Algorithm: Add Two Polynomials

Polynomial addition means combining two polynomials by adding the coefficients of terms that have the same exponent.
For example:

$$P_1(x) = 5x^3 + 4x^2 + 2$$
$$P_2(x) = 3x^3 + 2x + 7$$

After addition:

$$P(x) = (5 + 3)x^3 + 4x^2 + 2x + (2 + 7)$$
$$P(x) = 8x^3 + 4x^2 + 2x + 9$$

Step 1: Initialize

1. Create an empty linked list result.
2. Set pointer p1 to head of poly1.
3. Set pointer p2 to head of poly2.

Step 2: Compare Terms

4. While p1 is not NULL AND p2 is not NULL, repeat:
    - If p1->exp == p2->exp
        - Add coefficients.
        - Create a new node with:
            - Coefficient = p1->coef + p2->coef
            - Exponent = p1->exp
        - Insert this node into result.
        - Move p1 and p2 to next nodes.
    - Else if p1->exp > p2->exp
        - Copy term from poly1 into result.
        - Move p1 to next node.
    - Else
        - Copy term from poly2 into result.
        - Move p2 to next node.

Step 3: Add Remaining Terms

5. While p1 is not NULL:
    - Copy remaining terms of poly1 into result.
    - Move p1 forward.
6. While p2 is not NULL:
    - Copy remaining terms of poly2 into result.
    - Move p2 forward.
    -

Step 4: Return Result

7. Return the head of result.

```
#include <stdio.h>
#include <stdlib.h>

/* Structure for Polynomial Node */
struct node {
   int coef;
   int exp;
   struct node *next;
};

/* Function to create new node */
struct node* createNode(int c, int e) {
   struct node *newnode;
   newnode = (struct node*)malloc(sizeof(struct node));
   newnode->coef = c;
   newnode->exp = e;
   newnode->next = NULL;
   return newnode;
}

/* Insert node in descending order of exponent */
void insert(struct node **head, int c, int e) {
   struct node *newnode = createNode(c, e);
   struct node *temp;

   if (*head == NULL || (*head)->exp < e) {
      newnode->next = *head;
      *head = newnode;
   } else {
      temp = *head;
      while (temp->next != NULL && temp->next->exp > e)
         temp = temp->next;

      newnode->next = temp->next;
      temp->next = newnode;
   }
}

/* Function to add two polynomials */
struct node* addPolynomials(struct node *p1, struct node *p2) {
   struct node *result = NULL;

   while (p1 != NULL && p2 != NULL) {

      if (p1->exp == p2->exp) {
         insert(&result, p1->coef + p2->coef, p1->exp);
         p1 = p1->next;
         p2 = p2->next;
```

```
        }
        else if (p1->exp > p2->exp) {
            insert(&result, p1->coef, p1->exp);
            p1 = p1->next;
        }
        else {
            insert(&result, p2->coef, p2->exp);
            p2 = p2->next;
        }
    }

    while (p1 != NULL) {
        insert(&result, p1->coef, p1->exp);
        p1 = p1->next;
    }

    while (p2 != NULL) {
        insert(&result, p2->coef, p2->exp);
        p2 = p2->next;
    }

    return result;
}

/* Display polynomial */
void display(struct node *head) {
    while (head != NULL) {
        printf("%dx^%d", head->coef, head->exp);
        if (head->next != NULL)
            printf(" + ");
        head = head->next;
    }
    printf("\n");
}

/* Main Function */
int main() {
    struct node *poly1 = NULL;
    struct node *poly2 = NULL;
    struct node *result = NULL;
    int n, c, e, i;

    printf("Enter number of terms in Polynomial 1: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("Enter coefficient and exponent: ");
        scanf("%d %d", &c, &e);
        insert(&poly1, c, e);
    }
```

```
    printf("Enter number of terms in Polynomial 2: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("Enter coefficient and exponent: ");
        scanf("%d %d", &c, &e);
        insert(&poly2, c, e);
    }

    result = addPolynomials(poly1, poly2);

    printf("\nPolynomial 1: ");
    display(poly1);

    printf("Polynomial 2: ");
    display(poly2);

    printf("Sum: ");
    display(result);

    return 0;
}
```

Output

```
Enter number of terms in Polynomial 1: 2
Enter coefficient and exponent: 5 3
Enter coefficient and exponent: 4 1

Enter number of terms in Polynomial 2: 2
Enter coefficient and exponent: 3 3
Enter coefficient and exponent: 2 0

Polynomial 1: 5x^3 + 4x^1
Polynomial 2: 3x^3 + 2x^0
Sum: 8x^3 + 4x^1 + 2x^0
```

Circular Linked List

A **Circular Linked List** is a special type of linked list in which the nodes are connected in such a way that the last node links back to the first node, forming a closed loop. Unlike a linear linked list, there is no NULL pointer marking the end of the list. Instead, the structure is continuous and circular.

Conceptually, a circular linked list represents a repeating sequence of elements. Once the traversal reaches the last node, it automatically returns to the starting node. Because of this property, the list can be traversed indefinitely unless a specific stopping condition is applied.

Head = 1000

- **Fundamental Structure**

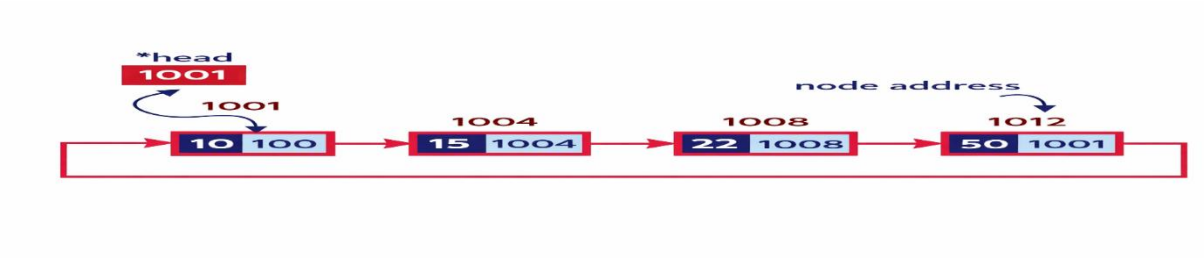
Each node in a circular linked list contains:

- **Data field** – Stores the value
- **Link field (next pointer)** – Stores the address of the next node

In a circular singly linked list:

- The last node's next pointer stores the address of the first node.
- There is no NULL pointer in the structure.

Representation of Circular Linked List



- The diagram represents a **circular singly linked list**.
- The **head pointer** stores the address of the first node.
- In the diagram, the head contains address **1001**, which is the starting node.
- Each node has **two parts**:
  - Data field (stores the value)
  - Link field (stores the address of the next node)
- The data values in the list are **10, 15, 22, and 50**.
- Each node points to the next node using its link (address).
- The nodes are connected in this order:
  10 → 15 → 22 → 50
- The last node does **not** contain NULL.
- The last node (50) points back to the first node (address 1001).
- Because the last node connects to the first node, the list forms a **circle**, so it is called a circular linked list.

Operations of Circular Linked List

Insertion:

Insertion is the operation of adding a new node into the circular singly linked list. A node can be inserted at the beginning, at the end, or at any specific position. While inserting, the links must be updated correctly so that the circular structure is maintained. If inserted at the end, the new node must point back to the head node.

Deletion:

Deletion is the process of removing a node from the list. The node to be deleted may be the first node, last node, or any intermediate node. After removing the node, the previous node's link must be adjusted so that the circular connection of the list remains intact.

Traversal:

Traversal means visiting each node of the list one by one. The process starts from the head node and continues until the head is reached again. Since the list is circular, there is no NULL at the end, so traversal stops when it returns to the starting node.

Searching:

Searching is used to find a particular element in the list. The list is traversed from the head node, comparing each node's data with the required value. The process continues until the element is found or the traversal reaches the head again.

Updating:

Updating is the operation of modifying the data stored in a node. First, the required node is located using the searching process. Once found, the existing data is replaced with the new value without disturbing the links between nodes.

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
   int data;
   struct Node* next;
};

struct Node* head = NULL;

// Insert at end
void insert(int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;

   if (head == NULL) {
      head = newNode;
      newNode->next = head;   // Point to itself (circular)
   } else {
      struct Node* temp = head;
      while (temp->next != head) {
         temp = temp->next;
      }
      temp->next = newNode;
      newNode->next = head;   // Maintain circular link
   }
}

// Traverse and display
void display() {
   if (head == NULL) {
      printf("List is empty\n");
      return;
   }

   struct Node* temp = head;
   do {
      printf("%d ", temp->data);
      temp = temp->next;
   } while (temp != head);
}
```

```
int main() {
    insert(10);
    insert(15);
    insert(22);
    insert(50);

    printf("Circular Linked List: ");
    display();

    return 0;
}
```

Output:

Circular Linked List: 10 15 22 50

Doubly Linked List

A **Doubly Linked List** is a type of linked list in which each node is connected to both its previous and next nodes. Unlike a singly linked list, every node contains two links: one pointing to the next node and another pointing to the previous node. The first node's previous pointer is **NULL**, and the last node's next pointer is **NULL**.

Conceptually, a doubly linked list allows traversal in **both forward and backward directions**, making it more flexible than a singly linked list.

◇ **Fundamental Structure**

Each node in a doubly linked list contains:

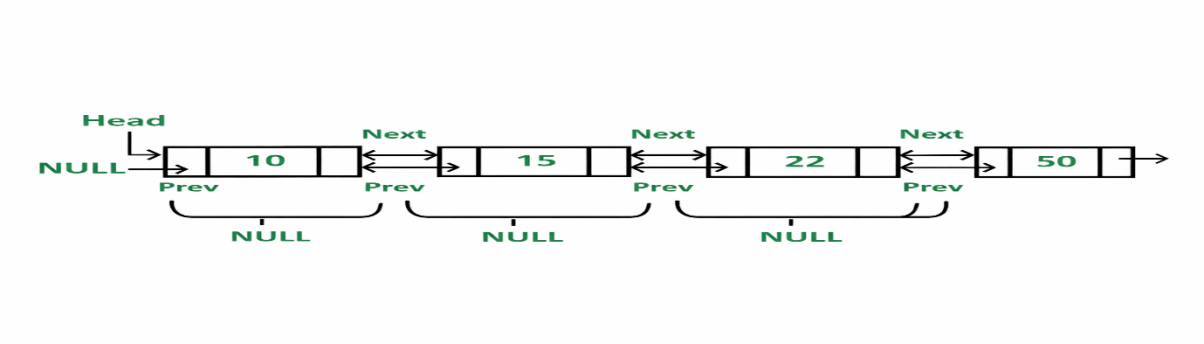
- **Data field** – Stores the value
- **Next pointer** – Stores the address of the next node
- **Previous pointer** – Stores the address of the previous node

In a doubly linked list:

- The first node's **prev = NULL**
- The last node's **next = NULL**
- Nodes are connected in both directions

◇ **Representation of Doubly Linked List**

- The list has a **head pointer** that stores the address of the first node.
- Each node has three parts:
    - Previous link
    - Data
    - Next link



- Example data values: **10, 15, 22, 50**
- Connection order:

NULL ← 10 ⇄ 15 ⇄ 22 ⇄ 50 → NULL

- Each node points forward to the next node.
- Each node also points backward to the previous node.
- The first node's previous is NULL.
- The last node's next is NULL.

Operations of Doubly Linked List

Insertion:

Insertion is the process of adding a new node into the doubly linked list. The node may be inserted at the beginning, at the end, or at any specified position. While inserting, both the **next** and **previous** pointers of the new node and its adjacent nodes must be updated correctly to maintain proper two-way connections.

Deletion:

Deletion refers to removing a node from the list. The node to be deleted may be the first node, last node, or any intermediate node. After deletion, the previous node's **next** pointer and the next node's **previous** pointer must be adjusted to preserve the continuity of the list.

Traversal:

Traversal means accessing each node of the list sequentially. In a doubly linked list, traversal can be performed in two directions: forward traversal using the **next** pointer and backward traversal using the **previous** pointer.

Searching:

Searching is the operation of locating a specific element in the list. The list is traversed node by node, either forward or backward, comparing each node's data with the required value until the element is found or the list ends.

Updating:

Updating involves changing the data stored in a particular node. First, the desired node is located through searching, and then its existing data is replaced with a new value without disturbing the links between nodes.

C Program for Doubly Linked List (Insertion at End & Display)

```
#include <stdio.h>
#include <stdlib.h>
```

```
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

struct Node* head = NULL;

// Insert at end
void insert(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (head == NULL) {
        newNode->prev = NULL;
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
}

// Display forward
void display() {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

int main() {
    insert(10);
    insert(15);
    insert(22);
    insert(50);

    printf("Doubly Linked List: ");
    display();
```

```
    return 0;
}
```

Output:

Doubly Linked List: 10 15 22 50

Advantages of Doubly Linked List

1. **Two-Way Traversal**
   A doubly linked list allows movement in both forward and backward directions because each node has both next and previous pointers.
2. **Easy Deletion**
   Deleting a node is easier compared to a singly linked list since we can directly access the previous node using the prev pointer.
3. **Efficient Insertion**
   Insertion before or after a given node can be done easily without needing to traverse the entire list.
4. **Better Navigation**
   It is useful in applications that require backtracking, such as browser history, undo-redo operations, and playlist navigation.
5. **No Need to Traverse from Head for Backward Move**
   Unlike singly linked lists, we do not need to start from the head to move backward.
6. **Flexible Structure**
   The list can grow or shrink dynamically during program execution without wasting memory.

Hashing

Hashing is a technique used to store and retrieve data efficiently by directly mapping a key to a specific location in memory. Instead of searching through all elements one by one, hashing uses a mathematical rule called a **hash function** to calculate where the data should be placed.

Hashing works like assigning roll numbers to lockers. If you know the roll number, you can directly open the correct locker without checking every locker. Similarly, when a key is given, the hash function converts it into an index, and the data is stored at that position in a **hash table**.

The main goal of hashing is to reduce the time required for searching. In most cases, data can be accessed in constant time (O(1)), making it much faster than linear search methods.

However, sometimes two different keys may generate the same index. This situation is called a **collision**. To manage collisions, special techniques are used so that data is stored properly without loss.

In simple terms, hashing is a smart method of storing data so that it can be found quickly using a calculated address instead of searching step by step.

Components of Hashing

Hashing is a technique used in data structures to store and retrieve data efficiently. It is based on three fundamental components: the **key**, the **hash function**, and the **hash table**. These components work together to ensure fast access to data.

Key

A **key** is a unique identifier used to represent a data item. It is the input provided to the hashing process. The key may be a number, string, or any other type of value that uniquely identifies a record. In practical applications, examples of keys include roll numbers, employee IDs, usernames, or account numbers. The key is not stored directly based on its original value; instead, it is processed by a hash function to determine its storage location.

Hash Function

A **hash function** is a mathematical function that transforms the given key into a fixed-size numerical value known as a hash value or index. This index corresponds to a specific position in the hash table. The primary purpose of the hash function is to compute the storage address of the data quickly. An effective hash function should be simple to compute, distribute keys uniformly across the table, and minimize collisions. The performance of the hashing technique largely depends on the quality of the hash function.

Hash Table

A **hash table** is a data structure, typically implemented as an array, used to store data elements. The index generated by the hash function determines the position in the hash table where the data will be stored. Each location in the table is called a slot or bucket. The size of the hash table plays an important role in determining efficiency. If the table becomes too full, the chances of collisions increase, which may reduce performance. Therefore, proper selection of table size and collision handling methods is important.

Static Hashing

Static Hashing is a hashing technique in which the number of buckets (or memory locations) in the hash table is fixed at the time of its creation and remains constant throughout the execution of the program. The size of the hash table does not change dynamically, regardless of the number of records inserted or deleted.

In this method, a hash function is applied to a key to determine the bucket address where the corresponding record will be stored. Since the number of buckets is predetermined, the mapping between keys and storage locations remains fixed.

◇ Basic Concept

Let:

- m= number of buckets (fixed)
- $h(key)$= hash function

The hash function computes an index between 0and $m-1$. Each index corresponds to a bucket in the hash table.

If two different keys produce the same index, a **collision** occurs. Because the table size cannot expand, collision resolution techniques such as:

- Separate chaining
- Linear probing

- Quadratic probing

must be applied.

◇ Working of Static Hashing

1. A fixed number of buckets is allocated in memory.
2. A hash function is selected.
3. For each key, the hash function generates a bucket address.
4. The record is stored in the corresponding bucket.
5. If the bucket is already occupied, collision resolution is applied.
6. The structure remains unchanged in size even if more records are added.

◇ Example

Assume:

- Number of buckets $m = 7$
- Hash function:

$$h(key) = key \bmod 7$$

Insert keys: **10, 21, 32, 43**

$$h(10) = 10 \bmod 7 = 3$$
$$h(21) = 21 \bmod 7 = 0$$
$$h(32) = 32 \bmod 7 = 4$$
$$h(43) = 43 \bmod 7 = 1$$

Now suppose we insert **17**:

$$h(17) = 17 \bmod 7 = 3$$

Index 3 already contains 10 → collision occurs.
Since the table size is fixed, we must handle this collision without increasing the number of buckets.

◇ Characteristics of Static Hashing

- Fixed number of buckets.
- Hash function remains unchanged.
- Memory allocation is done once at the beginning.
- Collision handling is necessary when multiple keys map to the same bucket.
- Performance depends on load factor.

◇ Advantages

1. Simple to design and implement.
2. Requires less overhead compared to dynamic hashing.
3. Suitable when the number of records is known in advance.
4. Efficient for small and stable datasets.

◇ **Disadvantages**

1. Cannot grow dynamically when data increases.
2. High load factor leads to more collisions.
3. Rehashing requires creating a new larger table and reinserting all records.
4. Memory may be wasted if the table is underutilized.

Hash Function Methods

A **hash function** is a mathematical technique used to transform a given key into a fixed-size numerical value called a hash value or index. Different methods are used to design hash functions in order to achieve uniform distribution of keys and minimize collisions.

Hash Function Methods –

The commonly used hash function methods are:

1. **Division Method**
2. **Multiplication Method**
3. **Mid-Square Method**
4. **Folding Method**
5. **Digit Extraction Method**

Division Method

The **Division Method** is one of the simplest and most commonly used techniques for constructing a hash function. In this method, the key value is divided by the size of the hash table, and the remainder obtained is used as the index for storing the data.

Mathematically, it is represented as:

$$h(key) = key \bmod m$$

where:

- *key* is the data item to be stored,
- *m* is the size of the hash table,
- *h(key)* is the index generated.

Conceptually, this method works by grouping keys according to their remainder values. Since the remainder of a division operation always lies between 0 and $m - 1$, it naturally fits within the valid index range of the hash table. This makes the division method simple, efficient, and easy to implement.

Example

Assume the hash table size $m = 10$.

Let us insert the following keys: **101, 112, 125, 138**

1. $h(101) = 101 \bmod 10 = 1$
   → Store 101 at index 1
2. $h(112) = 112 \bmod 10 = 2$
   → Store 112 at index 2

3. $h(125) = 125 \text{mod } 10 = 5$
   → Store 125 at index 5
4. $h(138) = 138 \text{mod } 10 = 8$
   → Store 138 at index 8

Thus, the elements are placed directly into the table based on the remainder values.
If we insert another key, **111**:

$$h(111) = 111 \text{mod } 10 = 1$$

Index 1 is already occupied by 101. This situation is called a **collision**, and it must be handled using collision resolution techniques such as chaining or probing.

Multiplication Method

The **Multiplication Method** is a hashing technique in which the key is multiplied by a constant fractional value, and the fractional part of the result is used to determine the index of the hash table. Unlike the division method, this technique does not depend strictly on choosing a prime table size and generally provides better distribution of keys.
Mathematically, it is represented as:

$$h(key) = \lfloor m(key \times A \text{mod } 1) \rfloor$$

where:

- *key* is the data item to be stored,
- *m* is the size of the hash table,
- *A* is a constant such that $0 < A < 1$,
- *mod 1* means taking only the fractional part,
- *h(key)* is the index generated.

Conceptually, this method works by first scaling the key using a constant value and then extracting the fractional part to ensure uniform distribution. The fractional part is then multiplied by the table size to obtain a valid index between 0and $m - 1$. This approach reduces clustering and spreads keys more evenly across the table.

Example
Assume the hash table size $m = 10$
Let the constant $A = 0.618$
Let us insert the following keys: **101, 112, 125, 138**

1. For key 101

$$101 \times 0.618 = 62.418$$

Fractional part = 0.418

$$0.418 \times 10 = 4.18$$

Taking the integer part → **4**
→ Store 101 at index 4

2. For key 112

$$112 \times 0.618 = 69.216$$

Fractional part = 0.216

$$0.216 \times 10 = 2.16$$

Integer part → **2**
→ Store 112 at index 2

3. For key 125

$$125 \times 0.618 = 77.25$$

Fractional part = 0.25

$$0.25 \times 10 = 2.5$$

Integer part → **2**
→ Store 125 at index 2
Since index 2 is already occupied by 112, a **collision** occurs.

4. For key 138

$$138 \times 0.618 = 85.284$$

Fractional part = 0.284

$$0.284 \times 10 = 2.84$$

Integer part → **2**
→ Store 138 at index 2

Mid-Square Method

The **Mid-Square Method** is a hashing technique in which the key is first squared, and then the middle digits of the squared value are selected to form the hash index. This method reduces the effect of patterns in the original key and often provides better distribution of values in the hash table.

Mathematically, the method can be described as:

1. Compute key^2
2. Extract the middle digits of the result
3. Use those digits as the index

Conceptually, squaring the key spreads the influence of all digits of the original number. By selecting the middle portion, the method avoids bias caused by leading or trailing digits. This helps in distributing keys more uniformly across the table.

Example

Assume the hash table size allows 2-digit indices.

Example 1

Key = 12

Step 1:

$$12^2 = 144$$

Step 2:

Middle digit(s) = 4

So, index = 4

Folding Method

The **Folding Method** is a hashing technique in which the key is divided into smaller parts (groups of digits), and these parts are combined—usually by addition—to produce the hash index. This method is particularly useful when the key is a large number.

Conceptually, instead of using the entire key directly, the folding method breaks the key into equal-sized segments and then folds them together to generate a smaller value that fits within the hash table size. This helps in distributing keys more uniformly.

Working Principle

1. Divide the key into equal parts.
2. Add (or sometimes reverse and add) the parts together.
3. If necessary, apply modulo operation to fit the result within the table size.

Example

Assume the hash table size $m = 100$

Let the key = **12345678**

Step 1: Divide into Parts

Split into 2-digit groups:

12 | 34 | 56 | 78

Step 2: Add the Parts

12 + 34 + 56 + 78 = 180

Step 3: Apply Modulo (if required)

$$180 \bmod 100 = 80$$

So, the index = **80**

Types of Folding

1 Shift Folding

- Simply divide and add the parts directly.

2 Boundary Folding

- Reverse alternate parts before adding.

Example (Boundary Folding):

12 | 34 | 56 | 78

Reverse 34 → 43

Reverse 78 → 87

Now add:

12 + 43 + 56 + 87 = 198

If table size = 100

$$198 \bmod 100 = 98$$

Index = **98**

Digit Extraction Method

The **Digit Extraction Method** is a hashing technique in which specific digits are selected from the given key to form the hash index. Instead of using the entire key or performing complex calculations, this method extracts predetermined digit positions and combines them to generate the address in the hash table.

In this method, certain digits of the key are chosen based on their positions (for example, 2nd and 4th digits). These selected digits are then used directly as the index or further processed to fit within the table size.

The idea behind this method is that some digits of a key may vary more uniformly than others. By carefully selecting those digits, better distribution of keys can be achieved.

Working Principle

1. Identify which digit positions will be used.
2. Extract those digits from the key.
3. Combine the extracted digits to form the hash index.
4. If necessary, apply modulo operation to fit within table size.

Example

Assume the hash table size = 100

Key = **987654**

Let us select the **3rd and 5th digits from the right**.

Key = 9 8 7 6 5 4

Digits from right:

4 (1st), 5 (2nd), 6 (3rd), 7 (4th), 8 (5th), 9 (6th)

Selected digits:

3rd digit = 6

5th digit = 8

Combine them → **68**

So, index = **68**

Limitations / Disadvantages of Hash Functions

Although hash functions are widely used for fast data storage and retrieval, they have certain limitations. These disadvantages mainly arise due to collisions and design constraints.

Collision Problem

The most common limitation of hashing is **collision**. Different keys may generate the same hash index. When collisions occur frequently, performance decreases and additional collision-handling techniques are required.

Performance Depends on Hash Function

The efficiency of hashing depends heavily on the quality of the hash function.

- A poorly designed hash function may cause clustering.
- Uneven distribution of keys increases search time.

Fixed Table Size

Hash tables usually have a fixed size.

- If the table becomes full, performance decreases.
- Rehashing (resizing and reinserting elements) is required, which is time-consuming.

Wastage of Memory

If the table size is large and the number of stored elements is small, memory may be underutilized. Empty slots lead to space wastage.

Collision in Hashing

A **collision in hashing** occurs when two or more different keys generate the same hash index in a hash table. Since a hash table has a fixed number of slots and the number of possible keys is usually much larger, collisions are inevitable in any practical hashing system.

In hashing, a **hash function** converts a key into an index of a hash table. Ideally, each key should map to a unique index. However, because the table size is limited, different keys may produce the same index value. This situation is called a **collision**.

For example, consider the hash function:

$$h(key) = key \text{mod } 10$$

If we insert the keys 27 and 37:

$$h(27) = 27 \text{mod } 10 = 7$$
$$h(37) = 37 \text{mod } 10 = 7$$

Both keys map to index 7. Since only one element can occupy that position (in open addressing), a collision occurs.

Causes of Collision

Collisions in hashing occur when two or more distinct keys are mapped to the same index in a hash table. This situation arises due to the following reasons:

Limited Table Size

A hash table contains a finite number of slots, whereas the number of possible keys may be very large. Since the key space is much larger than the table space, it is mathematically unavoidable that multiple keys will map to the same index. This limitation of storage capacity is one of the primary causes of collisions.

Poor Hash Function Design

The efficiency of hashing depends greatly on the quality of the hash function. If the hash function does not distribute keys uniformly across the table, certain slots may receive a large number of keys while others remain empty. This uneven distribution leads to clustering and increases the likelihood of collisions.

High Load Factor

The load factor is defined as the ratio of the number of stored elements to the number of available slots in the hash table. When the load factor increases, more elements compete for limited positions, thereby increasing the probability of collisions. As the table becomes more filled, collisions become more frequent.

Similar Key Patterns

When keys follow similar patterns or share common digits, and the hash function does not sufficiently randomize them, they may produce identical or closely related hash values. This results in multiple keys being assigned to the same or neighboring indices, causing collisions.

Collision Handling Techniques/Introduction to Overflow Handling in Hashing

In hashing, data is stored in a hash table using a hash function that maps keys to specific locations called buckets. However, since the number of possible keys is usually much larger than the number of available buckets, multiple keys may map to the same location. When a bucket becomes full and cannot store additional records, this situation is called **overflow**. Overflow mainly occurs due to **collisions**, where different keys produce the same hash value. Because the hash table has limited space, it is not always possible to store all records in their original computed locations. Therefore, special techniques are required to handle overflow effectively.

Overflow handling in hashing refers to the methods used to manage and store extra records when the assigned bucket is already occupied. The primary goal of overflow handling is to ensure that:

- No data is lost,
- Insertions can still be performed,
- Search operations remain efficient,
- Overall performance of the hash table is maintained.

Proper overflow handling becomes especially important when the **load factor** (ratio of number of records to number of buckets) increases. As the load factor rises, the probability of overflow also increases.

Common overflow handling techniques include:

- **Separate Chaining**
- **Open Addressing**
  Linear Probing,
  Quadratic Probing,
  Double Hashing

Separate Chaining

Separate Chaining is a collision resolution technique used in hashing to handle situations where multiple keys are mapped to the same index in a hash table. Instead of trying to find another empty slot within the table, this method stores all elements that hash to the same index in a linked list (or chain) associated with that index.

In separate chaining, each slot of the hash table contains a pointer to a linked list. When a collision occurs, the new element is simply added to the linked list at that index. Thus, multiple elements can occupy the same hash index without overwriting each other.

This method ensures that all keys are stored properly, even when many collisions occur. The hash table itself stores references to chains, while the actual records are stored in linked list nodes.

Working

1. Compute the hash value using the hash function.
2. If the slot is empty, insert the element directly.
3. If the slot is already occupied, add the element to the linked list at that index.
4. During search, traverse the linked list at the computed index to find the required key.

Example

Suppose the hash function is:

$$h(key) = key \bmod 10$$

`Separate Chaining Example`

Given hash function:

$$h(key) = key \bmod 10$$

Insert the keys: **25, 35, 45, 52, 40**

Step-by-Step Calculation

1 **25**

$$h(25) = 25 \bmod 10 = 5$$

→ Store at index **5**

2 **35**

$$h(35) = 35 \bmod 10 = 5$$

→ Collision at index 5

→ Add to chain at index 5

3 **45**

$$h(45) = 45 \bmod 10 = 5$$

→ Collision at index 5

→ Add to chain at index 5

4 **52**

$$h(52) = 52 \bmod 10 = 2$$

→ Store at index **2**

5 **40**

$$h(40) = 40 \bmod 10 = 0$$

→ Store at index **0**

Final Hash Table (Using Separate Chaining)

Index	Elements (Chain)
0	40
2	52

5	25 → 35 → 45

Advantages

- Simple to implement.
- Table never becomes completely full (as long as memory is available).
- Performance degrades gradually as load factor increases.

Disadvantages

- Requires extra memory for linked lists.
- Search time increases if chains become long.
- Cache performance may be lower due to pointer usage.

Open Addressing

Open Addressing is a collision resolution technique used in hashing. In this method, when a collision occurs, the system searches for another empty slot within the same hash table instead of using a linked list. All elements are stored directly in the hash table itself.
When the computed index is already occupied, a probing sequence is followed to find the next available position. The three common probing methods are:

- Linear Probing
- Quadratic Probing
- Double Hashing

Linear Probing

Linear Probing is a collision resolution technique used in open addressing hashing. In this method, when a collision occurs (i.e., when two keys hash to the same index), the system searches sequentially for the next available empty slot in the hash table.

In linear probing, all elements are stored directly in the hash table. When the computed hash index is already occupied, the algorithm checks the next slot in a linear manner (one by one) until an empty position is found.
The probing sequence follows a fixed linear pattern, which means the interval between successive probes is constant (usually 1). Because of this simple sequential search, the method is easy to implement.
The general probing formula is:

$$h_i(key) = (h(key) + i) \text{mod } m$$

where:

- $h(key)$is the original hash function,
- $i = 0,1,2,3, \ldots$
- $m$is the size of the hash table.

Working with Example

Assume:
Hash table size $m = 10$
Hash function:

$$h(key) = key \bmod 10$$

Insert the keys: **25, 35, 45**

1 Insert 25

$$h(25) = 25 \bmod 10 = 5$$

→ Store 25 at index 5

2 Insert 35

$$h(35) = 35 \bmod 10 = 5$$

Index 5 is already occupied → Collision occurs
Apply linear probing:

$$(5 + 1) \bmod 10 = 6$$

Index 6 is empty → Store 35 at index 6

3 Insert 45

$$h(45) = 45 \bmod 10 = 5$$

Index 5 is occupied
Check index 6 → occupied
Check next slot:

$$(5 + 2) \bmod 10 = 7$$

Index 7 is empty → Store 45 at index 7

Final Hash Table
Index 5 → 25
Index 6 → 35
Index 7 → 45

Advantages

- Simple and easy to implement.
- No extra memory required (all elements stored in table).

Disadvantages

- Causes **primary clustering**, where consecutive slots become filled.
- Performance decreases as load factor increases.

- Worst-case search time may become $O(n)$

Quadratic Probing

Quadratic Probing is a collision resolution technique used in open addressing hashing. It is an improvement over linear probing. Instead of checking the next slot sequentially, quadratic probing uses a quadratic function to determine the next position. This helps in reducing **primary clustering**.

In quadratic probing, when a collision occurs at a given index, the next position is calculated using square increments (1^2, 2^2, 3^2, …). This spreads the elements more evenly across the table compared to linear probing.
The probing formula is:

$$h_i(key) = (h(key) + i^2) \text{mod } m$$

where:

- $h(key)$is the original hash value,
- $i = 0,1,2,3, \ldots$
- $m$is the size of the hash table.

Since the step size increases quadratically, elements do not cluster in consecutive slots. Thus, primary clustering is reduced.

Working with Example

Assume:
Hash table size $m = 10$
Hash function:

$$h(key) = key \text{mod } 10$$

Insert keys: **25, 35, 45**

1 Insert 25

$$h(25) = 25 \text{mod } 10 = 5$$

→ Store 25 at index 5

2 Insert 35

$$h(35) = 35 \text{mod } 10 = 5$$

Collision at index 5
Apply quadratic probing:

$$(5 + 1^2) \text{mod } 10 = 6$$

→ Store 35 at index 6

3 Insert 45

$$h(45) = 45 \bmod 10 = 5$$

Collision at index 5
Check:

$$(5 + 1^2) \bmod 10 = 6$$

Index 6 is occupied
Next:

$$(5 + 2^2) \bmod 10 = 9$$

→ Store 45 at index 9

Final Hash Table
Index 5 → 25
Index 6 → 35
Index 9 → 45

Why It Reduces Primary Clustering
In linear probing, consecutive slots (5, 6, 7…) get filled, forming clusters.
In quadratic probing, the jumps are 1, 4, 9… instead of 1, 2, 3…, which spreads elements more widely and avoids large continuous blocks of occupied slots.

Advantages

- Reduces primary clustering.
- No extra memory required.

Disadvantages

- May still suffer from **secondary clustering**.
- More complex than linear probing.
- May not examine all slots unless table size is chosen properly.

Double Hashing
Double Hashing is a collision resolution technique used in open addressing hashing. It uses **two different hash functions** to calculate the probe sequence. The first hash function determines the initial index, and the second hash function determines the step size for probing when a collision occurs.
Double hashing is considered one of the most effective open addressing methods because it significantly reduces both **primary clustering** and **secondary clustering**.

In double hashing, when a collision occurs at the index given by the first hash function, a second hash function is used to compute the interval (step size) for the next probe.
The probing formula is:

$$h_i(key) = (h_1(key) + i \cdot h_2(key)) \bmod m$$

where:

- $h_1(key)$= primary hash function
- $h_2(key)$= secondary hash function
- $i = 0,1,2,3, \ldots$
- m= size of hash table

The second hash function must never return zero, otherwise the probing sequence will not move forward.

Working with Example

Assume:

Table size $m = 10$

Primary hash function:

$$h_1(key) = key \bmod 10$$

Secondary hash function:

$$h_2(key) = 7 - (key \bmod 7)$$

Insert keys: **25, 35, 45**

1 Insert 25

$$h_1(25) = 25 \bmod 10 = 5$$

Index 5 is empty → Store 25 at index 5

2 Insert 35

$$h_1(35) = 35 \bmod 10 = 5$$

Collision at index 5

Now compute second hash:

$$h_2(35) = 7 - (35 \bmod 7)$$
$$35 \bmod 7 = 0$$
$$h_2(35) = 7 - 0 = 7$$

Now probe:

$$(5 + 1 \cdot 7) \bmod 10 = 12 \bmod 10 = 2$$

Index 2 is empty → Store 35 at index 2

3 Insert 45

$$h_1(45) = 45 \bmod 10 = 5$$

Collision at index 5

Compute second hash:

$$h_2(45) = 7 - (45 \bmod 7)$$
$$45 \bmod 7 = 3$$
$$h_2(45) = 7 - 3 = 4$$

Probe:

$$(5 + 1 \cdot 4) \bmod 10 = 9$$

Index 9 is empty → Store 45 at index 9

Final Hash Table
Index 2 → 35
Index 5 → 25
Index 9 → 45

Advantages

- Reduces primary clustering.
- Reduces secondary clustering.
- Provides better key distribution than linear and quadratic probing.

Disadvantages

- More complex to implement.
- Requires careful choice of second hash function.

UNIT – III : Trees & Graphs

UNIT – IV : Sorting & Searching Techniques

Part – A : Searching Techniques

1. Sequential (Linear) Search
2. Binary Search

Part – B : Sorting Techniques

1. Insertion Sort
2. Quick Sort
3. Merge Sort
4. Heap Sort
5. Shell Sort

Part – C : Sorting Analysis

1. Best Computing Time for Sorting
2. Time Complexity of Sorting Algorithms

Part – D : Advanced Sorting

1. Sorting on Several Keys
2. List Sorts
3. Table Sorts

Part – E : Summary of Sorting

1. Summary of Internal Sorting
2. Comparison of Sorting Algorithms

**UNIT – I

Algorithms, Arrays & Stacks**

LONG ANSWER QUESTIONS (10–15 Marks)

1. Explain **algorithm specification** and **performance analysis of algorithms** with suitable examples.
2. Explain **time and space complexity**. Discuss **asymptotic notations** (Big-O, Big-Ω, Big-Θ).
3. Explain **recursive algorithms** with an example.
4. Explain **Array ADT**. Describe **polynomial representation** using arrays.
5. Explain **sparse matrix representation** using arrays.
6. Explain **Stack ADT** and its implementation using arrays.

7. Explain **evaluation of postfix expression** using stack.
8. Explain **infix to postfix conversion** using stack.

SHORT ANSWER QUESTIONS (2–5 Marks)

1. Define **algorithm**.
2. What is **time complexity**?
3. What is **space complexity**?
4. Define **Big-O notation**.
5. What is **Array ADT**?
6. What is a **sparse matrix**?
7. Define **Stack ADT**.
8. What is **postfix expression**?
9. What is **well-formed parenthesis**?
10. List **applications of stacks**.

**UNIT – II

Queues, Linked Lists & Hashing**

LONG ANSWER QUESTIONS (10–15 Marks)

1. Explain **Queue ADT** and its operations.
2. Explain **circular queue** with advantages.
3. Explain **singly linked list** and its operations.
4. Explain **linked stack and linked queue**.
5. Explain **polynomial representation using linked lists**.
6. Explain **operations on circular linked lists**.
7. Explain **doubly linked list** with advantages.
8. Explain **hashing**. Describe **hash functions and overflow handling techniques**.

SHORT ANSWER QUESTIONS (2–5 Marks)

1. Define **queue**.
2. What is a **circular queue**?
3. What is a **linked list**?

4. Define **singly linked list**.
5. What is a **doubly linked list**?
6. What is **hashing**?
7. Define **hash table**.
8. What is **overflow handling**?
9. Define **equivalence class**.
10. List **applications of linked lists**.

**UNIT – III

Trees & Graphs**

LONG ANSWER QUESTIONS (10–15 Marks)

1. Explain **binary trees** and their **traversal methods**.
2. Explain **Binary Search Tree (BST)**. Describe **search, insertion, and deletion** operations.
3. Explain **heaps** and heap operations.
4. Explain **AVL trees**. Describe **insertion and balancing**.
5. Explain **Graph ADT** and its representations.
6. Explain **DFS and BFS algorithms**.
7. Explain **Prim's algorithm** for minimum cost spanning tree.
8. Explain **Kruskal's algorithm** for minimum cost spanning tree.

SHORT ANSWER QUESTIONS (2–5 Marks)

1. Define **tree**.
2. What is a **binary tree**?
3. List **binary tree traversals**.
4. Define **BST**.
5. What is an **AVL tree**?
6. Define **graph**.
7. What is **DFS**?
8. What is **BFS**?
9. What is a **spanning tree**?
10. Define **heap**.

**UNIT – IV

Sorting & Searching**

LONG ANSWER QUESTIONS (10–15 Marks)

1. Explain **linear search and binary search** algorithms with complexity.
2. Explain **insertion sort** with example.
3. Explain **quick sort** algorithm and its performance.
4. Explain **merge sort** with time complexity.
5. Explain **heap sort** algorithm.
6. Explain **shell sort** and its advantages.
7. Compare **different internal sorting techniques**.
8. Explain **sorting on several keys**.

SHORT ANSWER QUESTIONS (2–5 Marks)

1. What is **searching**?
2. Define **linear search**.
3. Define **binary search**.
4. What is **sorting**?
5. What is **quick sort**?
6. Define **merge sort**.
7. What is **heap sort**?
8. What is **best case time complexity**?
9. What is **internal sorting**?
10. List **applications of sorting**.

UNIT – I : Algorithms, Arrays & Stacks

1. The time complexity of an algorithm mainly depends on:
   A) Machine used
   B) Programming language
   C) Number of basic operations
   D) Size of memory
   Answer: C
2. Which asymptotic notation gives the **upper bound** of an algorithm?
   A) Big-Ω
   B) Big-Θ
   C) Big-O

D) Little-o
Answer: C

3. A recursive algorithm must have:
   A) Loop
   B) Base condition
   C) Array
   D) Stack only
   Answer: B
4. A sparse matrix is one in which:
   A) All elements are zero
   B) Most elements are zero
   C) All elements are non-zero
   D) Rows = Columns
   Answer: B
5. In postfix expression evaluation, operands are:
   A) Placed after operators
   B) Placed before operators
   C) Placed on both sides
   D) Not used
   Answer: B

UNIT – II : Queues, Linked Lists & Hashing

6. A circular queue is preferred over a linear queue because it:
   A) Uses less memory
   B) Avoids wastage of space
   C) Is easier to implement
   D) Is faster always
   Answer: B
7. In a singly linked list, each node contains:
   A) Data only
   B) Data and two pointers
   C) Data and one pointer
   D) Only pointers
   Answer: C
8. Which linked list allows traversal in both directions?
   A) Singly linked list
   B) Circular linked list
   C) Doubly linked list
   D) Linear list
   Answer: C

9. Hashing is mainly used to:
   A) Sort data
   B) Search data efficiently
   C) Compress data
   D) Encrypt data
   Answer: B
10. Collision in hashing occurs when:
   A) Table is full
   B) Same key is inserted twice
   C) Two keys map to same location
   D) Hash function fails
   Answer: C

UNIT – III : Trees & Graphs

11. A binary tree in which left subtree values are smaller and right subtree values are larger is called:
   A) Heap
   B) AVL Tree
   C) Binary Search Tree
   D) Complete Binary Tree
   Answer: C
12. Inorder traversal of a Binary Search Tree gives:
   A) Random order
   B) Reverse sorted order
   C) Sorted order
   D) Level order
   Answer: C
13. An AVL tree is a:
   A) Complete binary tree
   B) Balanced binary search tree
   C) Full binary tree
   D) Heap
   Answer: B
14. Breadth First Search (BFS) uses which data structure?
   A) Stack
   B) Queue
   C) Tree
   D) Array
   Answer: B

15. A minimum cost spanning tree of a graph has:
    A) Maximum edges
    B) Minimum vertices
    C) All vertices and minimum total edge cost
    D) Only minimum edges
    Answer: C

UNIT – IV : Sorting & Searching

16. Binary search is efficient because it:
    A) Searches sequentially
    B) Divides the search space
    C) Uses recursion only
    D) Uses hashing
    Answer: B
17. Which sorting algorithm works on the **divide and conquer** principle?
    A) Insertion sort
    B) Selection sort
    C) Quick sort
    D) Bubble sort
    Answer: C
18. Merge sort is preferred for:
    A) Small datasets
    B) Linked lists
    C) Sorted arrays
    D) Internal sorting only
    Answer: B
19. Heap sort is based on:
    A) Binary Search Tree
    B) AVL Tree
    C) Heap data structure
    D) Graph
    Answer: C
20. The best-case time complexity of insertion sort occurs when the list is:
    A) Reverse sorted
    B) Random
    C) Already sorted
    D) Unsorted
    Answer: C