# Basic Data Structures using C++ (23CSH-103)

## ALL UNITS - NOTES & QUESTIONS

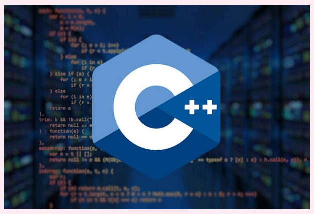
Compiled by : **Subhayu**

Contents : (Click on the link to skip to that particular unit)

Unit 1……………………………………………………………………………………………………………………………….
Unit 2……………………………………………………………………………………………………………………………….
Unit 3……………………………………………………………………………………………………………………………….

# UNIT-1: Fundamentals of C++

Contact Hours: 10



## 1. Fundamentals of C++

### Features of Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes. The key features of OOP are:

- **Encapsulation**: Bundling the data (attributes) and methods (functions) that operate on the data into a single unit known as a class.

- **Abstraction**: Hiding the complex implementation details and exposing only the necessary interface to the user.

- **Inheritance**: Mechanism by which one class can inherit properties and behaviors (methods) from another class, promoting code reusability.

- **Polymorphism**: Ability of a function or method to behave differently based on the object it is acting upon.

### Difference Between Object-Oriented and Procedure-Oriented Programming

- Object-Oriented Programming (OOP):

    - Based on the concept of "objects" which are instances of classes.

    - Focuses on data and methods that operate on that data.
    - Examples: C++, Java, Python.
- **Procedure-Oriented Programming** (POP):
    - Focuses on procedures or routines (functions) that operate on data.
    - Data and functions are separate, and the main focus is on the sequence of tasks.
    - Examples: C, Pascal.

**Example**: In C++, we define a class Car with properties and methods, while in C, we would write functions to perform operations on car-related data.

```
// OOP Example (C++ class)

class Car {

public:

   string model;

   int year;

   void drive() {

      cout << "Driving the car!";

   }

};
```

In a Procedure-Oriented language like C, you'd write a function that manipulates data directly, without encapsulating it in objects.

```
// POP Example (C)
struct Car {
    char model[50];
    int year;
};


void drive(struct Car c) {
    printf("Driving the car!");
}
```

**Input and Output Streams (Cin, Cout)**

- **Cout**: The cout stream is used to display output in C++.

- **Cin**: The cin stream is used to take input from the user.

**Example**:

```
#include <iostream>
using namespace std;


int main() {
    int age;
    cout << "Enter your age: ";
    cin >> age;
    cout << "Your age is: " << age;
    return 0;
}
```

## Introduction to Namespace

A **namespace** in C++ is used to avoid name conflicts by grouping identifiers (like functions, classes, variables) under a unique name.

```
#include <iostream>
using namespace std;

namespace MyNamespace {
    void greet() {
        cout << "Hello from MyNamespace!";
    }
}

int main() {
    MyNamespace::greet();
    return 0;
}
```

In the above example, the greet() function is part of the MyNamespace namespace, and we call it using the scope resolution operator ::.

---

## 2. Classes and Objects

### Specifying A Class

A class in C++ is a blueprint for creating objects. It defines the properties and behaviors of the object.

```
class Car {

public:

    string model;

    int year;

    void drive() {

        cout << "The car is driving";

    }

};
```

### Creating Objects

An object is an instance of a class. You can create an object using the class name.

```
Car myCar; // Creating an object of class Car

myCar.model = "Toyota";

myCar.year = 2020;

myCar.drive();
```

### Accessing Class Members

Class members (variables and methods) can be accessed using the object name.

```
cout << myCar.model; // Accessing the model of the car
myCar.drive(); // Calling the drive method
```

## Defining A Member Function Inside and Outside Class

- **Inside the class**: Functions can be defined inside the class definition.

- **Outside the class**: Functions can be defined outside the class definition using the scope resolution operator ::.

**Example**:

```
class Car {
public:
   string model;
   void drive(); // Declaration of function outside the class
};

void Car::drive() { // Definition of function outside the class
   cout << "The car is driving";
}
```

## Access Specifiers

C++ provides three access specifiers for class members:

- **Public**: Members are accessible from anywhere.
- **Private**: Members are only accessible within the class.
- **Protected**: Members are accessible within the class and derived classes.

```
class Car {
public:
   string model;
private:
   int year;
};
```

**Inline Function**

An inline function is defined using the keyword inline. It suggests to the compiler to replace the function call with the function's body.

```
class Car {
public:
   inline void drive() {
      cout << "The car is driving";
   }
};
```

**Constructor and Destructor**

- **Constructor**: A special function that is called when an object is created. It is used to initialize object data.

```
class Car {
public:
    string model;
    Car(string m) { // Constructor
        model = m;
    }
};
Car myCar("Toyota"); // Constructor is called here
```

- **Destructor**: A special function that is called when an object is destroyed. It is used to release resources.

```
class Car {
public:
    ~Car() { // Destructor
        cout << "Car object is destroyed!";
    }
};
```

---

## 3. Inheritance

## Concept of Inheritance

Inheritance is the mechanism by which one class can inherit properties and behaviors from another class. This allows code reuse and organization.

- **Base class**: The class whose properties and methods are inherited.

- **Derived class**: The class that inherits the properties and methods of the base class.

**Example**:

```
class Vehicle {
public:
    void start() {
        cout << "Vehicle is starting";
    }
};


class Car : public Vehicle {
public:
    void drive() {
        cout << "Car is driving";
    }
};
```

## Modes of Inheritance

C++ supports several modes of inheritance:

- **Single Inheritance**: A derived class inherits from a single base class.
- **Multiple Inheritance**: A derived class inherits from more than one base class.
- **Multilevel Inheritance**: A derived class inherits from another derived class.
- **Hierarchical Inheritance**: Multiple derived classes inherit from the same base class.
- **Hybrid Inheritance**: A combination of two or more types of inheritance.

**Examples**:

- **Single Inheritance**:

```
class Car : public Vehicle {};
```

- **Multiple Inheritance**:

```
class Engine {};
class Car : public Vehicle, public Engine {};
```

**Types of Inheritance**

- **Public Inheritance**: Inherited members remain accessible in the derived class as they are in the base class.
- **Protected Inheritance**: Inherited members are accessible within the derived class, but not outside it.
- **Private Inheritance**: Inherited members are not accessible outside the derived class.

**Example**:

```
class Car : public Vehicle { // Public inheritance
  // Members of Vehicle are public in Car
};
```

## Question Bank for Unit 1 : Fundamentals of C++

### 2-Mark Questions (Revised):

1. Define Object-Oriented Programming (OOP) and explain how encapsulation is implemented in C++.

2. Explain the difference between the cin and cout streams in C++, including their purpose and usage with examples.

3. What are the advantages of using a namespace in C++? Illustrate with an example of a potential naming conflict and how a namespace resolves it.

4. What are the different access specifiers available in C++? Provide a brief example demonstrating each access specifier.

5. What is the role of a constructor in C++? Differentiate between a default constructor and a parameterized constructor with an example.

---

### 5-Mark Questions (Revised):

1. Discuss the features of Object-Oriented Programming (OOP) and explain their significance in C++ programming. Provide examples of how encapsulation, inheritance, and polymorphism are implemented in C++.

2. Create a C++ class Student that contains the attributes name, age, and marks. Write a constructor to initialize these values and a method to display the student details. Discuss how the constructor and destructor work in this context.

3. What is inheritance in C++? Explain the concept of multilevel inheritance and provide a C++ program example demonstrating multilevel inheritance involving classes Animal, Mammal, and Dog.

4. Explain the difference between a function declared inside a class and a function declared outside a class. In your answer, describe how member functions can be defined inside and outside a class in C++, and when each approach is preferred.

5. What is an inline function in C++? Discuss the scenarios where an inline function is beneficial, and explain how the compiler decides whether to actually inline a function. Provide an example of defining and using an inline function.

---

## 10-Mark Questions :

1. Write a detailed C++ program that demonstrates the creation and use of classes and objects. Define a Car class with data members like model, year, and price. Implement multiple constructors, member functions to set and get values, and a function to display car details. Ensure the program includes at least two objects of the Car class and displays their details.

2. Explain the principles of inheritance in C++, focusing on single, multiple, and multilevel inheritance. Write a C++ program that demonstrates multilevel inheritance, where a Vehicle class is inherited by Car, and Car is further inherited by ElectricCar. Discuss how constructors are called in the inheritance hierarchy.

3. Compare and contrast Object-Oriented Programming (OOP) and Procedure-Oriented Programming (POP). Provide examples in C++ to

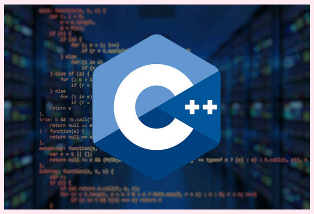
illustrate how OOP enables better code reuse and maintenance through encapsulation, inheritance, and polymorphism.

4. Discuss in detail the concepts of constructors and destructors in C++. Explain the types of constructors with code examples. Additionally, provide a scenario where the copy constructor is necessary. Discuss the role of destructors in memory management and resource release.

5. Explain in detail how namespaces are used to organize code and prevent clashes between identifiers. Provide a comprehensive C++ example demonstrating the creation and use of namespaces, and show how it can avoid name conflicts in a program with multiple libraries or modules.

# UNIT-2: Elementary Data Structures

Contact Hours: 10



## 1. Introduction to Data Structure

### Concept of Data and Information

- **Data** refers to raw facts and figures that by themselves may not have much meaning. For example, 23, John, 1001 are just data points.

- **Information** is the result of processing data in a way that it has meaning. For example, "John is 23 years old and his ID is 1001" is information derived from raw data.

### Introduction to Data Structures

A **data structure** is a way of organizing and storing data so that it can be accessed and modified efficiently. Data structures are essential for designing efficient algorithms and are classified into two main types:

- **Linear Data Structures**: Data elements are stored in a linear or sequential order.

    - Examples: Arrays, Linked Lists, Stacks, Queues

- **Non-Linear Data Structures**: Data elements are stored in a hierarchical or interconnected way.

    - Examples: Trees, Graphs

## Types of Data Structures

1. **Linear Data Structures**:
    - Data is organized in a sequential order where each element is connected to its previous and next element.
    - Examples: Arrays, Linked Lists, Stacks, Queues.
2. **Non-Linear Data Structures**:
    - Data elements are arranged in a hierarchical or interconnected manner.
    - Examples: Trees, Graphs.

## Operations on Data Structures

Common operations performed on data structures include:

- **Insertion**: Adding a new element to the data structure.
- **Deletion**: Removing an element from the data structure.
- **Traversal**: Accessing each element in the data structure.
- **Searching**: Finding a specific element in the data structure.
- **Sorting**: Arranging elements in a specific order (ascending or descending).

### Algorithm Complexity

Algorithm complexity refers to how the running time or space requirements of an algorithm grow as the size of the input data increases. Two important types of complexity are:

- **Time Complexity**: Measures how the time to complete an algorithm increases as the input size increases.

- **Space Complexity**: Measures how the memory requirements of an algorithm increase as the input size increases.

---

## 2. Arrays

### Basic Terminology

- **Array**: An array is a collection of elements, all of the same data type, stored in contiguous memory locations.

- **Index**: The position of an element in the array. Array indices typically start from 0.

- **Element**: Each item in the array is called an element.

### Linear Arrays and Their Representation

- A **linear array** is a simple data structure where elements are stored in a contiguous block of memory.

**Example** of an array:

```
int arr[5] = {10, 20, 30, 40, 50}; // Declaring and initializing an array
```

In this array, the element 10 is at index 0, 20 at index 1, and so on.

**Traversing a Linear Array**

- **Traversal** is the process of accessing and processing each element of the array.

- In C++, this can be done using a loop:

```
for(int i = 0; i < 5; i++) {
    cout << arr[i] << " "; // Outputs all elements of the array
}
```

**Insertion & Deletion in Arrays**

- **Insertion**: Adding an element at a specific index. Elements after the insertion point are shifted to the right.

- **Deletion**: Removing an element at a specific index. Elements after the deletion point are shifted to the left.

**Example of Insertion**:

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

int newValue = 25;

for (int i = 4; i >= 2; i--) {
    arr[i+1] = arr[i]; // Shift elements to the right
}

arr[2] = newValue; // Insert value 25 at index 2
```

**Example of Deletion**:

```
for (int i = 2; i < 4; i++) {
    arr[i] = arr[i+1]; // Shift elements to the left
}
```

**Searching in Arrays**

- **Linear Search**: In a linear search, each element of the array is checked one by one until the desired element is found.

**Example of Linear Search**:

```
int search(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) return i; // Return the index if key is found
    }
    return -1; // If key is not found
}
```

- **Binary Search**: Binary search is a more efficient method that works on sorted arrays. It repeatedly divides the search range in half until the element is found or the range is empty.

**Example of Binary Search**:

```
int binarySearch(int arr[], int low, int high, int key) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == key) return mid; // Return the index if key is found
        if (arr[mid] < key) low = mid + 1;
        else high = mid - 1;
    }
    return -1; // If key is not found
}
```

**Sorting in Arrays**

- **Bubble Sort**: Bubble sort compares adjacent elements and swaps them if they are in the wrong order. This process is repeated for each element until the array is sorted.

**Example of Bubble Sort**:

```
void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - 1 - i; j++) {
            if (arr[j] > arr[j+1]) {
                swap(arr[j], arr[j+1]); // Swap elements
            }
        }
    }
}
```

- **Insertion Sort**: In insertion sort, elements are picked one by one and inserted into their correct position in the sorted portion of the array.

**Example of Insertion Sort**:

```
void insertionSort(int arr[], int size) {
   for (int i = 1; i < size; i++) {
      int key = arr[i];
      int j = i - 1;
      while (j >= 0 && arr[j] > key) {
         arr[j+1] = arr[j]; // Shift elements to the right
         j--;
      }
      arr[j+1] = key; // Insert key in the correct position
   }
}
```

- **Selection Sort**: In selection sort, the smallest element is selected from the unsorted part and swapped with the first unsorted element.

**Example of Selection Sort**:

```
void selectionSort(int arr[], int size) {
   for (int i = 0; i < size - 1; i++) {
      int minIndex = i;
```

```
        for (int j = i + 1; j < size; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        swap(arr[i], arr[minIndex]); // Swap the smallest element with the first
unsorted element
    }
}
```

## 2D Array Declaration, Initialization, and Operations

A **2D array** is an array of arrays. It can be visualized as a table with rows and columns.

- **Declaration**:

```
int arr[3][3]; // A 2D array with 3 rows and 3 columns
```

- **Initialization**:

```
int arr[2][2] = {{1, 2}, {3, 4}}; // Initializing a 2x2 array
```

- **Traversing 2D Array**:

```
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
        cout << arr[i][j] << " "; // Output each element in the 2D array
    }
}
```

Operations on 2D arrays include insertion, deletion, searching, and sorting (similar to 1D arrays).

---

## 3. Pointers

### Introduction to Pointers

A **pointer** is a variable that stores the memory address of another variable. Pointers are essential for dynamic memory allocation, arrays, and data structures like linked lists.

**Example of Pointer Declaration**:

```
int x = 10;

int *ptr = &x; // Pointer to integer, stores address of x
```

### Concept of Linked List

A **linked list** is a linear data structure where elements (nodes) are stored at non-contiguous memory locations. Each node has two parts:

1. **Data**: Stores the value.
2. **Next**: A pointer that points to the next node in the list.

Linked lists allow efficient insertions and deletions but require extra memory for pointers.

**Example of a Linked List Node**:

cpp

CopyEdit

```
struct Node {
    int data;
    Node* next; // Pointer to the next node
};
```

A **singly linked list** is where each node points to the next node, and the last node points to NULL.

# Question Bank for Unit 2 : Elementary Data Structures

## 2-Mark Questions

1. What is the difference between static and dynamic data structures? Provide an example of each type and explain their use cases.

2. Define the concept of time complexity in algorithms. What is the time complexity of a linear search algorithm, and why is it considered inefficient for large data sets?

3. What is the significance of using a pointer in C++? How does a pointer help in managing dynamic memory allocation in the context of a linked list?

4. Explain what a 2D array is. How are elements accessed in a 2D array in C++? Illustrate with a simple example of initializing and printing a 2D array.

5. Describe the insertion sort algorithm. How does it work? What is its time complexity in the worst case, and how does it compare to bubble sort?

---

## 5-Mark Questions

1. Write a detailed explanation of the differences between linear and non-linear data structures. Create a real-life scenario where each type

would be used, explaining the advantages of using one over the other in your example.

2. Implement a C++ program to perform the following operations on an array of integers: insertion, deletion, and searching. Include code to handle boundary conditions like inserting into a full array or deleting from an empty array. After completing the program, describe the efficiency of each operation in terms of time complexity.

3. Describe the binary search algorithm. How does it compare to linear search in terms of performance? Write a C++ program that implements binary search on a sorted array, and explain the steps with sample data.

4. Create a C++ program that demonstrates the use of a linked list. Implement functions to insert and delete elements at the beginning, middle, and end of the list. Provide a clear explanation of how pointers are used to manage the linked list structure.

5. Write and explain a C++ program to implement bubble sort on a list of integers. After sorting the list, analyze its time complexity and compare it with the time complexities of other sorting algorithms like insertion sort and quicksort. Which one would be more efficient for large data sets?

---

## 10-Mark Questions

1. Explain the concept of algorithm complexity in detail. Discuss both time complexity and space complexity, comparing different sorting algorithms like bubble sort, insertion sort, and selection sort. Write C++ programs to demonstrate each sorting algorithm, and analyze their efficiency by

running them on data sets of different sizes. Include a conclusion on when to use each sorting technique based on your analysis.

2. Implement a C++ program that demonstrates an array-based implementation of a queue (FIFO). Include functions to enqueue, dequeue, and display the queue. Additionally, perform boundary checks for underflow and overflow conditions. Explain how circular queues improve efficiency in such scenarios.

3. Write a detailed explanation and C++ program to implement a singly linked list. Include operations to insert and delete elements at the head, tail, and at a specified position. Discuss how the memory allocation and deallocation process works when manipulating a linked list using pointers. Provide examples of real-life problems that linked lists help solve efficiently.

4. Design a program that reads a list of integers and performs the following tasks:

    - Sort the array using any one sorting algorithm (e.g., bubble sort, quicksort).

    - Implement a binary search to find a specific value in the sorted array.

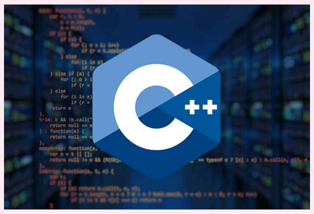
    - Include performance comparisons in terms of time complexity for sorting and searching.

5. Discuss how sorting and searching algorithms impact the overall performance of the program and how different algorithms behave as the input size increases.

6. Discuss the use of pointers in C++ for dynamic memory management. Write a program that creates a dynamic 2D array using pointers. Implement functions to initialize the array, assign values, and print the array. Afterward, explain how memory is managed in this scenario, how memory leaks can occur, and how to prevent them using proper memory deallocation techniques.

# UNIT-3: Stack, Queue, Linked List

Contact Hours: 10



## 1. Linked List

A **linked list** is a collection of elements (called nodes) that are connected using pointers. Unlike arrays, where data is stored in contiguous memory locations, linked lists use non-contiguous memory locations and have the ability to grow or shrink dynamically, making them more flexible for certain types of data management.

### Structure of a Linked List Node:

Each node in a linked list has:

1. **Data**: The information stored in the node (could be integers, strings, etc.).
2. **Next Pointer**: A reference or pointer to the next node in the list. In the last node, this pointer points to NULL (or nullptr in C++).

### Example of a linked list node structure in C++:

```
struct Node {
    int data;       // Data to store
    Node* next;     // Pointer to the next node
};
```

### Types of Linked Lists:

1. **Singly Linked List**: Each node points to the next node in the sequence, and the last node points to NULL. This is the most basic form of a linked list.
    - **Structure**: Each node has data and a pointer to the next node.
    - **Example**: Head -> Node1 -> Node2 -> NULL
2. **Doubly Linked List**: Each node contains two pointers: one pointing to the next node and another pointing to the previous node. This allows traversal in both directions, making certain operations easier (e.g., deletion from both ends).
    - **Structure**: Each node has data, a pointer to the next node, and a pointer to the previous node.
    - **Example**: NULL <- Head <-> Node1 <-> Node2 -> NULL
3. **Circular Linked List**: In this type of list, the last node points back to the first node, forming a circular structure. It can be either singly or doubly linked.
    - **Structure**: The last node points back to the head or first node.
    - **Example**: Head -> Node1 -> Node2 -> Head

**Memory Representation:**

- **Singly Linked List**: The nodes are scattered in memory, and each node's next pointer holds the memory address of the next node.
    - A linked list is flexible since nodes can be added or removed dynamically without shifting the rest of the elements.
- **Doubly Linked List**: Each node has two pointers: one for the next node and one for the previous node. This makes it easier to traverse both ways but

requires more memory per node.

**Operations on Linked List:**

1. **Insertion**:

    - **At the beginning**: You create a new node and make its next pointer point to the current head. Then, the head pointer is updated to point to the new node.

    - **At the end**: You traverse to the last node, and then add the new node by making the last node's next pointer point to the new node.

    - **At a specific position**: You traverse the list until the desired position and adjust the pointers to insert the new node at that position.

**Example of insertion at the beginning**:

```
void insertAtBeginning(Node*& head, int value) {
    Node* newNode = new Node;
   newNode->data = value;
   newNode->next = head;  // Point to the old head
    head = newNode;        // Update head to new node
}
```

2. **Deletion**:

    - **From the beginning**: The head pointer is moved to the next node, effectively removing the first node.

- **From the end**: Traverse the list to find the second-to-last node and set its next pointer to NULL.
- **At a specific position**: Traverse the list to the node before the one you want to delete, adjust the pointers to exclude the node, and free the memory.

**Example of deletion at the beginning**:

```
void deleteAtBeginning(Node*& head) {
    if (head != NULL) {
        Node* temp = head;
        head = head->next;  // Move head to next node
        delete temp;        // Delete old head
    }
}
```

3. **Traversal**: To visit all nodes in the list, you start from the head and follow the next pointers until you reach NULL.

   **Example of traversal**:

```
        void traverse(Node* head) {
    Node* current = head;
    while (current != NULL) {
        cout << current->data << " ";
        current = current->next;  // Move to the next node
    }
```

}

4. **Advantages of Linked Lists:**

- **Dynamic Size**: They can grow and shrink dynamically, unlike arrays.
- **Efficient Insertions/Deletions**: Inserting or deleting an element does not require shifting other elements, unlike in arrays.

**Disadvantages of Linked Lists:**

- **Extra Memory**: Each node requires extra memory for the pointer(s).
- **Traversal**: Accessing elements requires sequential traversal, making it slower compared to arrays for direct access.

---

## 2. Stacks

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle, where the last element added to the stack is the first one to be removed.

**Basic Terminology:**

- **Push**: Adds an element to the top of the stack.
- **Pop**: Removes the top element from the stack.
- **Top/Peek**: Returns the top element of the stack without removing it.
- **IsEmpty**: Checks whether the stack is empty.
- **IsFull**: Checks whether the stack is full (relevant in array-based implementation).

**Sequential Representation (Array-based Stack):**

In an array-based stack, an array is used to store the elements. A pointer or variable (e.g., top) keeps track of the index of the top element.

**Example of stack implementation using an array**:

```
#define MAX 5

int stack[MAX];

int top = -1;  // Initially empty


// Push operation

void push(int value) {

    if (top == MAX - 1) {

        cout << "Stack Overflow" << endl;

    } else {

        stack[++top] = value;

    }

}


// Pop operation

int pop() {

    if (top == -1) {

        cout << "Stack Underflow" << endl;

        return -1;

    } else {
        return stack[top--];
    }
}

// Peek operation
int peek() {
    if (top == -1) {
        cout << "Stack is empty" << endl;
        return -1;
    } else {
        return stack[top];
    }
}
```

**Linked Representation (Linked List-based Stack):**

In a linked list-based stack, each element (node) contains a pointer to the next element. The stack operations are handled using the head pointer.

**Example of stack implementation using a linked list**:struct Node {

```
    int data;
    Node* next;
};

Node* top = NULL;  // Initially empty
```

```
// Push operation
void push(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = top;
    top = newNode;
}

// Pop operation
int pop() {
    if (top == NULL) {
        cout << "Stack Underflow" << endl;
        return -1;
    } else {
        Node* temp = top;
        int value = top->data;
        top = top->next;
        delete temp;
        return value;
    }
}
```

**Applications of Stacks:**

1. **Expression Evaluation**: Stacks are widely used for evaluating postfix and prefix expressions.

2. **Infix to Postfix Conversion**: When expressions are written in infix notation (like (a + b)), stacks can be used to convert them into postfix form (like a b +), which is easier to evaluate programmatically.

---

## 3. Queue

A **queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle, meaning the first element added to the queue is the first one to be removed.

**Types of Queues:**

1. **Linear Queue**: A basic queue where elements are added at the rear and removed from the front.

2. **Circular Queue**: In a circular queue, the last element points to the first element, forming a circular structure. This avoids the problem of wasted space in linear queues when elements are dequeued from the front.

3. **Double-ended Queue (Deque)**: Allows insertion and removal of elements from both ends.

**Operations on Queue:**

1. **Enqueue**: Adds an element to the rear of the queue.

2. **Dequeue**: Removes an element from the front of the queue.
3. **Front**: Returns the element at the front without removing it.
4. **Rear**: Returns the last element without removing it.
5. **IsEmpty**: Checks whether the queue is empty.
6. **IsFull**: Checks whether the queue is full.

**Sequential Representation (Array-based Queue):**

In an array-based queue, elements are stored in an array, and two pointers (front and rear) are used to manage the queue. When an element is dequeued, the front pointer is incremented. When an element is enqueued, the rear pointer is incremented.

**Example of queue implementation using an array**:

```
#define MAX 5

int queue[MAX];

int front = -1, rear = -1;


// Enqueue operation

void enqueue(int value) {

    if (rear == MAX - 1) {

        cout << "Queue Overflow" << endl;

    } else {

        if (front == -1) front = 0;
        queue[++rear] = value;
    }
}

// Dequeue operation
int dequeue() {
    if (front == -1 || front > rear) {
        cout << "Queue Underflow" << endl;
        return -1;
    } else {
        return queue[front++];
    }
}
```

**Circular Queue Representation:**

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

**Example of circular queue implementation**:

```
#define MAX 5
int queue[MAX];
int front = -1, rear = -1;

// Enqueue operation
```

```
void enqueue(int value) {
    if ((rear + 1) % MAX == front) {
        cout << "Queue Overflow" << endl;
    } else {
        if (front == -1) front = 0;
        rear = (rear + 1) % MAX;
        queue[rear] = value;
    }
}

// Dequeue operation
int dequeue() {
    if (front == -1) {
        cout << "Queue Underflow" << endl;
        return -1;
    } else {
        int value = queue[front];
        if (front == rear) {
            front = rear = -1;
        } else {
            front = (front + 1) % MAX;
        }
        return value;
    }
}
```

**Applications of Queues:**

1. **Scheduling**: Queues are often used in scheduling problems (e.g., CPU scheduling, printer queues).

2. **Buffering**: In systems with buffer management (like I/O operations), queues help in managing the flow of data.

# Question Bank for Unit 3 : Linked Lists, Stacks & Queues

## 2-MARK QUESTIONS:

1. Define a stack data structure and explain the concept of LIFO (Last In, First Out) with a simple example.

2. How does a queue implement the FIFO (First In, First Out) principle? Give an example of its real-world application.

3. What is a linked list? How is it different from an array in terms of memory allocation and access?

4. Explain the difference between a linear queue and a circular queue. When is a circular queue preferred over a linear queue?

5. What are the key operations associated with a stack? Briefly explain the purpose of each operation (Push, Pop, Peek).

---

## 5-MARK QUESTIONS:

1. Compare the structure and operations of a stack with that of a queue. Provide real-world scenarios where each data structure is useful.

2. Explain the process of traversing a singly linked list. How would you traverse it iteratively and recursively? Provide examples.

3. Discuss the operations on a circular queue, highlighting the differences between its sequential and circular representation. Why is circular queue preferred in scenarios like CPU scheduling?

4. What are the advantages of using linked lists over arrays? Discuss memory allocation and efficiency when using linked lists for dynamic data storage.

5. How can a stack be used to evaluate postfix expressions? Illustrate the process with an example of a postfix expression and explain step-by-step how the stack is utilized.

## 10-MARK QUESTIONS:

1. Compare and contrast the stack and queue data structures in terms of their structure, operations, and use cases. Discuss scenarios where one would be preferred over the other, and include examples like expression evaluation, undo operations, and scheduling tasks.

2. Design and implement a C++ program that performs operations on a singly linked list. Your program should include functions for insertion, deletion, and traversal. Explain the advantages and challenges of using a linked list for dynamic memory management.

3. Write a detailed explanation of how a circular queue works. Provide an implementation in C++ and compare it with the regular linear queue. Discuss the scenarios where a circular queue offers better performance in terms of space and time complexity.

4. Implement a C++ program that converts an infix expression to a postfix expression using a stack. Provide detailed step-by-step examples to demonstrate the conversion process.

5. Design and implement a double-ended queue (Deque) using both an array and a doubly linked list. Compare both implementations, highlighting the advantages and disadvantages of each with respect to operations like insertion, deletion, and accessing elements from both ends.