# DATA STRUCTURES: STACK, QUEUE , Double – Ended Queue

1. Stack basics: structure , operations (push , pop , peek ), Implementation using Array and Linked List.

ANS:

## Stack Basics

A stack is a linear data structure that follows the Last In First Out (LIFO) principle. The basic operations of a stack are:

- **Push:** Add an element to the top of the stack.
- **Pop:** Remove and return the top element of the stack.
- **Peek/Top:** Return the top element without removing it from the stack.
- **isEmpty:** Check if the stack is empty

Implementation of Stack using Array

```
#include <iostream>

#define MAX 1000


class Stack {
    int top;
public:
    int arr[MAX]; // Maximum size of Stack


    Stack() { top = -1; }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
};
```

```
bool Stack::push(int x) {
    if (top >= (MAX - 1)) {
        std::cout << "Stack Overflow" << std::endl;
        return false;
    } else {
        arr[++top] = x;
        std::cout << x << " pushed into stack" << std::endl;
        return true;
    }
}

int Stack::pop() {
    if (top < 0) {
        std::cout << "Stack Underflow" << std::endl;
        return 0;
    } else {
        int x = arr[top--];
        return x;
    }
}

int Stack::peek() {
    if (top < 0) {
        std::cout << "Stack is Empty" << std::endl;
```

```
        return 0;
    } else {
        int x = arr[top];
        return x;
    }
}

bool Stack::isEmpty() {
    return (top < 0);
}

int main() {
    Stack stack;
    stack.push(10);
    stack.push(20);
    stack.push(30);
    std::cout << stack.pop() << " popped from stack\n";
    std::cout << "Top element is " << stack.peek() << std::endl;
    std::cout << "Stack is " << (stack.isEmpty() ? "empty" : "not empty") <<
std::endl;
    return 0;
}
```

Implementation of Stack using Linked List

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

```
class Node {
public:
    int data;
    Node* next;

    Node(int value) {
        data = value;
        next = nullptr;
    }
};

class StackLinkedList {
private:
    Node* top;

public:
    StackLinkedList() {
        top = nullptr; // Initialize top to nullptr when stack is empty
    }

    bool isEmpty() {
        return (top == nullptr);
    }
```

```
void push(int x) {
    Node* newNode = new Node(x);
    newNode->next = top;
    top = newNode;
    cout << x << " pushed into stack\n";
}

void pop() {
    if (isEmpty()) {
        cout << "Error: Stack underflow\n";
        return;
    }
    Node* temp = top;
    top = top->next;
    int popped = temp->data;
    delete temp;
    cout << popped << " popped from stack\n";
}

int peek() {
    if (isEmpty()) {
        cout << "Error: Stack is empty\n";
        return -1;
```

```
        }
        return top->data;
    }
};

int main() {
    StackLinkedList stack;

    stack.push(10);
    stack.push(20);
    stack.push(30);

    cout << "Top element is: " << stack.peek() << endl;

    stack.pop();
    stack.pop();

    cout << "Top element is: " << stack.peek() << endl;

    stack.pop();
    stack.pop(); // This will cause underflow

    return 0;
}
```

2.Stack application: expression evaluation , function call ,undo mechanisms.

ANS:

### ♦Expression Evaluation

Stacks are commonly used to evaluate arithmetic expressions, both infix and postfix (Reverse Polish Notation, RPN). Here, we'll focus on evaluating postfix expressions using a stack.

#### *Example: Evaluating Postfix Expression*

Given a postfix expression `"34+2*"`:

```
#include <iostream>
#include <stack>
#include <string>
#include <cctype> // for isdigit()

using namespace std;

int evaluatePostfix(const string& expr) {
    stack<int> stk;

    for (char c : expr) {
        if (isdigit(c)) {
            stk.push(c - '0'); // Convert char to int and push to stack
        } else {
            int operand2 = stk.top();
            stk.pop();
            int operand1 = stk.top();
            stk.pop();
```

```
            switch (c) {
                case '+':
                    stk.push(operand1 + operand2);
                    break;
                case '-':
                    stk.push(operand1 - operand2);
                    break;
                case '*':
                    stk.push(operand1 * operand2);
                    break;
                case '/':
                    stk.push(operand1 / operand2);
                    break;
            }
        }
    }

    return stk.top();
}

int main() {
    string postfix_expr = "34+2*";
    cout << "Postfix expression evaluation result: " <<
evaluatePostfix(postfix_expr) << endl;
```

```
    return 0;
}
```

## ♦Function Call Management (Call Stack)

In programming languages, function calls are managed using a call stack. When a function is called, its context (variables, return address, etc.) is pushed onto the stack, and when it returns, its context is popped off the stack.

***Example: Simulating Function Calls with a Stack***

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

void func1() {
    cout << "Inside func1()\n";
}

void func2() {
    cout << "Inside func2()\n";
    func1();
}

void func3() {
    cout << "Inside func3()\n";
    func2();
}
```

```
int main() {
    stack<string> callStack;

    callStack.push("main()");
    cout << "Call Stack: main()\n";

    func3();

    callStack.pop();
    cout << "Call Stack: " << callStack.top() << endl;

    return 0;
}
```

**♦Undo Mechanism**

Stacks are also useful for implementing undo mechanisms in applications where users can perform actions that need to be reversible.

***Example: Implementing an Undo Mechanism***

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

class TextEditor {
private:
    string text;
```

```
    stack<string> undoStack;

public:
    TextEditor() {
        text = "";
    }

    void addText(const string& newText) {
        text += newText;
        undoStack.push("add " + newText);
        cout << "Text added: " << newText << endl;
    }

    void undo() {
        if (!undoStack.empty()) {
            string lastAction = undoStack.top();
            undoStack.pop();

            if (lastAction.substr(0, 3) == "add") {
                string addedText = lastAction.substr(4); // Extract added text
                text.erase(text.length() - addedText.length()); // Remove last added
text
                cout << "Undo: Removed text '" << addedText << "'" << endl;
            }
        } else {
```

```
            cout << "Nothing to undo\n";
        }
    }

    void displayText() {
        cout << "Current Text: " << text << endl;
    }
};

int main() {
    TextEditor editor;

    editor.addText("Hello ");
    editor.addText("World!");

    editor.displayText();

    editor.undo();

    editor.displayText();

    editor.undo(); // Nothing to undo

    return 0;
```

}

3.Queue basics: Structure , operation (Enqueue , Dequeue , Peek) , Implementation Using Array and linked list

ANS:

### ♦Queue Basics

A queue is a linear data structure that follows the First In, First Out (FIFO) principle, meaning the element that is inserted first is the first one to be removed. It supports the following operations:

- **Enqueue:** Adds an element to the rear (end) of the queue.
- **Dequeue:** Removes the element from the front (beginning) of the queue.
- **Peek (or Front):** Returns the element at the front of the queue without removing it.

♦Implementation Using Arrays

```
#include <iostream>

using namespace std;


#define MAX_SIZE 100 // Maximum size of the queue


class QueueArray {

private:

    int arr[MAX_SIZE];

    int front, rear;


public:

    QueueArray() {

        front = -1; // Initialize front and rear to -1 when queue is empty

        rear = -1;

    }
```

```
bool isEmpty() {
    return (front == -1 && rear == -1);
}

bool isFull() {
    return (rear == MAX_SIZE - 1);
}

void enqueue(int x) {
    if (isFull()) {
        cout << "Error: Queue is full\n";
        return;
    } else if (isEmpty()) {
        front = rear = 0;
    } else {
        rear++;
    }
    arr[rear] = x;
    cout << x << " enqueued into queue\n";
}

void dequeue() {
    if (isEmpty()) {
```

```
            cout << "Error: Queue is empty\n";
            return;
        } else if (front == rear) {
            cout << arr[front] << " dequeued from queue\n";
            front = rear = -1;
        } else {
            cout << arr[front] << " dequeued from queue\n";
            front++;
        }
    }

    int peek() {
        if (isEmpty()) {
            cout << "Error: Queue is empty\n";
            return -1;
        }
        return arr[front];
    }
};

int main() {
    QueueArray queue;

    queue.enqueue(10);
```

```
    queue.enqueue(20);
    queue.enqueue(30);

    cout << "Front element is: " << queue.peek() << endl;

    queue.dequeue();
    queue.dequeue();

    cout << "Front element is: " << queue.peek() << endl;

    queue.dequeue();
    queue.dequeue(); // This will cause underflow

    return 0;
}
```

♦Implementation Using Linked List

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

class Node {
public:
    int data;
    Node* next;
```

```
    Node(int value) {
        data = value;
        next = nullptr;
    }
};

class QueueLinkedList {
private:
    Node* front;
    Node* rear;

public:
    QueueLinkedList() {
        front = nullptr; // Initialize front and rear to nullptr when queue is empty
        rear = nullptr;
    }

    bool isEmpty() {
        return (front == nullptr);
    }

    void enqueue(int x) {
        Node* newNode = new Node(x);
        if (isEmpty()) {
```

```
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
    cout << x << " enqueued into queue\n";
}

void dequeue() {
    if (isEmpty()) {
        cout << "Error: Queue is empty\n";
        return;
    }
    Node* temp = front;
    cout << temp->data << " dequeued from queue\n";
    front = front->next;
    delete temp;
    if (front == nullptr) {
        rear = nullptr;
    }
}

int peek() {
    if (isEmpty()) {
```

```
            cout << "Error: Queue is empty\n";
            return -1;
        }
        return front->data;
    }
};

int main() {
    QueueLinkedList queue;

    queue.enqueue(10);
    queue.enqueue(20);
    queue.enqueue(30);

    cout << "Front element is: " << queue.peek() << endl;

    queue.dequeue();
    queue.dequeue();

    cout << "Front element is: " << queue.peek() << endl;

    queue.dequeue();
    queue.dequeue(); // This will cause underflow
```

```
    return 0;
}
```

4.Queue Applications: BFS traversal , priority Queue , scheduling Algorithms .

ANS:

### ♦BFS Traversal (Breadth-First Search)

Breadth-First Search (BFS) is a graph traversal algorithm that visits all the vertices of a graph in breadthward motion. It uses a queue to keep track of vertices to be explored next.

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

class Graph {
private:
    int V; // Number of vertices
    vector<vector<int>> adj; // Adjacency list

public:
    Graph(int vertices) {
        V = vertices;
        adj.resize(V);
    }

    void addEdge(int u, int v) {
        adj[u].push_back(v);
```

```
        adj[v].push_back(u); // Uncomment for undirected graph
    }

    void BFS(int startVertex) {
        vector<bool> visited(V, false);
        queue<int> q;

        visited[startVertex] = true;
        q.push(startVertex);

        while (!q.empty()) {
            int current = q.front();
            q.pop();
            cout << current << " ";

            for (int neighbor : adj[current]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
        cout << endl;
    }
};
```

```
int main() {
    Graph g(6); // Create a graph with 6 vertices

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 4);
    g.addEdge(3, 5);
    g.addEdge(4, 5);

    cout << "BFS traversal starting from vertex 0: ";
    g.BFS(0);

    return 0;
}
```

## ♦Priority Queue

A priority queue is an abstract data type similar to a regular queue or stack, but where each element has a priority assigned to it. In a priority queue, an element with higher priority is dequeued before an element with lower priority, regardless of the order in which they were added.

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

int main() {
```

```
    priority_queue<int> pq;

    pq.push(30);
    pq.push(10);
    pq.push(20);

    cout << "Priority Queue elements: ";

    while (!pq.empty()) {
        cout << pq.top() << " ";
        pq.pop();
    }

    cout << endl;

    return 0;
}
```

## ♦Scheduling Algorithms

In operating systems and computer systems, scheduling algorithms manage how resources are allocated to different tasks or processes. Queues, especially priority queues, are used to prioritize tasks based on certain criteria (e.g., CPU burst time, priority level) and efficiently manage their execution order.

Example: Scheduling Tasks Using Priority Queue (Simplified)

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

```
class Task {
public:
    string name;
    int priority;

    Task(string n, int p) : name(n), priority(p) {}

    bool operator<(const Task& other) const {
        // Higher priority tasks should have higher priority in the priority queue
        return priority < other.priority;
    }
};

int main() {
    priority_queue<Task> taskQueue;

    taskQueue.push(Task("Task1", 3));
    taskQueue.push(Task("Task2", 1));
    taskQueue.push(Task("Task3", 2));

    cout << "Tasks scheduled based on priority:\n";
    while (!taskQueue.empty()) {
        Task currentTask = taskQueue.top();
        taskQueue.pop();
```

```
        cout << "Task: " << currentTask.name << " (Priority: " << currentTask.priority << ")\n";
    }


    return 0;
}
```

## 5. Deque (Double – Ended Queue): Structure , operation , implementation

ANS:

### ♦Deque Basics

A deque, pronounced as "deck", is a double-ended queue that allows insertion and deletion of elements from both ends. It combines the functionalities of a stack and a queue, providing efficient insertion and deletion operations at both ends of the data structure.

### ♦Structure

A deque can be implemented using arrays or linked lists. Here's a brief overview of its structure:

- **Array-based Deque:** Uses a dynamic array to store elements, allowing efficient random access but potentially requiring resizing operations when full.
- **Linked List-based Deque:** Uses a doubly linked list where each node contains a data element and pointers to the next and previous nodes. This implementation allows for efficient insertion and deletion operations at both ends without the need for resizing.

### ♦Operations

A deque typically supports the following operations:

- **Insertion Operations:**
  - **Push Front:** Adds an element to the front of the deque.
  - **Push Back:** Adds an element to the back of the deque.
- **Deletion Operations:**
  - **Pop Front:** Removes an element from the front of the deque.
  - **Pop Back:** Removes an element from the back of the deque.
- **Access Operations:**
  - **Front:** Returns the element at the front of the deque without removing it.
  - **Back:** Returns the element at the back of the deque without removing it.
- **Other Operations:**
  - **Size:** Returns the number of elements in the deque.
  - **Empty:** Checks if the deque is empty.

♦Implementation Using Arrays

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

#define MAX_SIZE 100 // Maximum size of the deque

class DequeArray {
private:
    int arr[MAX_SIZE];
    int front, rear;
    int size;

public:
    DequeArray() {
        front = -1; // Initialize front and rear to -1 when deque is empty
        rear = 0;
        size = 0;
    }

    bool isEmpty() {
        return size == 0;
    }

    bool isFull() {
```

```
    return size == MAX_SIZE;
}

void pushFront(int x) {
    if (isFull()) {
        cout << "Error: Deque is full\n";
        return;
    }
    if (front == -1) {
        front = rear = 0;
    } else if (front == 0) {
        front = MAX_SIZE - 1;
    } else {
        front--;
    }
    arr[front] = x;
    size++;
    cout << x << " pushed to front\n";
}

void pushBack(int x) {
    if (isFull()) {
        cout << "Error: Deque is full\n";
        return;
```

```
    }
    if (front == -1) {
        front = rear = 0;
    } else if (rear == MAX_SIZE - 1) {
        rear = 0;
    } else {
        rear++;
    }
    arr[rear] = x;
    size++;
    cout << x << " pushed to back\n";
}

void popFront() {
    if (isEmpty()) {
        cout << "Error: Deque is empty\n";
        return;
    }
    int popped = arr[front];
    if (front == rear) {
        front = -1;
        rear = 0;
    } else if (front == MAX_SIZE - 1) {
        front = 0;
```

```
    } else {
        front++;
    }
    size--;
    cout << popped << " popped from front\n";
}

void popBack() {
    if (isEmpty()) {
        cout << "Error: Deque is empty\n";
        return;
    }
    int popped = arr[rear];
    if (front == rear) {
        front = -1;
        rear = 0;
    } else if (rear == 0) {
        rear = MAX_SIZE - 1;
    } else {
        rear--;
    }
    size--;
    cout << popped << " popped from back\n";
}
```

```
    int getFront() {
        if (isEmpty()) {
            cout << "Error: Deque is empty\n";
            return -1;
        }
        return arr[front];
    }

    int getBack() {
        if (isEmpty()) {
            cout << "Error: Deque is empty\n";
            return -1;
        }
        return arr[rear];
    }

    int getSize() {
        return size;
    }
};

int main() {
    DequeArray deque;
```

```
    deque.pushBack(10);
    deque.pushBack(20);
    deque.pushFront(30);

    cout << "Front element: " << deque.getFront() << endl;
    cout << "Back element: " << deque.getBack() << endl;

    deque.popFront();
    deque.popBack();

    cout << "Front element: " << deque.getFront() << endl;
    cout << "Back element: " << deque.getBack() << endl;

    return 0;
}
```

♦Implementation Using Linked List

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

class Node {
public:
    int data;
    Node* prev;
```

```
    Node* next;

    Node(int value) {
        data = value;
        prev = nullptr;
        next = nullptr;
    }
};

class DequeLinkedList {
private:
    Node* front;
    Node* rear;
    int size;

public:
    DequeLinkedList() {
        front = nullptr; // Initialize front and rear to nullptr when deque is empty
        rear = nullptr;
        size = 0;
    }

    bool isEmpty() {
        return size == 0;
```

```
}

void pushFront(int x) {
    Node* newNode = new Node(x);
    if (isEmpty()) {
        front = rear = newNode;
    } else {
        newNode->next = front;
        front->prev = newNode;
        front = newNode;
    }
    size++;
    cout << x << " pushed to front\n";
}

void pushBack(int x) {
    Node* newNode = new Node(x);
    if (isEmpty()) {
        front = rear = newNode;
    } else {
        rear->next = newNode;
        newNode->prev = rear;
        rear = newNode;
    }
```

```
    size++;
    cout << x << " pushed to back\n";
}

void popFront() {
    if (isEmpty()) {
        cout << "Error: Deque is empty\n";
        return;
    }
    Node* temp = front;
    int popped = temp->data;
    if (front == rear) {
        front = rear = nullptr;
    } else {
        front = front->next;
        front->prev = nullptr;
    }
    delete temp;
    size--;
    cout << popped << " popped from front\n";
}

void popBack() {
    if (isEmpty()) {
```

```
        cout << "Error: Deque is empty\n";
        return;
    }
    Node* temp = rear;
    int popped = temp->data;
    if (front == rear) {
        front = rear = nullptr;
    } else {
        rear = rear->prev;
        rear->next = nullptr;
    }
    delete temp;
    size--;
    cout << popped << " popped from back\n";
}

int getFront() {
    if (isEmpty()) {
        cout << "Error: Deque is empty\n";
        return -1;
    }
    return front->data;
}
```

```
    int getBack() {
        if (isEmpty()) {
            cout << "Error: Deque is empty\n";
            return -1;
        }
        return rear->data;
    }

    int getSize() {
        return size;
    }
};

int main() {
    DequeLinkedList deque;

    deque.pushBack(10);
    deque.pushBack(20);
    deque.pushFront(30);

    cout << "Front element: " << deque.getFront() << endl;
    cout << "Back element: " << deque.getBack() << endl;

    deque.popFront();
```

```
    deque.popBack();

    cout << "Front element: " << deque.getFront() << endl;
    cout << "Back element: " << deque.getBack() << endl;

    return 0;
}
```