AP-Assignment – 6

Name : - Vipul || Uid :- 22BCS16942 || Section: - 612-B

Stack Implementation :

1. **Implement Queue using Stack**

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

class QueueUsingStack {
    stack<int> s1, s2;
public:
    void enqueue(int x) {
        s1.push(x);
    }

    int dequeue() {
        if (s2.empty()) {
            while (!s1.empty()) {
                s2.push(s1.top());
                s1.pop();
            }
        }
        if (s2.empty()) {
            throw runtime_error("Queue is empty");
        }
        int top = s2.top();
```

```cpp
        s2.pop();
        return top;
    }
};
```

2. **Implement Deque using Stack**

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

class DequeUsingStack {
    stack<int> frontStack, backStack;
public:
    void pushFront(int x) {
        frontStack.push(x);
    }

    void pushBack(int x) {
        backStack.push(x);
    }

    int popFront() {
        if (frontStack.empty() && backStack.empty()) {
            throw runtime_error("Deque is empty");
        }
        if (frontStack.empty()) {
            while (!backStack.empty()) {
                frontStack.push(backStack.top());
                backStack.pop();
```

```cpp
            }
        }
        int top = frontStack.top();
        frontStack.pop();
        return top;
    }

    int popBack() {
        if (frontStack.empty() && backStack.empty()) {
            throw runtime_error("Deque is empty");
        }
        if (backStack.empty()) {
            while (!frontStack.empty()) {
                backStack.push(frontStack.top());
                frontStack.pop();
            }
        }
        int top = backStack.top();
        backStack.pop();
        return top;
    }
};
```

3. **Implement Min Stack using Two Stacks**

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

class MinStack {
```

```
    stack<int> mainStack, minStack;
public:
    void push(int x) {
        mainStack.push(x);
        if (minStack.empty() || x <= minStack.top()) {
            minStack.push(x);
        }
    }

    void pop() {
        if (mainStack.top() == minStack.top()) {
            minStack.pop();
        }
        mainStack.pop();
    }

    int top() {
        return mainStack.top();
    }

    int getMin() {
        return minStack.top();
    }
};
```

4. **Implement Max Stack using Two Stacks**

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

```cpp
class MaxStack {
    stack<int> mainStack, maxStack;
public:
    void push(int x) {
        mainStack.push(x);
        if (maxStack.empty() || x >= maxStack.top()) {
            maxStack.push(x);
        }
    }

    void pop() {
        if (mainStack.top() == maxStack.top()) {
            maxStack.pop();
        }
        mainStack.pop();
    }

    int top() {
        return mainStack.top();
    }

    int getMax() {
        return maxStack.top();
    }
};
```

5. **Implement Priority Queue using Stack**

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

```
class PriorityQueueUsingStack {
    stack<int> s;
public:
    void push(int x) {
        stack<int> temp;
        while (!s.empty() && s.top() > x) {
            temp.push(s.top());
            s.pop();
        }
        s.push(x);
        while (!temp.empty()) {
            s.push(temp.top());
            temp.pop();
        }
    }

    int pop() {
        if (s.empty()) {
            throw runtime_error("Priority Queue is empty");
        }
        int top = s.top();
        s.pop();
        return top;
    }

    int top() {
        if (s.empty()) {
            throw runtime_error("Priority Queue is empty");
```

```cpp
        }
        return s.top();
    }
};
```

6. **Implement BST (Inorder Traversal) using Stack (Iterative DFS)**

```cpp
#include <stack>
#include <vector>
using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

vector<int> inorderTraversal(TreeNode* root) {
    vector<int> result;
    stack<TreeNode*> s;
    TreeNode* current = root;

    while (current != nullptr || !s.empty()) {
        while (current != nullptr) {
            s.push(current);
            current = current->left;
        }
        current = s.top();
```

```
        s.pop();
        result.push_back(current->val);
        current = current->right;
    }

    return result;
}
```

Queue :

8. **Implement Stack using Queue**

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

class StackUsingQueue {
    queue<int> q1, q2;
public:
    void push(int x) {
        q2.push(x);
        while (!q1.empty()) {
            q2.push(q1.front());
            q1.pop();
        }
        swap(q1, q2);
    }

    int pop() {
        if (q1.empty()) {
            throw runtime_error("Stack is empty");
```

```
        }
        int top = q1.front();
        q1.pop();
        return top;
    }

    int top() {
        if (q1.empty()) {
            throw runtime_error("Stack is empty");
        }
        return q1.front();
    }

    bool empty() {
        return q1.empty();
    }
};
```

9. **Implement Deque using Queue**

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

class DequeUsingQueue {
    deque<int> dq;
public:
    void pushFront(int x) {
        dq.push_front(x);
    }
```

```
    void pushBack(int x) {
        dq.push_back(x);
    }

    int popFront() {
        if (dq.empty()) {
            throw runtime_error("Deque is empty");
        }
        int front = dq.front();
        dq.pop_front();
        return front;
    }

    int popBack() {
        if (dq.empty()) {
            throw runtime_error("Deque is empty");
        }
        int back = dq.back();
        dq.pop_back();
        return back;
    }

    bool empty() {
        return dq.empty();
    }
};
```

10. **Implement Circular Queue using Queue**

```
#include <vector>

using namespace std;


class CircularQueue {

   vector<int> queue;

   int front, rear, size, capacity;

public:

   CircularQueue(int k): capacity(k), front(-1), rear(-1), size(0), queue(k) {}


   bool enqueue(int value) {

      if (size == capacity) return false; // Queue is full

      if (front == -1) front = 0; // Initialize front

      rear = (rear + 1) % capacity;

      queue[rear] = value;

      size++;

      return true;

   }


   bool dequeue() {

      if (size == 0) return false; // Queue is empty

      front = (front + 1) % capacity;

      size--;

      return true;

   }


   int getFront() {

      if (size == 0) throw runtime_error("Queue is empty");

      return queue[front];

   }
```

```cpp
    int getRear() {
        if (size == 0) throw runtime_error("Queue is empty");
        return queue[rear];
    }

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

    bool isFull() {
        return size == capacity;
    }
};
```

Array Implementation :

13. **Implement Stack using an Array**

```cpp
class Solution {
    int *arr;
    int top;
    int capacity;

public:
    Solution(int size) {
        arr = new int[size];
        capacity = size;
        top = -1;
    }
```

```
    void push(int x) {
        if (top == capacity - 1) {
            throw runtime_error("Stack Overflow");
        }
        arr[++top] = x;
    }

    int pop() {
        if (top == -1) {
            throw runtime_error("Stack Underflow");
        }
        return arr[top--];
    }

    int peek() {
        if (top == -1) {
            throw runtime_error("Stack is empty");
        }
        return arr[top];
    }

    bool isEmpty() {
        return top == -1;
    }
};
```

14. **Implement Queue using an Array**

```
class Solution {
    int *arr;
    int front, rear, capacity;

public:
    Solution(int size) {
        arr = new int[size];
        capacity = size;
        front = rear = -1;
    }

    void enqueue(int x) {
        if (rear == capacity - 1) {
            throw runtime_error("Queue Overflow");
        }
        if (front == -1) front = 0;
        arr[++rear] = x;
    }

    int dequeue() {
        if (front == -1 || front > rear) {
            throw runtime_error("Queue Underflow");
        }
        return arr[front++];
    }

    bool isEmpty() {
        return front == -1 || front > rear;
```

```
    }
};
```

15. **Implement Circular Queue using an Array**

```
class Solution {
    int *arr;
    int front, rear, size, capacity;

public:
    Solution(int k) {
        capacity = k;
        arr = new int[k];
        front = rear = -1;
        size = 0;
    }

    bool enqueue(int value) {
        if (size == capacity) return false; // Queue is full
        if (front == -1) front = 0; // Initialize front
        rear = (rear + 1) % capacity;
        arr[rear] = value;
        size++;
        return true;
    }

    int dequeue() {
        if (size == 0) throw runtime_error("Queue Underflow");
        int val = arr[front];
        front = (front + 1) % capacity;
```

```
        size--;
        return val;
    }

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

16. **Implement Deque using an Array**

```
class Solution {
    int *arr;
    int front, rear, capacity, size;

public:
    Solution(int k) {
        capacity = k;
        arr = new int[k];
        front = -1;
        rear = 0;
        size = 0;
    }

    void pushFront(int x) {
        if (size == capacity) throw runtime_error("Deque Overflow");
        if (front == -1) { // Initialize
            front = rear = 0;
            arr[front] = x;
        } else {
```

```
        front = (front - 1 + capacity) % capacity;
        arr[front] = x;
    }
    size++;
}

void pushBack(int x) {
    if (size == capacity) throw runtime_error("Deque Overflow");
    if (rear == -1) { // Initialize
        rear = front = 0;
        arr[rear] = x;
    } else {
        rear = (rear + 1) % capacity;
        arr[rear] = x;
    }
    size++;
}

int popFront() {
    if (size == 0) throw runtime_error("Deque Underflow");
    int val = arr[front];
    front = (front + 1) % capacity;
    size--;
    return val;
}

int popBack() {
    if (size == 0) throw runtime_error("Deque Underflow");
    int val = arr[rear];
```

```
        rear = (rear - 1 + capacity) % capacity;
        size--;
        return val;
    }
};
```

17. **Implement Two Stacks in One Array**

```cpp
class Solution {
    int *arr;
    int top1, top2, capacity;

public:
    Solution(int size) {
        capacity = size;
        arr = new int[size];
        top1 = -1; // Stack 1 starts from the left
        top2 = size; // Stack 2 starts from the right
    }

    void push1(int x) {
        if (top1 + 1 == top2) throw runtime_error("Stack Overflow");
        arr[++top1] = x;
    }

    void push2(int x) {
        if (top1 + 1 == top2) throw runtime_error("Stack Overflow");
        arr[--top2] = x;
    }
```

```
    int pop1() {
        if (top1 == -1) throw runtime_error("Stack Underflow");
        return arr[top1--];
    }

    int pop2() {
        if (top2 == capacity) throw runtime_error("Stack Underflow");
        return arr[top2++];
    }
};
```

LinkList Implementation :

25. **Implement Stack using Linked List**

```
class Solution {
    struct Node {
        int data;
        Node* next;
        Node(int x) : data(x), next(nullptr) {}
    };
    Node* top;

public:
    Solution() {
        top = nullptr;
    }

    void push(int x) {
```

```
        Node* newNode = new Node(x);
        newNode->next = top;
        top = newNode;
    }

    int pop() {
        if (!top) {
            throw runtime_error("Stack Underflow");
        }
        int data = top->data;
        Node* temp = top;
        top = top->next;
        delete temp;
        return data;
    }

    int peek() {
        if (!top) {
            throw runtime_error("Stack is empty");
        }
        return top->data;
    }

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

26. **Implement Queue using Linked List**

```
class Solution {
   struct Node {
      int data;
      Node* next;
      Node(int x) : data(x), next(nullptr) {}
   };
   Node *front, *rear;

public:
   Solution() {
      front = rear = nullptr;
   }

   void enqueue(int x) {
      Node* newNode = new Node(x);
      if (!rear) {
         front = rear = newNode;
         return;
      }
      rear->next = newNode;
      rear = newNode;
   }

   int dequeue() {
      if (!front) {
         throw runtime_error("Queue Underflow");
      }
      int data = front->data;
```

```cpp
        Node* temp = front;
        front = front->next;
        if (!front) rear = nullptr;
        delete temp;
        return data;
    }

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

27. **Implement Deque using Doubly Linked List**

```cpp
class Solution {
    struct Node {
        int data;
        Node *prev, *next;
        Node(int x) : data(x), prev(nullptr), next(nullptr) {}
    };
    Node *front, *rear;

public:
    Solution() {
        front = rear = nullptr;
    }

    void pushFront(int x) {
        Node* newNode = new Node(x);
```

```
    if (!front) {
        front = rear = newNode;
        return;
    }
    newNode->next = front;
    front->prev = newNode;
    front = newNode;
}

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

int popFront() {
    if (!front) {
        throw runtime_error("Deque Underflow");
    }
    int data = front->data;
    Node* temp = front;
    front = front->next;
    if (front) front->prev = nullptr;
    else rear = nullptr;
```

```
        delete temp;
        return data;
    }

    int popBack() {
        if (!rear) {
            throw runtime_error("Deque Underflow");
        }
        int data = rear->data;
        Node* temp = rear;
        rear = rear->prev;
        if (rear) rear->next = nullptr;
        else front = nullptr;
        delete temp;
        return data;
    }

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

Tree Implementation :

38. **Implement BST (Binary Search Tree) using Linked List**

```cpp
class Solution {
    struct TreeNode {
        int val;
        TreeNode* left;
        TreeNode* right;
        TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    };
    TreeNode* root;

public:
    Solution() {
        root = nullptr;
    }

    TreeNode* insert(TreeNode* node, int value) {
        if (!node) return new TreeNode(value);
        if (value < node->val)
            node->left = insert(node->left, value);
        else
            node->right = insert(node->right, value);
        return node;
    }

    void insert(int value) {
        root = insert(root, value);
```

```
    }

    bool search(TreeNode* node, int value) {
        if (!node) return false;
        if (node->val == value) return true;
        if (value < node->val)
            return search(node->left, value);
        else
            return search(node->right, value);
    }

    bool search(int value) {
        return search(root, value);
    }
};
```

39. **Implement AVL Tree using BST**

```cpp
class Solution {
    struct TreeNode {
        int val;
        TreeNode* left;
        TreeNode* right;
        int height;
        TreeNode(int x) : val(x), left(nullptr), right(nullptr), height(1) {}
    };
    TreeNode* root;

public:
```

```
Solution() {
    root = nullptr;
}

int height(TreeNode* node) {
    return node ? node->height : 0;
}

int getBalance(TreeNode* node) {
    return node ? height(node->left) - height(node->right) : 0;
}

TreeNode* rotateRight(TreeNode* y) {
    TreeNode* x = y->left;
    TreeNode* T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    return x;
}

TreeNode* rotateLeft(TreeNode* x) {
    TreeNode* y = x->right;
    TreeNode* T2 = y->left;
```

```
    y->left = x;
    x->right = T2;

    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    return y;
}

TreeNode* insert(TreeNode* node, int val) {
    if (!node) return new TreeNode(val);

    if (val < node->val)
        node->left = insert(node->left, val);
    else if (val > node->val)
        node->right = insert(node->right, val);
    else
        return node;

    node->height = 1 + max(height(node->left), height(node->right));

    int balance = getBalance(node);

    // Left Left Case
    if (balance > 1 && val < node->left->val)
        return rotateRight(node);

    // Right Right Case
    if (balance < -1 && val > node->right->val)
```

```
            return rotateLeft(node);

        // Left Right Case
        if (balance > 1 && val > node->left->val) {
            node->left = rotateLeft(node->left);
            return rotateRight(node);
        }

        // Right Left Case
        if (balance < -1 && val < node->right->val) {
            node->right = rotateRight(node->right);
            return rotateLeft(node);
        }

        return node;
    }

    void insert(int val) {
        root = insert(root, val);
    }
};
```

40. **Implement Trie using HashMap**

```cpp
#include <unordered_map>
using namespace std;

class Solution {
    struct TrieNode {
```

```
    unordered_map<char, TrieNode*> children;
    bool isEndOfWord;
    TrieNode() : isEndOfWord(false) {}
  };
  TrieNode* root;

public:
  Solution() {
    root = new TrieNode();
  }

  void insert(string word) {
    TrieNode* current = root;
    for (char c : word) {
      if (current->children.find(c) == current->children.end()) {
        current->children[c] = new TrieNode();
      }
      current = current->children[c];
    }
    current->isEndOfWord = true;
  }

  bool search(string word) {
    TrieNode* current = root;
    for (char c : word) {
      if (current->children.find(c) == current->children.end()) {
        return false;
      }
      current = current->children[c];
```

```
        }
        return current && current->isEndOfWord;
    }

    bool startsWith(string prefix) {
        TrieNode* current = root;
        for (char c : prefix) {
            if (current->children.find(c) == current->children.end()) {
                return false;
            }
            current = current->children[c];
        }
        return true;
    }
};
```