Stack-Based Implementations: -

1. Implement Queue using Stack:

```
class MyQueue {
stack<int> inputStack ,outputStack;
void transfer(){
    if(outputStack.empty()){
        while(!inputStack.empty()){
            outputStack.push(inputStack.top());
            inputStack.pop();}}}
public:
MyQueue() {}
    void push(int x) {
        inputStack.push(x);}

    int pop() {
        transfer();
        int topElement = outputStack.top();
        outputStack.pop();
        return topElement; }

    int peek() {
        transfer();
        return outputStack.top();}

    bool empty() {
        return inputStack.empty()&& outputStack.empty();}};
```

2. Implement Deque using Stack

```
class MyCircularDeque {
private:
    stack<int> stack1, stack2;
    int capacity;
    int size;
    void transferStack(stack<int> &from, stack<int> &to) {
        while (!from.empty()) {
            to.push(from.top());
            from.pop();} }
public:
    MyCircularDeque(int k) {
        capacity = k;
        size = 0;}

    bool insertFront(int value) {
        if (isFull()) return false;
        stack1.push(value);
        size++;
        return true;
    }

    bool insertLast(int value) {
        if (isFull()) return false;
        stack2.push(value);
        size++;
```

```
    bool deleteFront() {
        if (isEmpty()) return false;
        if (stack1.empty()) transferStack(stack2, stack1);
        if (!stack1.empty()) {
            stack1.pop();
            size--;
        }
        return true;
    }

    bool deleteLast() {
        if (isEmpty()) return false;
        if (stack2.empty()) transferStack(stack1, stack2);
        if (!stack2.empty()) {
            stack2.pop();
            size--;
        }
        return true;
    }

    int getFront() {
        if (isEmpty()) return -1;
        if (stack1.empty()) transferStack(stack2, stack1);
        return stack1.empty() ? -1 : stack1.top();
    }
```

```
    int getFront() {
        if (isEmpty()) return -1;
        if (stack1.empty()) transferStack(stack2, stack1);
        return stack1.empty() ? -1 : stack1.top();
    }

    int getRear() {
        if (isEmpty()) return -1;
        if (stack2.empty()) transferStack(stack1, stack2);
        return stack2.empty() ? -1 : stack2.top();
    }

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

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

3. Implement Min Stack using Two Stacks

```
class MinStack {
    stack<int> main , minS;
public:
    MinStack() { }

    void push(int val) {
        main.push(val);
        if(minS.empty()|| val<=minS.top()) minS.push(val);
    }

    void pop() {
        int popValue = main.top();
        if(popValue==minS.top()) minS.pop();
        main.pop();
    }

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

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

4.Implement Max Stack using Two Stacks

```
class MaxStack {
    stack<int> main , maxS;
public:
MaxStack() {}
    void push(int val) {
        main.push(val);
        if(maxS.empty()|| val>=maxS.top()) maxS.push(val);
    }
    void pop() {
        if (main.empty()) {
            cout << "Stack is empty! Cannot pop.\n";
            return;
        }
        int popValue = main.top();
        if(popValue==maxS.top()) maxS.pop();
        main.pop(); }
    int top() {
        if (main.empty()) {
            cout << "Stack is empty! No top element.\n";
            return -1; }
        return main.top();}

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

5.Implement Priority Queue using Stack

```
class priorityQueue{
    stack<int>main , temp;
public:
void push(int val){
    while (!main.empty()&&main.top()>val)
    { temp.push(main.top());
        main.pop();}
    main.push(val);
    while (!temp.empty()) {
        main.push(temp.top());
        temp.pop();
    }}
void pop(){
    if(main.empty()){
        cout<<"priority queue is empty"<<endl;
        return;}
        main.pop();}
int top(){
    if(main.empty()){
        cout<<"priority queue is empty"<<endl;
        return -1;
    } return main.top();
}
bool isEmpty() {
    return main.empty();
}
};
```

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

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int>result;
        stack<TreeNode*>s;
        TreeNode* curr = root;
        while(curr!=NULL||!s.empty()){
            while(curr!=NULL){
                s.push(curr);
            curr=curr->left;
            }
           curr = s.top();
        s.pop();
        result.push_back(curr->val);
        curr = curr->right;
        }
        return result;
    }
};
```

7. Implement Graph DFS using Stack (Iterative DFS)

```
class Graph{
int v;
vector<vector<int>> adj;
public:
Graph(int v){
    this->v=v;
    adj.resize(v);
}
void addEdge(int u , int v){
    adj[u].push_back(v);
    adj[v].push_back(u);
}
void DFS(int start){
    vector<bool> visited(v,false);
    stack<int> s;
    s.push(start);
    while(!s.empty()){
        int node = s.top();
        s.pop();
        if(!visited[node]){
            cout<<node<<" ";
            visited[node]=true; }
        for(int neighbour : adj[node]){
            if(!visited[neighbour]){
                s.push(neighbour);
            } }}}};
```