Queue-Based Implementations: -

8. Implement Stack using Queue

```
class MyStack {
    queue<int> main ,q;
public: MyStack() {}
    void push(int x) {
        main.push(x);}
    int pop() {
        if(main.empty())return -1;
        while(main.size()>1){
            q.push(main.front());
            main.pop(); }
        int topElement = main.front();
        main.pop();
        swap(main,q);
        return topElement;}
    int top() {
        if(main.empty())return -1;
        while(main.size()>1){
            q.push(main.front());
            main.pop();
        }
        int topElement = main.front();
        q.push(main.front());
        main.pop();
        swap(main,q);
        return topElement; }
```

9. Implement Deque using Queue

```
class MyCircularDeque {
    queue<int> q1, q2;
    int capacity, sizee;
public:
    MyCircularDeque(int k) {
        capacity = k;
        sizee = 0;
    }
    bool insertFront(int value) {
        if (isFull()) return false;
        q2.push(value);
        while (!q1.empty()) {
            q2.push(q1.front());
            q1.pop();
        }
        swap(q1, q2);
        sizee++;
        return true;
    }
    bool insertLast(int value) {
        if (isFull()) return false;
        q1.push(value);
        sizee++;
        return true;
    }
```

```
    bool deleteFront() {
        if (isEmpty()) return false;
        q1.pop();
        sizee--;
        return true;
    }

    bool deleteLast() {
        if (isEmpty()) return false;
        while (q1.size() > 1) {
            q2.push(q1.front());
            q1.pop();
        }
        q1.pop();
        swap(q1, q2);
        sizee--;
        return true;
    }

    int getFront() {
        if (isEmpty()) return -1;
        return q1.front();
    }
```

```
int getRear() {
    if (isEmpty()) return -1;
    int lastElement;
    queue<int> temp = q1;
    while (!temp.empty()) {
        lastElement = temp.front();
        temp.pop();
    }
    return lastElement;
}

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

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

10. Implement circular queue using queue.

```
class MyCircularQueue {
    queue<int> q;
    int  capacity;
public:
    MyCircularQueue(int k) {
        capacity = k; }

    bool enQueue(int value) {
        if (q.size() >= capacity) return false;
        q.push(value);
        return true; }

    bool deQueue() {
    if (q.empty()) return false;
        q.pop();
        return true; }

    int Front() {return q.empty() ? -1 : q.front(); }

    int Rear() {return q.empty() ? -1 : q.back();}

    bool isEmpty() { return q.empty();}

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

11. Implement BST Level Order Traversal using Queue (BFS)

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
       vector<vector<int>>result;
        if(!root)return result;
        queue<TreeNode*>BFSQueue;
        BFSQueue.push(root);
while(!BFSQueue.empty()){
    int levelSize = BFSQueue.size();
    vector<int>ans;
    for(int i =0;i<levelSize;i++){
        TreeNode*temp = BFSQueue.front();
    BFSQueue.pop();
    ans.push_back(temp->val);
    if(temp->left)BFSQueue.push(temp->left);
     if(temp->right)BFSQueue.push(temp->right);
    }

   result.push_back(ans);
} return result;
    }
};
```

12. Implement Graph BFS using Queue:

```
class Graph {
    int V;
    vector<vector<int>> adj;
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);
    }
    void BFS(int start) {
        vector<bool> visited(V, false);
        queue<int> q;
        visited[start] = true;
        q.push(start);
        cout << "BFS Traversal: ";
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            cout << node << " ";
            for (int neighbor : adj[node]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }}}cout << endl; }};
```