Data Structure projects

NAME: Youssef Mohamed ali

Id:200033377

Q(1): Write C++ program to implement all features of Priority Queue

Code:

Output:

```cpp
#include <iostream>
#include <vector>

using namespace std;

class PriorityQueue {
private:
    vector<int> heap;
    void heapifyUp(int index) {
        while (index > 0) {
            int parent = (index - 1) / 2;
            if (heap[index] > heap[parent]) {
                swap(heap[index], heap[parent]);
                index = parent;
            }
            else {
                break;
            }
        }
    }
    void heapifyDown(int index) {
        int size = heap.size();
        while (index < size) {
            int leftChild = 2 * index + 1;
            int rightChild = 2 * index + 2;
            int largest = index;

            if (leftChild < size && heap[leftChild] > heap[largest])
                largest = leftChild;

            if (rightChild < size && heap[rightChild] > heap[largest])
                largest = rightChild;

            if (largest != index) {
                swap(heap[index], heap[largest]);
                index = largest;
            }
            else {
```

```
                break;
            }
        }
    }

public:
    void push(int val) {
        heap.push_back(val);
        heapifyUp(heap.size() - 1);
    }
    void pop() {
        if (heap.empty()) {
            cout << "Priority queue is empty.\n";
            return;
        }
        heap[0] = heap.back();
        heap.pop_back();
        heapifyDown(0);
    }
    int top() {
        if (heap.empty()) {
            cout << "Priority queue is empty.\n";
            return -1;
        }
        return heap[0];
    }
    bool empty() {
        return heap.empty();
    }
};

int main() {
    PriorityQueue pq;
    pq.push(5);
    pq.push(10);
    pq.push(3);
    pq.push(7);

    cout << "Current maximum element: " << pq.top() << endl;

    pq.pop();

    cout << "After popping, new maximum element: " << pq.top() << endl;

    return 0;
}
```

Current maximum priority element: 10

After popping, new maximum priority element: 7

Answer:

Priority queue is a FiFo data structure that serves the highest priority 1st.

Q(2): Write C++ program to implement all features of Double Ended Queue

CODE:

```
      #include <iostream>

using namespace std;

struct Node {
    int data;
    Node* next;
    Node* prev;
    Node(int val) : data(val), next(nullptr), prev(nullptr) {}
};

class Deque {
private:
    Node* head;
    Node* tail;

public:

    Deque() : head(nullptr), tail(nullptr) {}


    ~Deque() {
        while (head) {
            Node* temp = head;
            head = head->next;
            delete temp;
        }
    }
    bool empty() {
        return head == nullptr;
    }
    void push_front(int val) {
        Node* newNode = new Node(val);
        if (empty()) {
            head = tail = newNode;
        }
        else {
            newNode->next = head;
            head->prev = newNode;
```

```
            head = newNode;
        }
    }
    void push_back(int val) {
        Node* newNode = new Node(val);
        if (empty()) {
            head = tail = newNode;
        }
        else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }
    int pop_front() {
        if (empty()) {
            cout << "Deque is empty.\n";
            return -1;
        }
        int frontVal = head->data;
        Node* temp = head;
        if (head == tail) {
            head = tail = nullptr;
        }
        else {
            head = head->next;
            head->prev = nullptr;
        }
        delete temp;
        return frontVal;
    }
    int pop_back() {
        if (empty()) {
            cout << "Deque is empty.\n";
            return -1;
        }
        int backVal = tail->data;
        Node* temp = tail;
        if (head == tail) {
            head = tail = nullptr;
        }
        else {
            tail = tail->prev;
            tail->next = nullptr;
        }
        delete temp;
        return backVal;
    }
    int front() {
        if (empty()) {
            cout << "Deque is empty.\n";
            return -1;
        }
        return head->data;
    }
    int back() {
        if (empty()) {
            cout << "Deque is empty.\n";
            return -1;
        }
        return tail->data;
    }
};
```

```
int main() {
    Deque dq;
    dq.push_back(5);
    dq.push_front(3);
    dq.push_back(7);

    cout << "Front element: " << dq.front() << endl;
    cout << "Back element: " << dq.back() << endl;

    dq.pop_front();
    dq.pop_back();

    cout << "Front element after popping: " << dq.front() << endl;
    cout << "Back element after popping: " << dq.back() << endl;

    return 0;
}
```

Output:

Front element: 3

Back element: 7

Front element after popping: 5

Back element after popping: 5

Because we popped means we deleted front and back now we only have 5 so the front and the back will point at 5.

Q(3): Write C++ program to implement all features of Double Ended Queue

code:

```
#include <iostream>
#define MAX_SIZE 100

using namespace std;

class Deque {
private:
    int arr[MAX_SIZE];
    int frontIndex;
    int backIndex;

public:

    Deque() : frontIndex(-1), backIndex(-1) {
}
    bool empty() {
        return frontIndex == -1;
    }
    bool full() {
        return (frontIndex == 0 && backIndex == MAX_SIZE - 1) || (frontIndex ==
backIndex + 1);
    }
    void push_front(int val) {
        if (full()) {
            cout << "Deque is full.\n";
            return;
        }
        if (empty()) {
            frontIndex = backIndex = 0;
        }
        else if (frontIndex == 0) {
            frontIndex = MAX_SIZE - 1;
        }
        else {
            frontIndex--;
        }
        arr[frontIndex] = val;
    }
    void push_back(int val) {
        if (full()) {
            cout << "Deque is full.\n";
            return;
        }
        if (empty()) {
```

```
            frontIndex = backIndex = 0;
        }
        else if (backIndex == MAX_SIZE - 1) {
            backIndex = 0;
        }
        else {
            backIndex++;
        }
        arr[backIndex] = val;
    }
    int pop_front() {
        if (empty()) {
            cout << "Deque is empty.\n";
            return -1;
        }
        int frontVal = arr[frontIndex];
        if (frontIndex == backIndex) {
            frontIndex = backIndex = -1;
        }
        else if (frontIndex == MAX_SIZE - 1) {
            frontIndex = 0;
        }
        else {
            frontIndex++;
        }
        return frontVal;
    }
    int pop_back() {
        if (empty()) {
            cout << "Deque is empty.\n";
            return -1;
        }
        int backVal = arr[backIndex];
        if (frontIndex == backIndex) {
            frontIndex = backIndex = -1;
        }
        else if (backIndex == 0) {
            backIndex = MAX_SIZE - 1;
        }
        else {
            backIndex--;
        }
        return backVal;
    }
    int front() {
        if (empty()) {
            cout << "Deque is empty.\n";
            return -1;
        }
        return arr[frontIndex];
    }
    int back() {
        if (empty()) {
            cout << "Deque is empty.\n";
            return -1;
        }
        return arr[backIndex];
    }
};

int main() {
    Deque dq;
    dq.push_back(5);
```

```
    dq.push_front(3);
    dq.push_back(7);

    cout << "Front element: " << dq.front() << endl;
    cout << "Back element: " << dq.back() << endl;

    dq.pop_front();
    dq.pop_back();

    cout << "Front element after popping: " << dq.front() << endl;
    cout << "Back element after popping: " << dq.back() << endl;

    return 0;
}
```

Output:

Front element: 3

Back element: 7

Front element after popping: 5

Back element after popping: 5

Because we popped means we deleted front and back now we only have 5 so the front and the back will point at 5.

Q(4): Write C++ program to implement all features of Full

Binary Tree

code:

```
#include <iostream>
#include <queue>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class BinaryTree {
private:
    Node* root;

public:

    BinaryTree() : root(nullptr) {}

    ~BinaryTree() {
        deleteTree(root);
    }

    void deleteTree(Node* root) {
        if (root == nullptr) return;
        deleteTree(root->left);
        deleteTree(root->right);
        delete root;
    }

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

    Node* insertUtil(Node* root, int val) {
        if (root == nullptr) {
            root = new Node(val);
        }
        else {
            if (root->left == nullptr)
                root->left = insertUtil(root->left, val);
            else if (root->right == nullptr)
                root->right = insertUtil(root->right, val);
```

```
            else
                root->left = insertUtil(root->left, val);
        }
        return root;
    }


    void inorderTraversal(Node* root) {
        if (root == nullptr) return;
        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
    }


    bool isFullBinaryTree(Node* root) {
        if (root == nullptr) return true;


        if (root->left == nullptr && root->right == nullptr)
            return true;


        if (root->left != nullptr && root->right != nullptr)
            return isFullBinaryTree(root->left) && isFullBinaryTree(root->right);


        return false;
    }

    bool isFullBinaryTree() {
        return isFullBinaryTree(root);
    }
};

int main() {
    BinaryTree tree;

    tree.insert(1);
    tree.insert(2);
    tree.insert(3);
    tree.insert(4);
    tree.insert(5);
    tree.insert(6);

    if (tree.isFullBinaryTree()) {
        cout << "Tree is a full binary tree." << endl;
    }
    else {
        cout << "Tree is not a full binary tree." << endl;
    }

    return 0;
}
```

OUTPUT:

Tree is not a full binary tree.

Q(5): Write C++ program to implement all features of Degenerate Binary Tree

Code:

```cpp
#include <iostream>

using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class DegenerateBinaryTree {
private:
    Node* root;

public:
    DegenerateBinaryTree() : root(nullptr) {}
    ~DegenerateBinaryTree() {
        deleteTree(root);
    }
    void deleteTree(Node* root) {
        if (root == nullptr) return;
        deleteTree(root->left);
        deleteTree(root->right);
        delete root;
    }
    void insert(int val) {
        root = insertUtil(root, val);
    }
    Node* insertUtil(Node* root, int val) {
        if (root == nullptr) {
            root = new Node(val);
        }
        else {
            root->left = insertUtil(root->left, val);
        }
        return root;
    }
    void inorderTraversal(Node* root) {
        if (root == nullptr) return;
        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
```

```
    }
    bool isDegenerateBinaryTree(Node* root) {
        if (root == nullptr) return true;
        if (root->left != nullptr && root->right != nullptr)
            return false;
        return isDegenerateBinaryTree(root->left) && isDegenerateBinaryTree(root-
>right);
    }
    bool isDegenerateBinaryTree() {
        return isDegenerateBinaryTree(root);
    }
};

int main() {
    DegenerateBinaryTree tree;
    tree.insert(1);
    tree.insert(2);
    tree.insert(3);
    tree.insert(4);
    tree.insert(5);
    if (tree.isDegenerateBinaryTree()) {
        cout << "Tree is a degenerate binary tree." << endl;
    }
    else {
        cout << "Tree is not a degenerate binary tree." << endl;
    }

    return 0;
}
```

Output:

Tree is a degenerate binary tree.

Because it only has one child on each right side.

Q(6): Write C++ program to implement all features of Complete Binary Tree

Code:

```
#include <iostream>
#include <queue>

using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class CompleteBinaryTree {
private:
    Node* root;

public:

    CompleteBinaryTree() : root(nullptr) {}
    ~CompleteBinaryTree() {
        deleteTree(root);
    }
    void deleteTree(Node* root) {
        if (root == nullptr) return;
        deleteTree(root->left);
        deleteTree(root->right);
        delete root;
    }
    void insert(int val) {
        if (root == nullptr) {
            root = new Node(val);
            return;
        }
        queue<Node*> q;
        q.push(root);
        while (!q.empty()) {
            Node* temp = q.front();
            q.pop();
            if (temp->left == nullptr) {
                temp->left = new Node(val);
                break;
            }
            else {
                q.push(temp->left);
            }
            if (temp->right == nullptr) {
```

```
                temp->right = new Node(val);
                break;
            }
            else {
                q.push(temp->right);
            }
        }
    }


    void inorderTraversal(Node* root) {
        if (root == nullptr) return;
        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
    }


    bool isCompleteBinaryTree() {
        if (root == nullptr) return true;


        queue<Node*> q;
        q.push(root);


        bool nonFullNode = false;

        while (!q.empty()) {
            Node* temp = q.front();
            q.pop();


            if (temp->left == nullptr && temp->right != nullptr)
                return false;


            if (temp->left == nullptr && temp->right == nullptr)
                nonFullNode = true;


            if (nonFullNode && (temp->left != nullptr || temp->right != nullptr))
                return false;

            if (temp->left != nullptr)
                q.push(temp->left);
            if (temp->right != nullptr)
                q.push(temp->right);
        }

        return true;
    }
};

int main() {
    CompleteBinaryTree tree;
    tree.insert(1);
    tree.insert(2);
    tree.insert(3);
    tree.insert(4);
    tree.insert(5);
    tree.insert(6);
    if (tree.isCompleteBinaryTree()) {
```

```
        cout << "Tree is a complete binary tree." << endl;
    }
    else {
        cout << "Tree is not a complete binary tree." << endl;
    }

    return 0;
}
```

Output :

Tree is a complete binary tree

Because At each level, all nodes are filled from left to right.

If we encounter a node that has only a right child or a node with no children after encountering a node with only one child, the tree
cannot be complete.

Q(7): Write C++ program to implement all features of Perfect Binary Tree

Code:

```
#include <iostream>
#include <queue>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int value) : data(value), left(nullptr), right(nullptr) {}
};
Node* newNode(int value) {
    return new Node(value);
}
void levelOrderTraversal(Node* root) {
    if (root == nullptr)
        return;

    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        Node* current = q.front();
        cout << current->data << " ";
        q.pop();

        if (current->left)
            q.push(current->left);
        if (current->right)
            q.push(current->right);
    }
}

int main() {
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    cout << "Level Order Traversal of the perfect binary tree: ";
    levelOrderTraversal(root);
```

```
    return 0;
}
```

Output: Level Order Traversal of the perfect binary tree: 1 2 3 4 5 6 7

Because Every node has either 0 or 2 children also All leaf nodes are at the same level and The number of nodes at each level doubles as you move down the tree from the root.

Q(8): Write C++ program to implement all features of Balanced Binary Tree

Code:

```
#include <iostream>
using namespace std;
struct Node {
    int key;
    Node* left;
    Node* right;
    int height;
};
Node* newNode(int key) {
    Node* node = new Node;
    node->key = key;
    node->left = nullptr;
    node->right = nullptr;
    node->height = 1;
    return node;
}
int height(Node* node) {
    if (node == nullptr)
        return 0;
    return node->height;
}
int max(int a, int b) {
    return (a > b) ? a : b;
}
```

```
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* 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;
}
Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* 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;
}
int getBalance(Node* node) {
    if (node == nullptr)
        return 0;
    return height(node->left) - height(node->right);
}
Node* insert(Node* node, int key) {
    if (node == nullptr)
        return newNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else
        return node;
    node->height = 1 + max(height(node->left), height(node->right));
    int balance = getBalance(node);
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}
Node* search(Node* root, int key) {
    if (root == nullptr || root->key == key)
        return root;

    if (root->key < key)
        return search(root->right, key);

    return search(root->left, key);
}
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
```

```
    }
}

int main() {
    Node* root = nullptr;
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    cout << "Inorder traversal of the AVL tree: ";
    inorder(root);
    cout << endl;
    int keyToSearch = 30;
    Node* foundNode = search(root, keyToSearch);
    if (foundNode)
        cout << "Key " << keyToSearch << " found in the AVL tree." << endl;
    else
        cout << "Key " << keyToSearch << " not found in the AVL tree." << endl;

    return 0;
}
```

Output:

Inorder traversal of the AVL tree: 10 20 25 30 40 50

Key 30 found in the AVL tree.

There are 3 types of balanced binary tree

1-AVl tree

2-Red Black tree

3-B tree

Q(9): Write C++ program to implement all features of AVL Tree

CODE:

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

struct Node {
    int key;
    Node* left;
    Node* right;
    int height;
};

int max(int a, int b) {
    return (a > b) ? a : b;
}

int height(Node* node) {
    if (node == nullptr)
        return 0;
    return node->height;
}

Node* newNode(int key) {
    Node* node = new Node;
    node->key = key;
    node->left = nullptr;
    node->right = nullptr;
    node->height = 1;
    return node;
}

Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* 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;
}

Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* 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;
}

int getBalance(Node* node) {
    if (node == nullptr)
        return 0;
    return height(node->left) - height(node->right);
}

Node* insert(Node* node, int key) {
    if (node == nullptr)
        return newNode(key);

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

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

    int balance = getBalance(node);

    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
}
```

```
int main() {
    Node* root = nullptr;
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);
    cout << "Inorder traversal of the AVL tree: ";
    inorder(root);
    cout << endl;

    return 0;
}
```

Output:

Inorder traversal of the AVL tree:

10 20 25 30 40 50

Because we reordered the numbers to make it more balance.

Q(10): Write C++ program to implement all features of Red-Black Tree

Code:

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

enum Color { RED, BLACK };

struct Node {
```

```
    int key;
    Color color;
    Node* left;
    Node* right;
    Node* parent;

    Node(int key) : key(key), color(RED), left(nullptr), right(nullptr),
parent(nullptr) {}
};

class RedBlackTree {
private:
    Node* root;

    void rotateLeft(Node* x) {
        Node* y = x->right;
        x->right = y->left;
        if (y->left != nullptr)
            y->left->parent = x;
        y->parent = x->parent;
        if (x->parent == nullptr)
            root = y;
        else if (x == x->parent->left)
            x->parent->left = y;
        else
            x->parent->right = y;
        y->left = x;
        x->parent = y;
    }

    void rotateRight(Node* y) {
        Node* x = y->left;
        y->left = x->right;
        if (x->right != nullptr)
            x->right->parent = y;
        x->parent = y->parent;
        if (y->parent == nullptr)
            root = x;
        else if (y == y->parent->left)
            y->parent->left = x;
        else
            y->parent->right = x;
        x->right = y;
        y->parent = x;
    }

    void fixViolation(Node* z) {
        while (z != root && z->parent->color == RED) {
            if (z->parent == z->parent->parent->left) {
                Node* y = z->parent->parent->right;
                if (y != nullptr && y->color == RED) {
                    z->parent->color = BLACK;
                    y->color = BLACK;
                    z->parent->parent->color = RED;
                    z = z->parent->parent;
                }
                else {
                    if (z == z->parent->right) {
                        z = z->parent;
                        rotateLeft(z);
                    }
                    z->parent->color = BLACK;
                    z->parent->parent->color = RED;
```

```
                    rotateRight(z->parent->parent);
                }
            }
            else {
                Node* y = z->parent->parent->left;
                if (y != nullptr && y->color == RED) {
                    z->parent->color = BLACK;
                    y->color = BLACK;
                    z->parent->parent->color = RED;
                    z = z->parent->parent;
                }
                else {
                    if (z == z->parent->left) {
                        z = z->parent;
                        rotateRight(z);
                    }
                    z->parent->color = BLACK;
                    z->parent->parent->color = RED;
                    rotateLeft(z->parent->parent);
                }
            }
        }
        root->color = BLACK;
    }

    void inorderHelper(Node* root) {
        if (root == nullptr)
            return;
        inorderHelper(root->left);
        cout << root->key << " ";
        inorderHelper(root->right);
    }

public:
    RedBlackTree() : root(nullptr) {}

    void insert(int key) {
        Node* newNode = new Node(key);
        Node* parent = nullptr;
        Node* current = root;

        while (current != nullptr) {
            parent = current;
            if (newNode->key < current->key)
                current = current->left;
            else
                current = current->right;
        }

        newNode->parent = parent;
        if (parent == nullptr)
            root = newNode;
        else if (newNode->key < parent->key)
            parent->left = newNode;
        else
            parent->right = newNode;

        fixViolation(newNode);
    }

    void inorder() {
        inorderHelper(root);
    }
```

```
};

int main() {
    RedBlackTree rbTree;
    rbTree.insert(10);
    rbTree.insert(20);
    rbTree.insert(30);
    rbTree.insert(40);
    rbTree.insert(50);
    rbTree.insert(25);
    cout << "Inorder traversal of the Red-Black tree: ";
    rbTree.inorder();
    cout << endl;

    return 0;
}
```

Output:

Inorder traversal of the Red-Black tree: 10 20 25 30 40 50

Q(11): Write C++ program to implement all features of B-tree

Code:

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

const int MAX_KEYS = 3;

struct Node {
    vector<int> keys;
    vector<Node*> children;

    Node() {}
};

class BTree {
private:
    Node* root;

    Node* insertIntoNode(Node* node, int key) {
        if (node->keys.size() < MAX_KEYS) {
            node->keys.push_back(key);
            sort(node->keys.begin(), node->keys.end());
            return nullptr;
        }
```

```
        else {
            Node* newNode = new Node();
            newNode->keys.push_back(node->keys[MAX_KEYS / 2]);
            for (int i = MAX_KEYS / 2 + 1; i < MAX_KEYS; ++i)
                newNode->keys.push_back(node->keys[i]);
            for (int i = MAX_KEYS / 2 + 1; i <= MAX_KEYS; ++i)
                newNode->children.push_back(node->children[i]);
            node->keys.resize(MAX_KEYS / 2);
            node->children.resize(MAX_KEYS / 2 + 1);
            if (key < newNode->keys.front())
                insertIntoNode(node, key);
            else
                insertIntoNode(newNode, key);
            return newNode;
        }
    }

    Node* insert(Node* node, int key) {
        if (node == nullptr) {
            node = new Node();
            node->keys.push_back(key);
            return nullptr;
        }

        if (node->children.empty())
            return insertIntoNode(node, key);
        else {
            int i = 0;
            while (i < node->keys.size() && key > node->keys[i])
                ++i;
            Node* child = insert(node->children[i], key);
            if (child != nullptr) {
                node->keys.insert(node->keys.begin() + i, child->keys.front());
                node->children.insert(node->children.begin() + i, child);
                if (node->keys.size() > MAX_KEYS) {
                    Node* newNode = new Node();
                    newNode->keys.push_back(node->keys[MAX_KEYS / 2]);
                    for (int j = MAX_KEYS / 2 + 1; j < MAX_KEYS; ++j)
                        newNode->keys.push_back(node->keys[j]);
                    for (int j = MAX_KEYS / 2 + 1; j <= MAX_KEYS; ++j)
                        newNode->children.push_back(node->children[j]);
                    node->keys.resize(MAX_KEYS / 2);
                    node->children.resize(MAX_KEYS / 2 + 1);
                    return newNode;
                }
            }
            return nullptr;
        }
    }

    bool searchInNode(Node* node, int key) {
        for (int i = 0; i < node->keys.size(); ++i) {
            if (key == node->keys[i])
                return true;
            else if (key < node->keys[i])
                return node->children.empty() ? false : searchInNode(node-
>children[i], key);
        }
        return node->children.empty() ? false : searchInNode(node-
>children.back(), key);
    }

public:
```

```
    BTree() : root(nullptr) {}

    void insert(int key) {
        Node* newNode = insert(root, key);
        if (newNode != nullptr) {
            Node* newRoot = new Node();
            newRoot->keys.push_back(newNode->keys.front());
            newRoot->children.push_back(root);
            newRoot->children.push_back(newNode);
            root = newRoot;
        }
    }

    bool search(int key) {
        return root == nullptr ? false : searchInNode(root, key);
    }
};

int main() {
    BTree btree;

    // Insert some keys
    btree.insert(10);
    btree.insert(20);
    btree.insert(5);
    btree.insert(30);

    // Search for keys
    cout << "Search for key 20: " << (btree.search(20) ? "Found" : "Not found")
<< endl;
    cout << "Search for key 25: " << (btree.search(25) ? "Found" : "Not found")
<< endl;

    return 0;
}
```

Output:

Search for key 20: found

Search for key 25: Not found

Because after inserting the values if we searched at the b-tree we will find 20 but we will not find 25

Q(12): Write C++ program to implement Huffman Code

Code:

```cpp
#include <iostream>
#include <queue>
#include <map>
#include <string>

using namespace std;
struct HuffmanNode {
    char data;
    int freq;
    HuffmanNode* left, * right;
};
struct compare {
    bool operator()(HuffmanNode* l, HuffmanNode* r) {
        return (l->freq > r->freq);
    }
};
HuffmanNode* createNode(char data, int freq, HuffmanNode* left, HuffmanNode*
right) {
    HuffmanNode* node = new HuffmanNode();
    node->data = data;
    node->freq = freq;
    node->left = left;
    node->right = right;
    return node;
}
void encode(HuffmanNode* root, string str, map<char, string>& huffmanCode) {
    if (root == nullptr)
        return;

    if (!root->left && !root->right) {
        huffmanCode[root->data] = str;
    }

    encode(root->left, str + "0", huffmanCode);
    encode(root->right, str + "1", huffmanCode);
}
map<char, string> buildHuffmanTree(string text) {
    map<char, int> freq;
    for (char ch : text) {
        freq[ch]++;
    }

    priority_queue<HuffmanNode*, vector<HuffmanNode*>, compare> pq;

    for (auto it = freq.begin(); it != freq.end(); ++it) {
        pq.push(createNode(it->first, it->second, nullptr, nullptr));
    }

    while (pq.size() != 1) {
        HuffmanNode* left = pq.top(); pq.pop();
        HuffmanNode* right = pq.top(); pq.pop();

        int sum = left->freq + right->freq;
        pq.push(createNode('\0', sum, left, right));
    }

    HuffmanNode* root = pq.top();

    map<char, string> huffmanCode;
    encode(root, "", huffmanCode);

    return huffmanCode;
}
```

```
int main() {
    string text = "Huffman coding is a data compression algorithm.";
    map<char, string> huffmanCode = buildHuffmanTree(text);

    cout << "Huffman Codes:\n";
    for (auto it = huffmanCode.begin(); it != huffmanCode.end(); ++it) {
        cout << it->first << " : " << it->second << "\n";
    }

    return 0; }
```

output:

Huffman Codes:

: 011

. : 111110

H : 111111

a : 010

c : 11001

d : 10111

e : 00100

f : 0011

g : 11011

h : 110000

i : 1110

l : 00101

m : 1001

n : 1000

o : 000

p : 10110

r : 11010

s : 1010

t : 11110

u : 110001

Q13:

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

// Function to create adjacency matrix from a list of edges
void createAdjMatrix(int** adjMatrix, const vector<pair<int, int>>& edges, int vertices) {
    // Initialize the adjacency matrix with 0
    for (int i = 0; i < vertices; ++i) {
        for (int j = 0; j < vertices; ++j) {
            adjMatrix[i][j] = 0;
```

```
        }
    }

    // Populate the adjacency matrix
    for (auto edge : edges) {
        int start = edge.first;
        int end = edge.second;
        adjMatrix[start][end] = 1; // Assuming a directed graph
        // Use the next line instead for an undirected graph
        // adjMatrix[start][end] = adjMatrix[end][start] = 1;
    }
}

// Function to print the adjacency matrix
void printMatrix(int** matrix, int vertices) {
    for (int i = 0; i < vertices; ++i) {
```

```
        for (int j = 0; j < vertices;
++j) {
            cout << matrix[i][j] << "
";
        }
        cout << endl;
    }
}

int main() {
    int vertices = 5; // Number of
vertices in the graph
    vector<pair<int, int>> edges = {
        {0, 1}, {1, 2}, {2, 3}, {3,
4}, {4, 0} // Example edges
    };

    // Create a 2D array for the
adjacency matrix
    int** adjMatrix = new int*
[vertices];
    for (int i = 0; i < vertices; ++i)
{
```

```
        adjMatrix[i] = new
int[vertices];
    }

    // Create adjacency matrix
    createAdjMatrix(adjMatrix, edges,
vertices);

    // Print the adjacency matrix
    cout << "Adjacency Matrix:" <<
endl;
    printMatrix(adjMatrix, vertices);

    // Memory clean up
    for (int i = 0; i < vertices; ++i)
{
        delete[] adjMatrix[i];
    }
    delete[] adjMatrix;

    return 0;
}
```

Q14:

```
#include <iostream>
#include <vector>
#include <list>

using namespace std;

// Function to create an adjacency
list from a list of edges
void createAdjList(vector<list<int>>&
adjList, const vector<pair<int, int>>&
edges) {
    for (auto edge : edges) {
        int start = edge.first;
        int end = edge.second;
        adjList[start].push_back(end);
        // Uncomment the next line for
an undirected graph
        //
adjList[end].push_back(start);
    }
```

```
}


// Function to print the adjacency
list
void printAdjList(const
vector<list<int>>& adjList) {
    for (int i = 0; i <
adjList.size(); i++) {
        cout << "Adjacency list of
vertex " << i << ":\n head";
        for (auto v : adjList[i]) {
            cout << " -> " << v;
        }
        cout << endl;
    }
}


int main() {
    int vertices = 5; // Number of
vertices in the graph
    vector<pair<int, int>> edges = {
```

```
        {0, 1}, {1, 2}, {2, 3}, {3,
4}, {4, 0} // Example edges
    };

    // Create a vector of lists to
represent the adjacency list
    vector<list<int>>
adjList(vertices);

    // Create adjacency list
    createAdjList(adjList, edges);

    // Print the adjacency list
    cout << "The graph as an adjacency
list:" << endl;
    printAdjList(adjList);

    return 0;
}
```

Q15:

```
#include <iostream>
```

```
#include <vector>
#include <queue>
#include <list>

using namespace std;

// Graph class represents a directed graph using adjacency list representation
class Graph {
    int V;    // No. of vertices
    list<int>* adj;    // Pointer to an array containing adjacency lists
public:
    Graph(int V);  // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void BFS(int s);  // prints BFS traversal from a given source s
};

Graph::Graph(int V) {
```

```
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w) {
    adj[v].push_back(w); // Add w to
v's list.
}

void Graph::BFS(int s) {
    // Mark all the vertices as not
visited
    vector<bool> visited(V, false);

    // Create a queue for BFS
    queue<int> queue;

    // Mark the current node as
visited and enqueue it
    visited[s] = true;
    queue.push(s);
```

```
    while (!queue.empty()) {
        // Dequeue a vertex from queue
and print it
        s = queue.front();
        cout << s << " ";
        queue.pop();

        // Get all adjacent vertices
of the dequeued vertex s
        // If a adjacent has not been
visited, then mark it visited and
enqueue it
        for (auto i = adj[s].begin();
i != adj[s].end(); ++i) {
            if (!visited[*i]) {
                visited[*i] = true;
                queue.push(*i);
            }
        }
    }
}
```

```
int main() {
    // Create a graph given in the
above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Breadth
First Traversal (starting from vertex
2) \n";
    g.BFS(2);

    return 0;
}
```

Q16:

```
#include <iostream>
#include <list>
#include <vector>
```

```
using namespace std;


class Graph {
    int V;    // No. of vertices
    list<int>* adj; // Pointer to an
array containing adjacency lists
public:
    Graph(int V);  // Constructor
    void addEdge(int v, int w); //
Function to add an edge to the graph
    void DFS(int v); // DFS traversal
of the vertices reachable from v
    void DFSUtil(int v, vector<bool>&
visited); // Utility function used by
DFS
};


Graph::Graph(int V) {
    this->V = V;
    adj = new list<int>[V];
}
```

```
void Graph::addEdge(int v, int w) {
    adj[v].push_back(w); // Add w to
v’s list.
}


void Graph::DFSUtil(int v,
vector<bool>& visited) {
    // Mark the current node as
visited and print it
    visited[v] = true;
    cout << v << " ";


    // Recur for all the vertices
adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i !=
adj[v].end(); ++i) {
        if (!visited[*i]) {
            DFSUtil(*i, visited);
        }
    }
```

```
}

void Graph::DFS(int v) {
    // Mark all the vertices as not visited
    vector<bool> visited(V, false);

    // Call the recursive helper function to print DFS traversal
    DFSUtil(v, visited);
}

int main() {
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);
```

```
    cout << "Following is Depth First
Traversal (starting from vertex 2)
\n";
    g.DFS(2);

    return 0;
}
```

Q17:

```
#include <iostream>
#include <vector>
#include <list>

using namespace std;

class Graph {
    int V;    // Number of vertices
    list<int>* adj; // Adjacency list
public:
    Graph(int V);  // Constructor
    void addEdge(int v, int w); // To
add an edge
```

```
    void greedyColoring(); // To color
the vertices
};

Graph::Graph(int V) {
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w) {
    adj[v].push_back(w);
    adj[w].push_back(v); // Note: the
graph is undirected
}

void Graph::greedyColoring() {
    vector<int> result(V);
    result[0] = 0; // Assign the first
color to first vertex

    // Initialize remaining V-1
vertices as unassigned
```

```
    for (int u = 1; u < V; u++)
        result[u] = -1;  // no color
is assigned to u


    // A temporary array to store the
available colors. False value of
    // available[cr] would mean that
the color cr is assigned to one of its
adjacent vertices
    vector<bool> available(V, false);


    // Assign colors to remaining V-1
vertices
    for (int u = 1; u < V; u++) {
        // Process all adjacent
vertices and flag their colors as
unavailable
        list<int>::iterator i;
        for (i = adj[u].begin(); i !=
adj[u].end(); ++i)
            if (result[*i] != -1)
                available[result[*i]]
= true;
```

```
        // Find the first available
color
        int cr;
        for (cr = 0; cr < V; cr++)
            if (available[cr] ==
false)
                break;

        result[u] = cr; // Assign the
found color

        // Reset the values back to
false for the next iteration
        for (i = adj[u].begin(); i !=
adj[u].end(); ++i)
            if (result[*i] != -1)
                available[result[*i]]
= false;
    }

    // Print the result
```

```
    for (int u = 0; u < V; u++)
        cout << "Vertex " << u << " --
-> Color " << result[u] << endl;
}

int main() {
    Graph g(5);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(1, 3);
    g.addEdge(2, 3);
    g.addEdge(3, 4);
    cout << "Coloring of graph 1 \n";
    g.greedyColoring();

    return 0;
}
```

Q18:

```
#include <iostream>
#include <stack>
#include <string>

using namespace std;

// Function to return precedence of
operators
int precedence(char op) {
    if (op == '+' || op == '-') {
        return 1;
    }
    if (op == '*' || op == '/') {
        return 2;
    }
    if (op == '^') {
        return 3;
    }
    return 0;
}
```

```
// Function to perform an operation
and return output
bool isOperator(char c) {
    return (!isalpha(c) &&
!isdigit(c));
}

// Function to check if the given
character is an operand
bool isOperand(char c) {
    return isalpha(c) || isdigit(c);
}

// Function to convert infix
expression to postfix
string infixToPostfix(string infix) {
    stack<char> st;
    string postfix = "";

    for (int i = 0; i <
infix.length(); i++) {
        char c = infix[i];
```

```
        // If the scanned character is
an operand, add it to output
        if (isOperand(c)) {
            postfix += c;
        }
        // If the scanned character is
'(', push it to the stack
        else if (c == '(') {
            st.push('(');
        }
        // If the scanned character is
')', pop and output from the stack
        // until an '(' is encountered
        else if (c == ')') {
            while (!st.empty() &&
st.top() != '(') {
                postfix += st.top();
                st.pop();
            }
            st.pop(); // remove '('
from the stack
```

```
        }
        // An operator is encountered
        else {
            while (!st.empty() &&
precedence(c) <= precedence(st.top()))
{
                postfix += st.top();
                st.pop();
            }
            st.push(c);
        }
    }

    // Pop all the remaining elements
from the stack
    while (!st.empty()) {
        postfix += st.top();
        st.pop();
    }

    return postfix;
}
```

```
int main() {
    string infix = "a+b*(c^d-e)^(f+g*h)-i";
    cout << "Infix Expression: " << infix << endl;
    string postfix = infixToPostfix(infix);
    cout << "Postfix Expression: " << postfix << endl;
    return 0;
}
```

Q19:

```
#include <iostream>
#include <stack>
#include <string>
#include <cmath>  // For pow() function

using namespace std;
```

```
// Function to check if the given
character is an operand
bool isOperand(char c) {
    return isdigit(c) || isalpha(c);
// Modify this if you expect multi-
digit numbers or variables
}


// Function to perform arithmetic
operations
int applyOp(int a, int b, char op) {
    switch (op) {
    case '+': return a + b;
    case '-': return a - b;
    case '*': return a * b;
    case '/': return b ? a / b : throw
invalid_argument("Division by zero.");
    case '^': return pow(a, b);  //
Assumes a^b
    default: throw
invalid_argument("Unsupported operator
encountered.");
    }
```

```
}

// Function to evaluate postfix
expression
int evaluatePostfix(string postfix) {
    stack<int> stack;

    for (int i = 0; i <
postfix.length(); i++) {
        char c = postfix[i];

        // If the character is an
operand, push it to the stack
        if (isOperand(c)) {
            stack.push(c - '0');  //
Assumes the token is a single digit
        }
        // Operator
        else {
            int val1 = stack.top();
// It's important to note that the
first pop is the second operand
```

```
            stack.pop();
            int val2 = stack.top();
            stack.pop();
            stack.push(applyOp(val2,
val1, c));
        }
    }

    return stack.top();
}

int main() {
    string postfix = "53+62/35+";
    cout << "Postfix Expression: " <<
postfix << endl;
    int result =
evaluatePostfix(postfix);
    cout << "Evaluated Result: " <<
result << endl;
    return 0;
}
```

Q20:

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

class Stack {
private:
    queue<int> q1, q2; // Two queues
    int curr_size;     // Current size
of the stack

public:
    Stack() {
        curr_size = 0;
    }

    void push(int x) {
        curr_size++;

        // Push x first in empty q2
        q2.push(x);
```

```
        // Push all the remaining
elements in q1 to q2.
        while (!q1.empty()) {
            q2.push(q1.front());
            q1.pop();
        }

        // Swap the names of two
queues
        queue<int> q = q1;
        q1 = q2;
        q2 = q;
    }

    void pop() {
        // if no elements are there to
pop
        if (q1.empty())
            return;
        q1.pop();
        curr_size--;
    }
```

```
    int top() {
        if (q1.empty())
            return -1;
        return q1.front();
    }

    int size() {
        return curr_size;
    }
};

int main() {
    Stack s;
    s.push(1);
    s.push(2);
    s.push(3);

    cout << "Current size: " <<
s.size() << endl;
```

```
    cout << "Top element: " << s.top()
<< endl;
    s.pop();
    cout << "Top element after one
pop: " << s.top() << endl;
    cout << "Current size: " <<
s.size() << endl;

    return 0;
}
```