Experiment 8a

Insertion, Deletion, and Traversal in Binary Search Tree

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

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class BST {
public:
    Node* root;

    BST() : root(nullptr) {}

    // Function to insert a new node with given data
    void insert(int data) {
        root = insertRec(root, data);
    }

    // Function to delete a node with given data
    void deleteNode(int data) {
        root = deleteRec(root, data);
    }
```

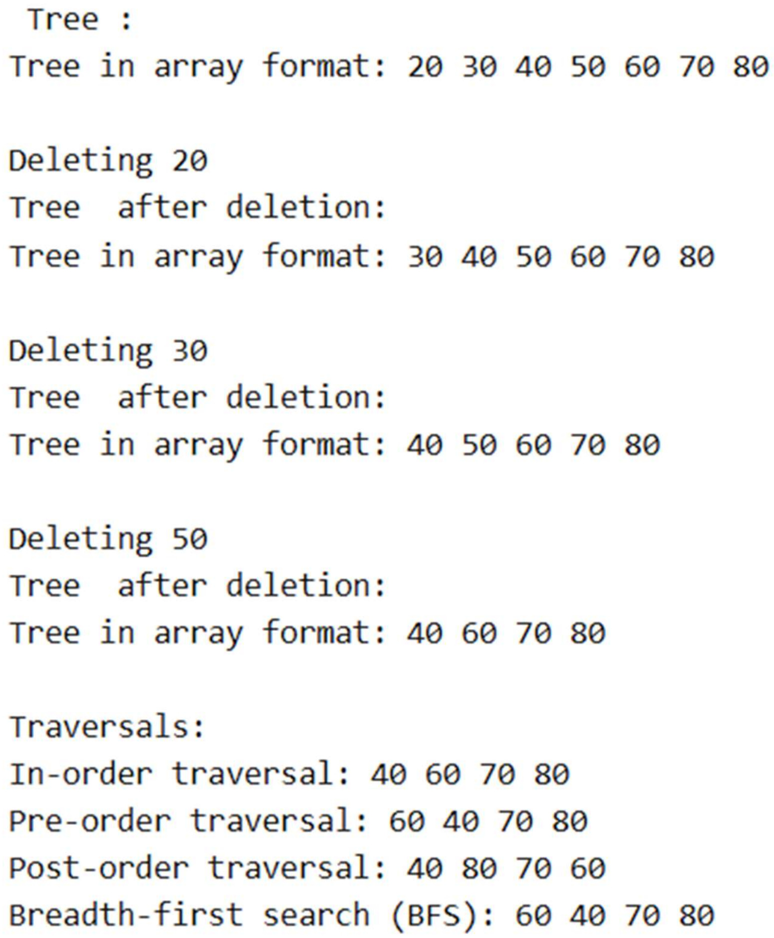
```
// Function to print the tree in array format
void printArray() {
    std::vector<int> arr;
    fillArray(root, arr);
    std::cout << "Tree in array format: ";
    for (int value : arr) {
        std::cout << value << " ";
    }
    std::cout << std::endl;
}

// In-order traversal
void inOrder() {
    std::cout << "In-order traversal: ";
    inOrderRec(root);
    std::cout << std::endl;
}

// Pre-order traversal
void preOrder() {
    std::cout << "Pre-order traversal: ";
    preOrderRec(root);
    std::cout << std::endl;
}

// Post-order traversal
void postOrder() {
    std::cout << "Post-order traversal: ";
    postOrderRec(root);
    std::cout << std::endl;
}
```

```
    // Breadth-first search (BFS)
    void bfs() {
        std::cout << "Breadth-first search (BFS): ";
        if (!root) return;

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

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

            if (current->left) q.push(current->left);
            if (current->right) q.push(current->right);
        }
        std::cout << std::endl;
    }

private:
    Node* insertRec(Node* node, int data) {
        if (!node) {
            return new Node(data);
        }
        if (data < node->data) {
            node->left = insertRec(node->left, data);
        } else {
            node->right = insertRec(node->right, data);
        }
        return node;
```

```
}

Node* deleteRec(Node* node, int data) {
    if (!node) return node;

    if (data < node->data) {
        node->left = deleteRec(node->left, data);
    } else if (data > node->data) {
        node->right = deleteRec(node->right, data);
    } else {
        if (!node->left) {
            Node* temp = node->right;
            delete node;
            return temp;
        } else if (!node->right) {
            Node* temp = node->left;
            delete node;
            return temp;
        }
        Node* temp = minValueNode(node->right);
        node->data = temp->data;
        node->right = deleteRec(node->right, temp->data);
    }
    return node;
}

Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left) {
        current = current->left;
    }
    return current;
}

void fillArray(Node* node, std::vector<int>& arr) {
    if (node) {
        fillArray(node->left, arr);
        arr.push_back(node->data);
        fillArray(node->right, arr);
    }
}

void inOrderRec(Node* node) {
    if (node) {
        inOrderRec(node->left);
        std::cout << node->data << " ";
        inOrderRec(node->right);
    }
}

void preOrderRec(Node* node) {
    if (node) {
        std::cout << node->data << " ";
        preOrderRec(node->left);
        preOrderRec(node->right);
    }
}

void postOrderRec(Node* node) {
    if (node) {
        postOrderRec(node->left);
        postOrderRec(node->right);
```

```
            std::cout << node->data << " ";
        }
    }
};

int main() {
    BST bst;
    bst.insert(50);
    bst.insert(30);
    bst.insert(20);
    bst.insert(40);
    bst.insert(70);
    bst.insert(60);
    bst.insert(80);
    std::cout << " Tree :\n";
    bst.printArray();

    int toDelete[] = {20, 30, 50};
    for (int value : toDelete) {
        std::cout << "\nDeleting " << value << "\n";
        bst.deleteNode(value);
        std::cout << "Tree  after deletion:\n";
        bst.printArray();
    }

    std::cout << "\nTraversals:\n";
    bst.inOrder();
    bst.preOrder();
    bst.postOrder();
    bst.bfs();
    return 0;
```

```
}
```



```
  Tree :
 Tree in array format: 20 30 40 50 60 70 80

 Deleting 20
 Tree  after deletion:
 Tree in array format: 30 40 50 60 70 80

 Deleting 30
 Tree  after deletion:
 Tree in array format: 40 50 60 70 80

 Deleting 50
 Tree  after deletion:
 Tree in array format: 40 60 70 80

 Traversals:
 In-order traversal: 40 60 70 80
 Pre-order traversal: 60 40 70 80
 Post-order traversal: 40 80 70 60
 Breadth-first search (BFS): 60 40 70 80
```

Experiment 8B

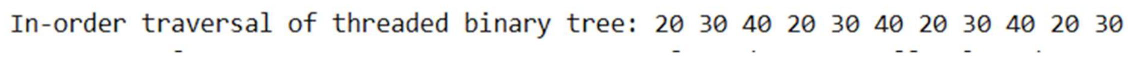
.Insertion,Deletion and Traversal in Threaded Binary Tree.

```
#include <iostream>


struct Node {
   int data;
   Node* left;
   Node* right;
   bool isThreaded;


   Node(int val) : data(val), left(nullptr), right(nullptr), isThreaded(false) {}
};
```

```
class ThreadedBinaryTree {
public:
    Node* root;

    ThreadedBinaryTree() : root(nullptr) {}

    // Insert a new node with given data
    void insert(int data) {
        if (!root) {
            root = new Node(data);
            return;
        }

        Node* current = root;
        Node* parent = nullptr;

        while (current) {
            parent = current;
            if (data < current->data) {
                if (!current->left) break;
                current = current->left;
            } else {
                if (current->isThreaded) break;
                current = current->right;
            }
        }

        Node* newNode = new Node(data);
        if (data < parent->data) {
            parent->left = newNode;
            newNode->right = parent;
            newNode->isThreaded = true;
        } else {
            newNode->right = parent->right;
            parent->right = newNode;
            parent->isThreaded = false;
        }
    }

    // In-order traversal for threaded binary tree
    void inOrder() {
        Node* current = leftMost(root);
        while (current) {
            std::cout << current->data << " ";
            if (current->isThreaded) {
                current = current->right;
            } else {
                current = leftMost(current->right);
            }
        }
        std::cout << std::endl;
    }

    // Utility function to find the leftmost node
    Node* leftMost(Node* node) {
        while (node && node->left) {
            node = node->left;
        }
        return node;
    }
};
```

```
int main() {
    ThreadedBinaryTree tbt;

    tbt.insert(50);
    tbt.insert(30);
    tbt.insert(20);
    tbt.insert(40);
    tbt.insert(70);
    tbt.insert(60);
    tbt.insert(80);

    std::cout << "In-order traversal of threaded binary tree: ";
    tbt.inOrder();

    return 0;
}
```

OUTPUT:



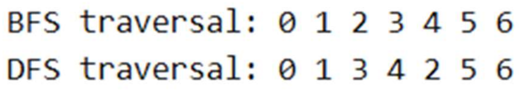
```
In-order traversal of threaded binary tree: 20 30 40 20 30 40 20 30 40 20 30
```

Experiment 10

Graph Depth First and Breadth First Traversal

```
#include <iostream>

#include <vector>

#include <queue>

#include <stack>


class Graph {

public:

    Graph(int vertices) : vertices(vertices) {

        adjList.resize(vertices);

    }


    // Function to add an edge to the graph

    void addEdge(int src, int dest) {

        adjList[src].push_back(dest);

        adjList[dest].push_back(src); // For undirected graph

    }


    // Breadth-First Search (BFS)

    void bfs(int start) {

        std::vector<bool> visited(vertices, false);

        std::queue<int> q;

        visited[start] = true;

        q.push(start);

        std::cout << "BFS traversal: ";

        while (!q.empty()) {

            int node = q.front();

            q.pop();

            std::cout << node << " ";
            for (int neighbor : adjList[node]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
        std::cout << std::endl;
    }

    // Depth-First Search (DFS)
    void dfs(int start) {
        std::vector<bool> visited(vertices, false);
        std::cout << "DFS traversal: ";
        dfsUtil(start, visited);
        std::cout << std::endl;
    }

private:
    int vertices;                    // Number of vertices
    std::vector<std::vector<int>> adjList; // Adjacency list

    // Utility function for DFS
    void dfsUtil(int node, std::vector<bool>& visited) {
        visited[node] = true;
        std::cout << node << " ";
        for (int neighbor : adjList[node]) {
            if (!visited[neighbor]) {
                dfsUtil(neighbor, visited);
            }
        }
    }
};

int main() {
    Graph graph(7);
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(1, 4);
    graph.addEdge(2, 5);
    graph.addEdge(2, 6);
    graph.bfs(0);
    graph.dfs(0);
    return 0;
}
```

OUTPUT



```
BFS traversal: 0 1 2 3 4 5 6
DFS traversal: 0 1 3 4 2 5 6
```