Nithin S
221IT085

IT206 Lab Assignment

Q1) Implement Insertion and deletion in a Binary Search Tree

CODE

```
#include <stdio.h>
#include <stdlib.h>

struct Node
{
    int data;
    struct Node *lChild;
    struct Node *rChild;
};

struct Node *insert(struct Node *p, int key)
{
    if (p == NULL)
    {
        struct Node *temp = NULL;
        temp = (struct Node *)malloc(sizeof(struct
Node));
        temp->data = key;
        temp->lChild = temp->rChild = NULL;
        return temp;
    }
    if (key > p->data)
        p->rChild = insert(p->rChild, key);
    else if (key < p->data)
        p->lChild = insert(p->lChild, key);
```

```
    return p;
}

int height(struct Node *root)
{
    int x = 0, y = 0;
    if (root == NULL)
        return 0;
    x = height(root->lChild);
    y = height(root->rChild);
    if (x > y)
        return x + 1;
    else
        return y + 1;
}

struct Node *InPre(struct Node *p)
{
    while (p && p->rChild != NULL)
        p = p->rChild;
    return p;
}

struct Node *InSucc(struct Node *p)
{
    while (p && p->lChild != NULL)
    {
        p = p->lChild;
    }
    return p;
}

struct Node *delete(struct Node *root, int key)
{
    struct Node *q = NULL;
    if (root == NULL)
        return NULL;
    if (root->lChild == NULL && root->rChild == NULL)
// ie a leaf node
    {
        free(root);
```

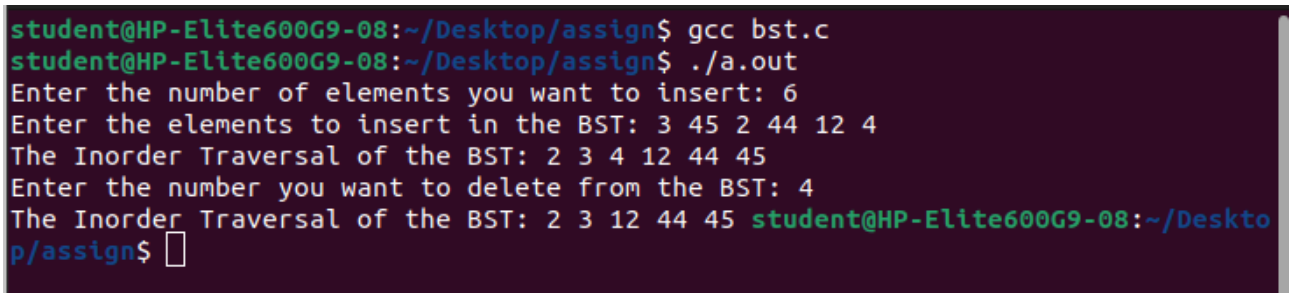
```
        return NULL;
    }
    if (key < root->data)
        root->lChild = delete (root->lChild, key);
    else if (key > root->data)
        root->rChild = delete (root->rChild, key);
    else // ie when the key is found
    {
        if (height(root->lChild) > height(root-
>rChild))
        {
            q = InPre(root->lChild);
            root->data = q->data;
            root->lChild = delete (root->lChild, q-
>data);
        }
        else
        {
            q = InSucc(root->rChild);
            root->data = q->data;
            root->rChild = delete (root->rChild, q-
>data);
        }
    }
    return root;
}

void inOrder(struct Node *p)
{
    if (p)
    {
        inOrder(p->lChild);
        printf("%d ", p->data);
        inOrder(p->rChild);
    }
}

int main()
{
    struct Node *bst1 = NULL;
    int n;
```

```
    printf("Enter the number of elements you want to
insert: ");
    scanf("%d", &n);
    printf("Enter the elements to insert in the BST:
");
    for (int i = 0; i < n; i++)
    {
        int x;
        scanf("%d", &x);
        bst1 = insert(bst1, x);
    }
    printf("The Inorder Traversal of the BST: ");
    inOrder(bst1);
    int del;
    printf("\nEnter the number you want to delete
from the BST: ");
    scanf("%d", &del);
    bst1 = delete (bst1, del);
    printf("The Inorder Traversal of the BST: ");
    inOrder(bst1);
    return 0;
}
```

OUTPUT



Q2) Implement BFS in graph.

CODE

```
// BFS algorithm in C
```

```c
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40

struct queue
{
    int items[SIZE];
    int front;
    int rear;
};

struct queue *createQueue();
void enqueue(struct queue *q, int);
int dequeue(struct queue *q);
void display(struct queue *q);
int isEmpty(struct queue *q);
void printQueue(struct queue *q);

struct node
{
    int vertex;
    struct node *next;
};

struct node *createNode(int);

struct Graph
{
    int numVertices;
    struct node **adjLists;
    int *visited;
};

// BFS algorithm
void bfs(struct Graph *graph, int startVertex)
{
    struct queue *q = createQueue();

    graph->visited[startVertex] = 1;
    enqueue(q, startVertex);
```

```
    while (!isEmpty(q))
    {
        printQueue(q);
        int currentVertex = dequeue(q);
        printf("Visited %d\n", currentVertex);

        struct node *temp = graph-
>adjLists[currentVertex];

        while (temp)
        {
            int adjVertex = temp->vertex;

            if (graph->visited[adjVertex] == 0)
            {
                graph->visited[adjVertex] = 1;
                enqueue(q, adjVertex);
            }
            temp = temp->next;
        }
    }
}

// Creating a node
struct node *createNode(int v)
{
    struct node *newNode = malloc(sizeof(struct
node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// Creating a graph
struct Graph *createGraph(int vertices)
{
    struct Graph *graph = malloc(sizeof(struct
Graph));
    graph->numVertices = vertices;
```

```
    graph->adjLists = malloc(vertices * sizeof(struct
node *));
    graph->visited = malloc(vertices * sizeof(int));

    int i;
    for (i = 0; i < vertices; i++)
    {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }

    return graph;
}

// Add edge
void addEdge(struct Graph *graph, int src, int dest)
{
    // Add edge from src to dest
    struct node *newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Create a queue
struct queue *createQueue()
{
    struct queue *q = malloc(sizeof(struct queue));
    q->front = -1;
    q->rear = -1;
    return q;
}

// Check if the queue is empty
int isEmpty(struct queue *q)
{
    if (q->rear == -1)
```

```
        return 1;
    else
        return 0;
}

// Adding elements into queue
void enqueue(struct queue *q, int value)
{
    if (q->rear == SIZE - 1)
        printf("\nQueue is Full!!");
    else
    {
        if (q->front == -1)
            q->front = 0;
        q->rear++;
        q->items[q->rear] = value;
    }
}

// Removing elements from queue
int dequeue(struct queue *q)
{
    int item;
    if (isEmpty(q))
    {
        printf("Queue is empty");
        item = -1;
    }
    else
    {
        item = q->items[q->front];
        q->front++;
        if (q->front > q->rear)
        {
            printf("Resetting queue ");
            q->front = q->rear = -1;
        }
    }
    return item;
}
```

```
// Print the queue
void printQueue(struct queue *q)
{
    int i = q->front;

    if (isEmpty(q))
    {
        printf("Queue is empty");
    }
    else
    {
        printf("\nQueue contains \n");
        for (i = q->front; i < q->rear + 1; i++)
        {
            printf("%d ", q->items[i]);
        }
    }
}

int main()
{
    struct Graph *graph = createGraph(6);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 4);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);

    bfs(graph, 0);

    return 0;
}
```

OUTPUT

```
student@HP-Elite600G9-08:~/Desktop/assign$ gcc bfs.c
student@HP-Elite600G9-08:~/Desktop/assign$ ./a.out

Queue contains
0 Resetting queue Visited 0

Queue contains
2 1 Visited 2

Queue contains
1 4 Visited 1

Queue contains
4 3 Visited 4

Queue contains
3 Resetting queue Visited 3
```