Lab No: 3 **Date: 2081/12/06**

Title: Write a program to evaluate the user input postfix or prefix expression.

Infix expression: The expression of the form "a operator b" (a + b) i.e., when an operator is in-between every pair of operands.

Postfix expression: The expression of the form "a b operator" (ab+) i.e., When every pair of operands is followed by an operator.

Examples:

Input: A + B * C + D

Output: ABC*+D+

The compiler scans the expression either from left to right or from right to left. Consider the expression: a + b * c + d

- The compiler first scans the expression to evaluate the expression b * c, then again scans the expression to add a to it.
- The result is then added to d after another scan.

The repeated scanning makes it very inefficient. Infix expressions are easily readable and solvable by humans whereas the computer cannot differentiate the operators and parenthesis easily so, it is better to convert the expression to postfix (or prefix) form before evaluation.

IDE: Visual Studio Code

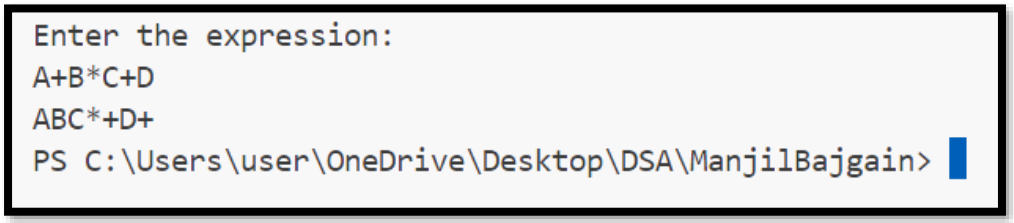
Language: C

Source code:

```
#include <stdio.h>
#include <ctype.h> // For isalnum

char stack[20];
int top = -1;

void push(char x) {
   stack[++top] = x;  // Increment top before assignment
}

char pop() {
   if (top == -1) {
      printf("Stack is empty\n");
      // return '\0';
   } else {
      return stack[top--];  // Decrement top after returning value
   }
}

int priority(char x) {
   if (x == '(') return 0;
   if (x == '+' || x == '-') return 1;
   if (x == '*' || x == '/') return 2;
   // return -1; // Invalid operator
}

int main() {
   char exp[20];
   char *e, x;

   printf("Enter the expression:\n");
   scanf("%s", exp);
   e = exp;

   while (*e != '\0') { // Null terminator
      if (isalnum(*e)) {
         printf("%c", *e);
      } else if (*e == '(') {
         push(*e);
      } else if (*e == ')') {
         while ((x = pop()) != '(') {
            printf("%c", x);
         }
      } else {
         while (top != -1 && priority(stack[top]) >= priority(*e)) {
            printf("%c", pop());
```

```
            }
            push(*e);
        }
        e++;
    }

    while (top != -1) {
        printf("%c", pop());
    }

    return 0;
}
```

Output:



Lab No: 19 **Date: 2081/12/07**

Title: Write a menu based program to insert a node at the beginning, at the end and at the specified position in singly linked list.

A **singly linked list** is a fundamental data structure, it consists of **nodes** where each node contains a **data** field and a **reference** to the next node in the linked list. The next of the last node is **null**, indicating the end of the list. Linked Lists support efficient insertion and deletion operations.

- createList(): To create the list with the 'n' number of nodes initially as defined by the user.
- traverse(): To see the contents of the linked list, it is necessary to traverse the given linked list. The given traverse() function traverses and prints the content of the linked list.
- insertAtFront(): This function simply inserts an element at the front/beginning of the linked list.
- insertAtEnd(): This function inserts an element at the end of the linked list.
- insertAtPosition(): This function inserts an element at a specified position in the linked list.
- deleteFirst(): This function simply deletes an element from the front/beginning of the linked list.
- deleteEnd(): This function simply deletes an element from the end of the linked list.
- deletePosition(): This function deletes an element from a specified position in the linked list.
- maximum(): This function finds the maximum element in a linked list.
- sort(): This function sorts the given linked list in ascending order.
- reverseLL(): This function reverses the given linked list.
- display(): This function displays the linked list.

IDE: Visual Studio Code

Language: C

Source code:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node* head = NULL;
void insert_at_beginning(int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = head;
    head = new_node;
}
void insert_at_end(int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;
    if (head == NULL) {
        head = new_node;
        return;
    }
    struct Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = new_node;
}
void insert_at_position(int data, int position) {
    if (position < 1) {
```

```
        printf("Invalid position!\n");
        return;
    }
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    if (position == 1) {
        new_node->next = head;
        head = new_node;
        return;
    }
    struct Node* temp = head;
    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of range!\n");
        return;
    }
    new_node->next = temp->next;
    temp->next = new_node;
}
void display() {
    struct Node* temp = head;
    if (temp == NULL) {
        printf("List is empty!\n");
        return;
    }
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
```

```
    printf("NULL\n");
}
int main() {
    int choice, data, position;
    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
        printf("3. Insert at Position\n");
        printf("4. Display List\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
switch (choice) {
 case 1:
            printf("Enter data: ");
               scanf("%d", &data);
               insert_at_beginning(data);
               break;
            case 2:
               printf("Enter data: ");
               scanf("%d", &data);
               insert_at_end(data);
               break;
            case 3:
               printf("Enter data: ");
               scanf("%d", &data);
               printf("Enter position: ");
               scanf("%d", &position);
               insert_at_position(data, position);
               break;
```

```
            case 4:
                display();
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        } }
    return 0;
}
```

```
Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Display List
5. Exit
Enter your choice: 1
Enter data: 23

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Display List
5. Exit
Enter your choice: 2
Enter data: 24

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Display List
5. Exit
Enter your choice: 3
Enter data: 25
Enter position: 3

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Display List
5. Exit
Enter your choice: 4
23 -> 24 -> 25 -> NULL

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Display List
5. Exit
Enter your choice: 5
PS C:\Users\user\OneDrive\Desktop\DSA\ManjilBajgain>
```

Lab No: 10 **Date: 2081/12/07**

Title: Write a menu based program to delete a node from the beginning, from the end and at the specified position in doubly linked list.

Doubly Linked List:

A doubly linked list is a more complex data structure than a singly linked list, but it offers several advantages. The main advantage of a doubly linked list is that it allows for efficient traversal of the list in both directions. This is because each node in the list contains a pointer to the previous node and a pointer to the next node. This allows for quick and easy insertion and deletion of nodes from the list, as well as efficient traversal of the list in both directions.

- createList(): To create the list with the 'n' number of nodes initially as defined by the user.
- traverse(): To see the contents of the linked list, it is necessary to traverse the given linked list. The given traverse() function traverses and prints the content of the linked list.
- insertAtFront(): This function simply inserts an element at the front/beginning of the linked list.
- insertAtEnd(): This function inserts an element at the end of the linked list.
- insertAtPosition(): This function inserts an element at a specified position in the linked list.
- deleteFirst(): This function simply deletes an element from the front/beginning of the linked list.

IDE: Visual Studio Code

Language: C

Source code:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
struct Node* head = NULL;
void insert_at_end(int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;
    if (head == NULL) {
        new_node->prev = NULL;
        head = new_node;
        return;
    }
    struct Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = new_node;
    new_node->prev = temp;
}
void delete_from_beginning() {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }
```

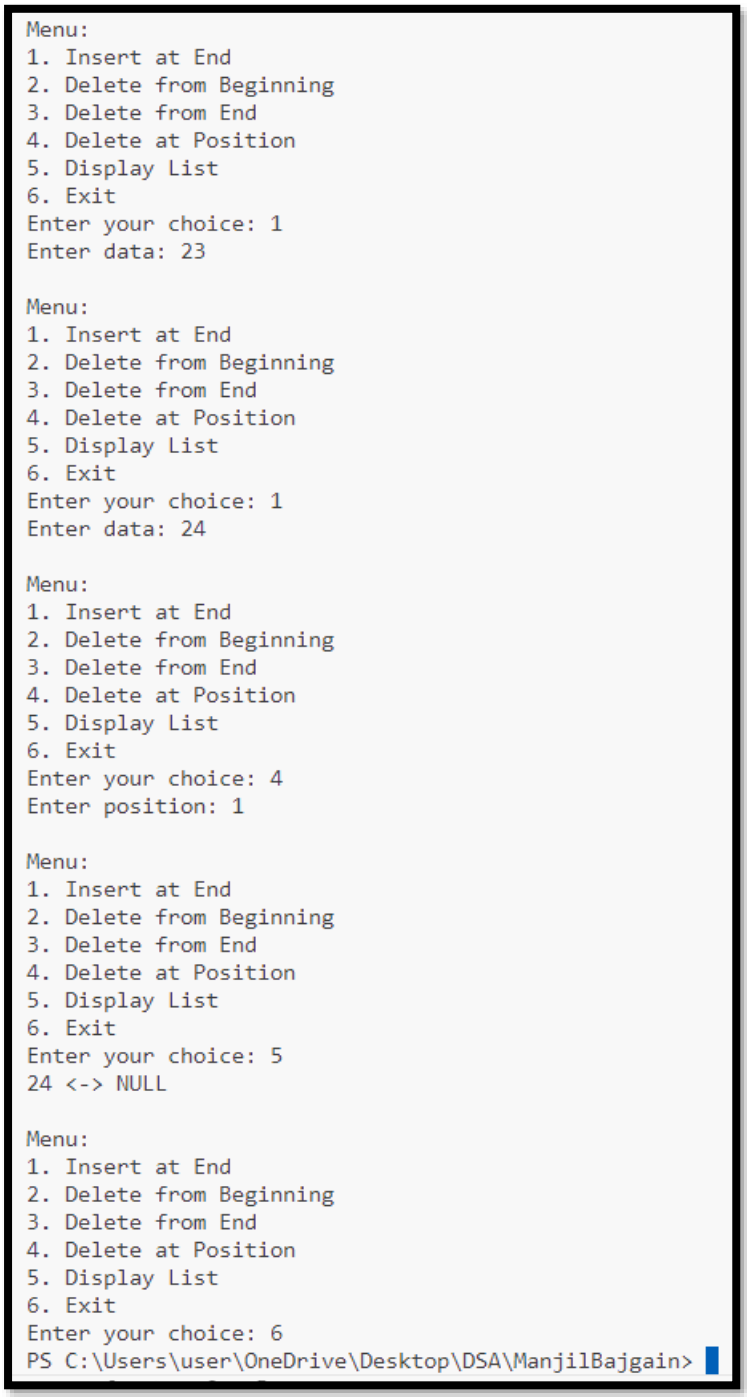
```
    struct Node* temp = head;
    head = head->next;
    if (head != NULL) {
        head->prev = NULL;
    }
    free(temp);
}
void delete_from_end() {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    if (temp->prev != NULL) {
        temp->prev->next = NULL;
    } else {
        head = NULL;
    }
    free(temp);
}
void delete_at_position(int position) {
    if (head == NULL || position < 1) {
        printf("Invalid position or list is empty!\n");
        return;
    }
    struct Node* temp = head;
    for (int i = 1; temp != NULL && i < position; i++) {
        temp = temp->next;
```

```
    }
    if (temp == NULL) {
        printf("Position out of range!\n");
        return;
    }
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    } else {
        head = temp->next;
    }
    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }
    free(temp);
}
void display() {
    struct Node* temp = head;
    if (temp == NULL) {
        printf("List is empty!\n");
        return;
    }
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
int main() {
    int choice, data, position;
    while (1) {
        printf("\nMenu:\n");
```

```
        printf("1. Insert at End\n");
        printf("2. Delete from Beginning\n");
        printf("3. Delete from End\n");
        printf("4. Delete at Position\n");
        printf("5. Display List\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
switch (choice) {
            case 1:
                printf("Enter data: ");
                scanf("%d", &data);
                insert_at_end(data);
                break;
            case 2:
                delete_from_beginning();
                break;
            case 3:
                delete_from_end();
                break;
            case 4:
                printf("Enter position: ");
                scanf("%d", &position);
                delete_at_position(position);
                break;
            case 5:
                display();
                break;
            case 6:
                exit(0);
            default:
```

```
            printf("Invalid choice! Please try again.\n");
        }
    }
    return 0;
}
```

Output:



Lab No: 12 **Date: 2081/12/07**

Title: Write a program to calculate shortest path using Dijkstra's Algorithm.

Dijkstra's Algorithm using Adjacency Matrix

The idea is to generate a SPT (shortest path tree) with a given source as a root. Maintain an Adjacency Matrix with two sets,

- One set contains vertices included in the shortest-path tree,
- The other set includes vertices not yet included in the shortest-path tree.

At every step of the algorithm, find a vertex that is in the other set (set not yet included) and has a minimum distance from the source.

Step-by-step approach:

- Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
- Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign the distance value as 0 for the source vertex so that it is picked first.
- While sptSet doesn't include all vertices
    - Pick a vertex u that is not there in sptSet and has a minimum distance value.
    - Include u to sptSet.
    - Then update the distance value of all adjacent vertices of u. Iterate through all adjacent and for every adjacent v, if the sum of the distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v .

IDE: Visual Studio Code

Language: C

Source code:

```
#include <stdio.h>
#define INFINITY 999

void dij(int n, int v, int cost[10][10], int dist[]);

int main()
{
   int a, v, i, j, cost[10][10], dist[10];

   printf("\nEnter the number of nodes: ");
   scanf("%d", &a);
   printf("\nEnter the cost matrix:\n");

   for (i = 1; i <= a; i++)
   {
      for (j = 1; j <= a; j++)
      {
         scanf("%d", &cost[i][j]);
         if (cost[i][j] == 0)
         {
            cost[i][j] = INFINITY;
         }
      }
   }

   printf("\nEnter the source node: ");
   scanf("%d", &v);

   dij(a, v, cost, dist);

   printf("\nShortest Paths from Node %d:\n", v);
   for (i = 1; i <= a; i++)
   {
      if (i != v)
      {
         printf("Node %d to Node %d, Cost = %d\n", v, i, dist[i]);
      }
   }

   return 0;
}
void dij(int n, int v, int cost[10][10], int dist[])
{
   int i, u, count, w, flag[10], min;
```

```
  for (i = 1; i <= n; i++)
  {
     flag[i] = 0;
     dist[i] = cost[v][i];
  }
 count = 2;
 while (count <= n)
  {
     min = INFINITY;
     for (w = 1; w <= n; w++)
     {
        if (dist[w] < min && !flag[w])
        {
           min = dist[w];
           u = w;
        }
     }
 flag[u] = 1;
     count++;
for (w = 1; w <= n; w++)
     {
        if ((dist[u] + cost[u][w] < dist[w]) && !flag[w])
        {
           dist[w] = dist[u] + cost[u][w];
        }
     }
  }
}
```

Output:

```
Enter the number of nodes: 2

Enter the cost matrix:
12
23
24
12

Enter the source node: 2

Shortest Paths from Node 2:
Node 2 to Node 1, Cost = 24
PS C:\Users\user\OneDrive\Desktop\DSA\ManjilBajgain>
```

Lab No: 16 **Date: 2081/12/08**

Title: Write a program to implement Double hashing and Quadratic hashing.

Hashing is an improvement technique over the Direct Access Table. The idea is to use a hash function that converts a given phone number or any other key to a smaller number and uses the small number as the index in a table called a hash table.

Double hashing

Double hashing is a collision resolution technique used in hash tables. It works by using two hash functions to compute two different hash values for a given key. The first hash function is used to compute the initial hash value, and the second hash function is used to compute the step size for the probing sequence. Double hashing has the ability to have a low collision rate, as it uses two hash functions to compute the hash value and the step size. This means that the probability of a collision occurring is lower than in other collision resolution techniques such as linear probing or quadratic probing.

Quadratic Probing

Quadratic probing is an open-addressing scheme where we look for the i2‘th slot in the i’th iteration if the given hash value x collides in the hash table. We have already discussed linear probing implementation.

IDE: Visual Studio Code

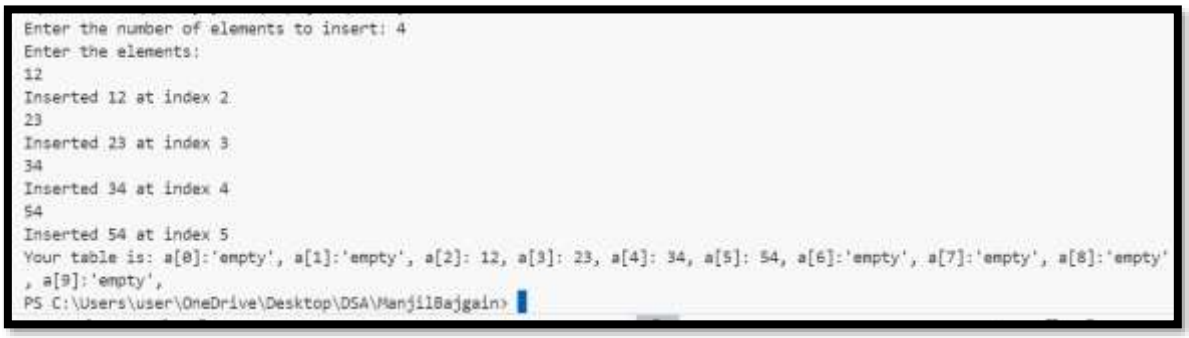
Language: C

Source code:

```
//Quadratic hashing
#include <stdio.h>
#define TABLE_SIZE 10

int hashTable[TABLE_SIZE]; // Hash table

// Hash function
int hash(int key)
{
   return key % TABLE_SIZE;
}

// Quadratic probing function to insert a key
void insert(int key)
{
   int index = hash(key);
   int i = 0;
   while (hashTable[(index + i * i) % TABLE_SIZE] != -1)
   {
      i++;
      if (i == TABLE_SIZE)
      {
         printf("Hash table is full, cannot insert %d\n", key);
         return;
      }
   }
   hashTable[(index + i * i) % TABLE_SIZE] = key;
   printf("Inserted %d at index %d\n", key, (index + i * i) % TABLE_SIZE);
}

int main()
{
   for (int i = 0; i < TABLE_SIZE; i++)
   {
      hashTable[i] = -1; // Initialize hash table with -1 (indicating empty slots)
   }

   int n, key;
   printf("Enter the number of elements to insert: ");
   scanf("%d", &n);
   printf("Enter the elements:\n");
   for (int i = 0; i < n; i++)
   {
      scanf("%d", &key);
      insert(key);
```

```
    }
    printf("Your table is: ");
    for (int i = 0; i < TABLE_SIZE; i++)
    {
        if(hashTable[i] == -1)
            printf("a[%d]:'empty', ",i);
        else
        printf("a[%d]: %d, ",i, hashTable[i]);
    }
    return 0;
}
```

Output:

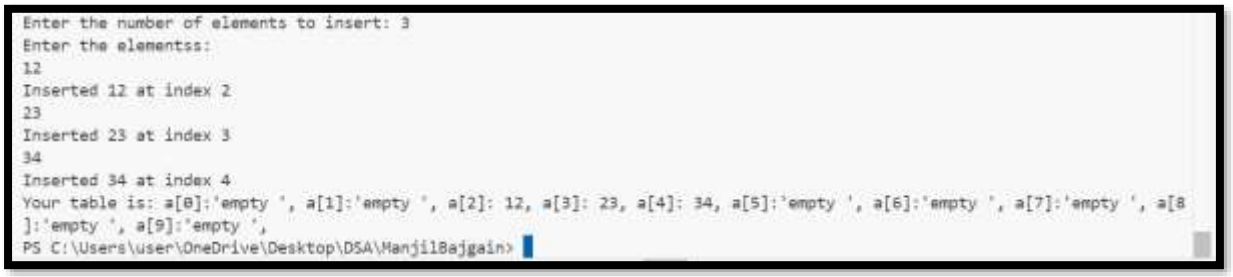


Source code:

```
//Double hashing
#include <stdio.h>
#define TABLE_SIZE 10
int hashTable[TABLE_SIZE]; // Hash table
int hash1(int key)
{
   return key % TABLE_SIZE;
}
int hash2(int key)
{
   return (7 - (key % 7)); // Using a prime number smaller than TABLE_SIZE
}

void insert(int key)
{
   int index = hash1(key);
   int step = hash2(key);
   int i = 0;

   while (hashTable[(index + i * step) % TABLE_SIZE] != -1)
   {
      i++;
      if (i == TABLE_SIZE)
      {
         printf("Hash table is full, cannot insert %d\n", key);
         return;
      }
   }
   hashTable[(index + i * step) % TABLE_SIZE] = key;
   printf("Inserted %d at index %d\n", key, (index + i * step) % TABLE_SIZE);
}

int main()
{
   for (int i = 0; i < TABLE_SIZE; i++)
   {
      hashTable[i] = -1; // Initialize hash table with -1 (indicating empty slots)
   }

   int n, key;
   printf("Enter the number of elements to insert: ");
   scanf("%d", &n);
   printf("Enter the elementss:\n");
   for (int i = 0; i < n; i++)
```

```
    {
        scanf("%d", &key);
        insert(key);
    }
printf("Your table is: ");
for (int i = 0; i < TABLE_SIZE; i++)
{
    if(hashTable[i] == -1)
        printf("a[%d]:'empty ', ",i);
    else
    printf("a[%d]: %d, ",i, hashTable[i]);
}
return 0;
}
```

Output:



Lab No: 13 **Date: 2081/12/08**

Title: Write a program to calculate minimum spanning tree using Prim's algorithm.

Prim's algorithm:

Prim's algorithm is a Greedy algorithm like Kruskal's algorithm. This algorithm always starts with a single node and moves through several adjacent nodes, in order to explore all of the connected edges along the way.

- The algorithm starts with an empty spanning tree.
- The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included.
- At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

IDE: Visual Studio Code

Language: C

Source code:

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
#define V 5  // Number of vertices in the graph
int minKey(int key[], bool mstSet[]) {
   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++) {
      if (!mstSet[v] && key[v] < min) {
         min = key[v], min_index = v;
      }
   }
   return min_index;
}
void printMST(int parent[], int graph[V][V]) {
   printf("Edge \tWeight\n");
   for (int i = 1; i < V; i++) {
      printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);
   }
}

void primMST(int graph[V][V]) {
   int parent[V];
   int key[V];
   bool mstSet[V];

   for (int i = 0; i < V; i++) {
      key[i] = INT_MAX;
      mstSet[i] = false;
   }
key[0] = 0;
   parent[0] = -1;

   for (int count = 0; count < V - 1; count++) {
      int u = minKey(key, mstSet);
      mstSet[u] = true;

      for (int v = 0; v < V; v++) {
         if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {
            parent[v] = u;
            key[v] = graph[u][v];
```

```
            }
        }
    }
    printMST(parent, graph);
}
int main() {
    int graph[V][V];
    printf("Enter the adjacency matrix for the graph (Enter 0 for no direct edge):\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
printf("Minimum Spanning Tree using Prim's Algorithm:\n");
    primMST(graph);
    return 0;
}
```

Output:

```
Enter the adjacency matrix for the graph (Enter 0 for no direct edge):
0 2 0 6 0
2 0 3 8 5
0 3 0 0 7
6 8 0 0 9
0 5 7 9 0
Minimum Spanning Tree using Prim's Algorithm:
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
PS C:\Users\user\OneDrive\Desktop\DSA\ManjilBajgain>
```

Lab No: 11 **Date: 2081/12/08**

Title: Write a program for BST Insertion, Deletion, Traversal, and Search.

Binary search:

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.A new key is always inserted at the leaf by maintaining the property of the binary search tree. We start searching for a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node. The below steps are followed while we try to insert a node into a binary search tree:

- Initilize the current node (say, currNode or node) with root node
- Compare the key with the current node.
- Move left if the key is less than or equal to the current node value.
- Move right if the key is greater than current node value.
- Repeat steps 2 and 3 until you reach a leaf node.
- Attach the new key as a left or right child based on the comparison with the leaf node's value.

IDE: Visual Studio Code

Language: C

Source code:

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

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

struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->left = newNode->right = NULL;
   return newNode;
}

struct Node* insert(struct Node* root, int data) {
   if (root == NULL) {
      return createNode(data);
   }
   if (data < root->data) {
      root->left = insert(root->left, data);
   } else if (data > root->data) {
      root->right = insert(root->right, data);
   }
   return root;
}

struct Node* findMin(struct Node* root) {
   while (root->left != NULL) {
      root = root->left;
   }
   return root;
}

struct Node* deleteNode(struct Node* root, int data) {
   if (root == NULL) {
      return root;
   }
   if (data < root->data) {
      root->left = deleteNode(root->left, data);
   } else if (data > root->data) {
      root->right = deleteNode(root->right, data);
   } else {
      if (root->left == NULL) {
```

```
            struct Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }
        struct Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

struct Node* search(struct Node* root, int data) {
    if (root == NULL || root->data == data) {
        return root;
    }
    if (data < root->data) {
        return search(root->left, data);
    }
    return search(root->right, data);
}

void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

int main() {
    struct Node* root = NULL;
    int choice, data, n;
    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Search\n");
        printf("4. Inorder Traversal\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
```

```
            case 1:
                printf("Enter number of elements to insert: ");
                scanf("%d", &n);
                printf("Enter %d elements: ", n);
                for (int i = 0; i < n; i++) {
                    scanf("%d", &data);
                    root = insert(root, data);
                }
                break;
            case 2:
                printf("Enter data to delete: ");
                scanf("%d", &data);
                root = deleteNode(root, data);
                break;
            case 3:
                printf("Enter data to search: ");
                scanf("%d", &data);
                if (search(root, data) != NULL) {
                    printf("Element found!\n");
                } else {
                    printf("Element not found!\n");
                }
                break;
            case 4:
                printf("Inorder Traversal: ");
                inorder(root);
                printf("\n");
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
    return 0;
}
```

Output:

```
Menu:
1. Insert
2. Delete
3. Search
4. Inorder Traversal
5. Exit
Enter your choice: 1
Enter number of elements to insert: 5
Enter 5 elements: 12
13
14
15
16

Menu:
1. Insert
2. Delete
3. Search
4. Inorder Traversal
5. Exit
Enter your choice: 2
Enter data to delete: 14

Menu:
1. Insert
2. Delete
3. Search
4. Inorder Traversal
5. Exit
Enter your choice: 3
Enter data to search: 13
Element found!

Menu:
1. Insert
2. Delete
3. Search
4. Inorder Traversal
5. Exit
Enter your choice: 4
Inorder Traversal: 12 13 15 16

Menu:
1. Insert
2. Delete
3. Search
4. Inorder Traversal
5. Exit
Enter your choice: 5
PS C:\Users\user\OneDrive\Desktop\DSA\ManjilBajgain>
```