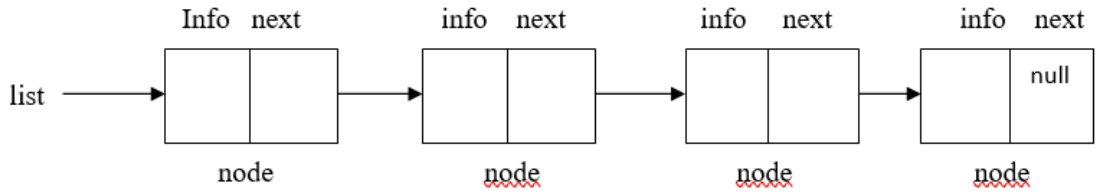
**Program 7 theory:**

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. A linked list is a collection of items in which each item contains within itself the address of thenext item. Each item in the list is called a node and contains two fields:

i) **Information field**: The information field holds the actual element on the list.
ii) **Nextaddress field**: the nextaddress field contains the address of the next node in the list(pointer).




**Notations:**

If p is a pointer to a node

Node(p) - refers to the node pointed by p

Info(p) - refers to the information portion of the node

Next(p) - refers to the next address portion which is a pointer

**Operations on linked lists**

1. **Creation of a list**
2. **Deletion of a node from the linked list**
3. **Traversing and displaying the elements in the list**
4. Counting the number of elements in the list
5. **Searching for an element in the list**
6. **Merging two lists(Concatenating lists)**

/*A music club is interested in creating a song playlist. The facilities to be provided for the users of the playlist are

a. Create a playlist

b. Play a song from starting of the playlist.

c. Play a song from end of the playlist

d. Delete a song from the starting of the playlist

e. Delete a song from the end of the playlist

Design and Implement a menu driven Program in C for the above operations using Singly Linked List.*/

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
struct node
{
  char song[25];
  struct node *next;
};

typedef struct node *NODEPTR;
NODEPTR list=NULL;


/*getnode()*/
NODEPTR getnode()
{
  NODEPTR r;
  r=(NODEPTR)malloc(sizeof(struct node));
  if(r==NULL)
  {
    printf("Allocation failed\n");
    return;
  }
```

```
  return r;
}


//create a list

NODEPTR create(NODEPTR list, char song[])
{
        NODEPTR p,q;
        p=getnode();
        strcpy(p->song,song);
        p->next=NULL;
        if(list==NULL)
                list=p;
        else{
                for(q=list; q->next!=NULL; q=q->next)
                ;
                q->next=p;
        }
        return(list);
}

void playbegin(NODEPTR list)
{
 NODEPTR p;
 p=list;
 if(list==NULL)
   printf("Empty playlist");
 else
 printf("\n Playing %s\n",p->song);
```

```
}

void playend(NODEPTR list)
{
 NODEPTR p;
 p=list;
 if(list==NULL)
  printf("Empty playlist");
 else {
 while(p->next!=NULL)
    p=p->next;
 printf("\n Playing %s\n",p->song);
}
}

NODEPTR deletebegin(NODEPTR list)
{
          NODEPTR p;
          p=list;
       if(list==NULL)
                    printf("\n Empty playlist");
          else{
          printf("\n Song deleted =%s",p->song);
          list=p->next;
          free(p);
          return(list);
       }
}

NODEPTR deleteend(NODEPTR list)
```

```
{
        NODEPTR p,q;
        p=list;
        q=NULL;
        if(list==NULL)
                printf("Empty playlist");
        else if(list->next==NULL)
          {
           printf("\n Song deleted =%s",p->song);
         list=p->next;
         free(p);
         return(list);
          }
        else{
           while(p->next!=NULL)
           {
              q=p;
        p=p->next;
           }
       q->next=p->next;
        printf("\n Song deleted =%s",p->song);
        free(p);
        return(list);
  }
}

void display(NODEPTR list)
{
   NODEPTR p;
   if(list==NULL)
```

```
    {
       printf("Empty list");
    }
    else{
    printf("The playlist contains:\n");
    for(p=list;p!=NULL;p=p->next)
      printf("%s->",p->song);
      printf("\n");
    }
}

main()
{
    int choice;
    char cont;
    char song[25];
    do{
    printf("1->CREATE PLAYLIST 2->PLAY FROM BEGINNINNG 3->PLAY FROM END
4->DELETE FROM BEGINNING  5->DELETE FROM END  6->DISPLAY  7->QUIT\n");
    printf("Enter your choice:");
    scanf("%d",&choice);
    switch(choice)
    {
     case 1:  do{
                        printf("Enter a song:");
                        scanf("%s",song);
                        list = create(list,song);
                        printf("Do you want to enter another song[Y/N]:");
                       scanf(" %c",&cont);
              }while(cont=='Y' || cont=='y');
              break;
```

```
    case 2:   playbegin(list);
              break;

    case 3:   playend(list);
              break;

    case 4:   list=deletebegin(list);
              break;

    case 5:   list=deleteend(list);
              break;

    case 6:   display(list);
              break;

    case 7:   exit(1);

    default : printf("\nNo such option");
              break;

   }
}while(choice!=7);
}
```

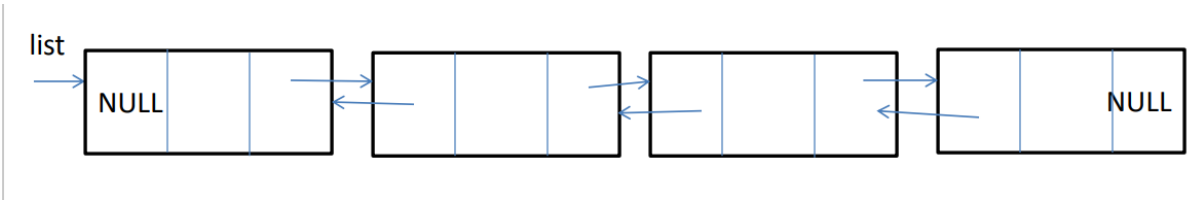
**Program 8 theory**

A doubly linked list is a linear non primitive data structure where each node contains three fields:

– An info field

– Right field pointing to the next node

– Left field pointing to the previous node




Node representation of Doubly linked list


Structure Definition of a node in a Doubly linked list

```
struct node {
    int page; // Page number
    struct node *left; // Pointer to the previous node
    struct node *right; // Pointer to the next node
};
```

P8) You are assigned the task to design a browser history where a person can visit any page and go backward or forward in browser history in any number of steps. Suppose we have to go forward x steps, but we can go only y(where y<x) steps forward because of the last Node, then we return the last node. Similarly, we will return the first node while traveling back.

Design and implement a menu driven Program in C to implement the above operations using Doubly Linked List.

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

// Structure definition for a doubly linked list node
struct node {
    int page; // Page number
    struct node *left; // Pointer to the previous node
    struct node *right; // Pointer to the next node
};

typedef struct node *NODEPTR;
NODEPTR list = NULL; // Global pointer to the head of the list

// Function prototypes
NODEPTR createlist(NODEPTR list, int page);
NODEPTR moveforward(NODEPTR list, int cp, int steps);
NODEPTR movebackward(NODEPTR list, int cp, int steps);
void display(NODEPTR list);
NODEPTR getnode();

// Function to create a new node in the list
NODEPTR createlist(NODEPTR list, int page) {
    NODEPTR p, q;
    p = getnode(); // Allocate memory for a new node
    p->page = page; // Set the page number
```

```
    p->left = NULL;
    p->right = NULL;

    if (list == NULL) { // If the list is empty
        list = p; // Set the new node as the head
    } else {
        // Traverse to the last node
        for (q = list; q->right != NULL; q = q->right);
        q->right = p; // Link the new node to the end of the list
        p->left = q; // Link back to the previous node
    }
    return list; // Return the updated list
}

// Function to move forward in the list by a specified number of steps
NODEPTR moveforward(NODEPTR list, int cp, int steps) {
    NODEPTR p, q;
    int count = 0, s;

    if (list == NULL) { // Handle an empty list
        printf("\n Empty list");
        return NULL;
    }

    // Locate the current page node (cp)
    p = list;
    for (count = 1; p != NULL && count < cp; count++) {
        p = p->right;
    }
```

```
    if (p == NULL) { // If the current page is invalid
        printf("\n Invalid current page!");
        return NULL;
    }

    // Move forward by the specified number of steps
    for (q = p, s = 0; s < steps && q->right != NULL; s++) {
        q = q->right;
    }

    // Return the last valid node reached
    return q;
}

// Function to move backward in the list by a specified number of steps
NODEPTR movebackward(NODEPTR list, int cp, int steps) {
    NODEPTR p, q;
    int count = 0, s;

    if (list == NULL) { // Handle an empty list
        printf("\n Empty list");
        return NULL;
    }

    // Locate the current page node (cp)
    p = list;
    for (count = 1; p != NULL && count < cp; count++) {
        p = p->right;
    }
```

```
    if (p == NULL) { // If the current page is invalid
        printf("\n Invalid current page!");
        return NULL;
    }

    // Move backward by the specified number of steps
    for (q = p, s = 0; s < steps && q->left != NULL; s++) {
        q = q->left;
    }

    // Return the last valid node reached
    return q;
}

// Function to display the contents of the list
void display(NODEPTR list) {
    NODEPTR p = list;

    if (p == NULL) { // Handle an empty list
        printf("\nEmpty list");
    } else {
        printf("\n The page list contains: ");
        while (p != NULL) { // Traverse the list
            printf("%d", p->page);
            if (p->right != NULL) {
                printf("<->"); // Print a bidirectional arrow
            }
            p = p->right;
        }
        printf("\n");
```

```
    }
}

// Function to allocate memory for a new node
NODEPTR getnode() {
    NODEPTR r = (NODEPTR)malloc(sizeof(struct node));
    if (r == NULL) { // Handle memory allocation failure
        printf("\n Node allocation failed");
        exit(0);
    }
    return r;
}

// Main function with a menu-driven interface
void main() {
    int page, choice, steps, cp;
    char cont;
    NODEPTR p;

    do {
        // Display menu
        printf("\n .....MENU.....");
        printf("\n 1->CREATE LIST\t  2->MOVE FORWARD\t  3->MOVE BACKWARD\t 4->DISPLAY 5->EXIT");
        printf("\n Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: // Create list of pages
                printf("\n CREATION OF DOUBLY LINKED LIST OF PAGES IS IN PROGRESS:\n");
```

```
    do {
        printf("Enter a page number: ");
        scanf("%d", &page);
        list = createlist(list, page);
        printf("Do you want to enter another page [Y/N]: ");
        scanf(" %c", &cont);
    } while (cont == 'Y' || cont == 'y');
    display(list);
    break;

case 2: // Move forward
    printf("\n MOVE FORWARD:\n");
    printf("Enter the current page: ");
    scanf("%d", &cp);
    printf("Enter the number of steps to move forward: ");
    scanf("%d", &steps);
    p = moveforward(list, cp, steps);
    if (p != NULL) {
        printf("\n Moved forward to page %d from %dth page", p->page, cp);
    }
    break;

case 3: // Move backward
    printf("\n MOVE BACKWARD:\n");
    printf("Enter the current page: ");
    scanf("%d", &cp);
    printf("Enter the number of steps to move backward: ");
    scanf("%d", &steps);
    p = movebackward(list, cp, steps);
    if (p != NULL) {
```

```
                printf("\n Moved backward to page %d from %dth page", p->page, cp);
            }
            break;

        case 4: // Display the list
            display(list);
            break;

        case 5: // Exit the program
            printf("\n Quitting operation List.....\n");
            break;

        default: // Handle invalid choices
            printf("\n Invalid choice");
            break;
      }
   } while (choice != 5);
}
```

sample output

.....MENU.....

1->CREATE LIST 2->MOVE FORWARD 3->MOVE BACKWARD 4->DISPLAY 5->EXIT

Enter your choice1

CREATION OF DOUBLY LINKED LIST OF PAGES IS IN PROGRESS:

Enter a page number:10

Do you want to enter another page[Y/N]:y

Enter a page number:20

Do you want to enter another page[Y/N]:y

Enter a page number:30

Do you want to enter another page[Y/N]:y

Enter a page number:40

Do you want to enter another page[Y/N]:y

Enter a page number:50

Do you want to enter another page[Y/N]:y

Enter a page number:60

Do you want to enter another page[Y/N]:n

The page list contains: 10<->20<->30<->40<->50<->60<->

.....MENU.....

1->CREATE LIST 2->MOVE FORWARD 3->MOVE BACKWARD 4->DISPLAY 5->EXIT

Enter your choice2

MOVE FORWARD:

Enter the current page:3

Enter the number of steps to move forward3

Moved forward to page 60 from 3th page

.....MENU.....

1->CREATE LIST 2->MOVE FORWARD 3->MOVE BACKWARD 4->DISPLAY 5->EXIT

Enter your choice

2

MOVE FORWARD:

Enter the current page:5

Enter the number of steps to move forward4

Moved forward to  page 60 from 5th page

.....MENU.....

1->CREATE LIST  2->MOVE FORWARD  3->MOVE BACKWARD  4->DISPLAY 5->EXIT

Enter your choice3

MOVE BACKWARD:

Enter the current page:4

Enter the number of steps to move backward3

Moved backwards to  page 10 from 4th page

.....MENU.....

1->CREATE LIST  2->MOVE FORWARD  3->MOVE BACKWARD  4->DISPLAY 5->EXIT

Enter your choice3

MOVE BACKWARD:

Enter the current page:3

Enter the number of steps to move backward5

Moved backwards to  page 10 from 3th page

.....MENU.....

1->CREATE LIST  2->MOVE FORWARD  3->MOVE BACKWARD  4->DISPLAY 5->EXIT

Enter your choice4

The page list contains: 10<->20<->30<->40<->50<->60<->

.....MENU.....

1->CREATE LIST 2->MOVE FORWARD 3->MOVE BACKWARD 4->DISPLAY 5->EXIT

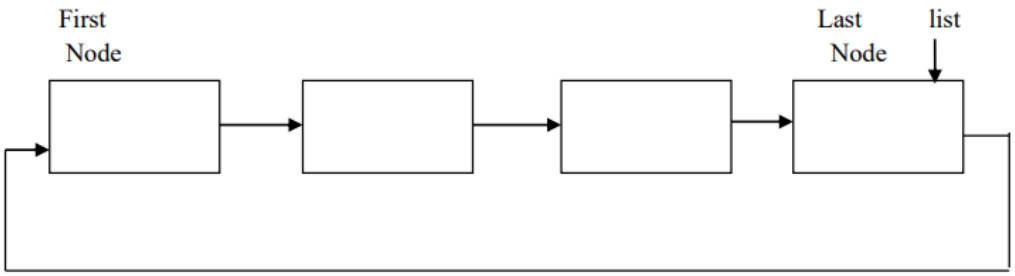
Enter your choice

5

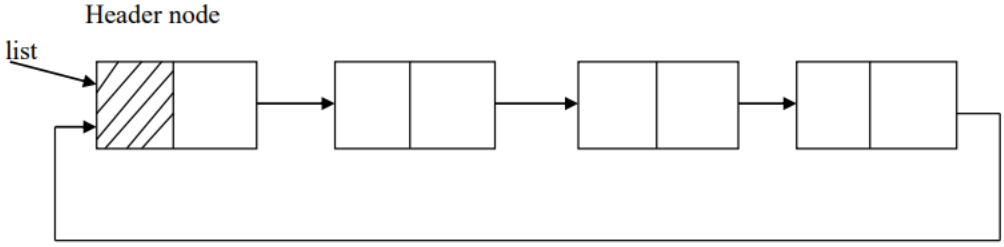
Quitting operation List.....

**Program 9 theory**

The circular linked list is a linked list where all nodes are connected to form a circle. In a circular linked list, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.




A circular list can be traversed by repeatedly executing p=p->next where p is a pointer to the beginning of the list. However since the list is circular, we will not know when the entirelist hs been traversed unless another pointer " list" points to the first node and the condition p==list is tested.An alternative method is to place a header node as the first node of the circular list. The list header may contain a special value in its info field to indicate that it is a header node. Now the list can be traversed using a single pointer and the traversal can be halted when the header node is reached. The external pointer "list" points to the header node.



Design, Develop and Implement a Program in C for the following operations on Singly Circular Linked List (SCLL) with header nodes

a. Represent a Polynomial of the form P(x,y,z) = 6x2 y 2 z-4 yz5 +3x3 yz+2xy5 z-2xyz3

b. Evaluate the polynomial

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

// Node structure for SCLL
struct node {
    int coefficient;
    int x_power;
    int y_power;
    int z_power;
    struct node* next;
};
typedef struct node *NODEPTR;

// Function prototypes
NODEPTR createNode(int coeff, int x, int y, int z);
NODEPTR insertEnd(NODEPTR list, int coeff, int x, int y, int z);
void displayPolynomial(NODEPTR list);
double evaluatePolynomial(NODEPTR list, double x, double y, double z);

// Create a new node
NODEPTR createNode(int coeff, int x, int y, int z)
```

```
{
    NODEPTR p= (NODEPTR)malloc(sizeof(struct node *));
    p->coefficient = coeff;
    p->x_power = x;
    p->y_power = y;
    p->z_power = z;
    p->next = p; // Self-loop for circular nature
    return p;
}

// Insert a node at the end of the SCLL
NODEPTR insertEnd(NODEPTR list, int coeff, int x, int y, int z) {
    NODEPTR p= createNode(coeff, x, y, z);
    if (list == NULL) {
        list = p;
    } else {
        NODEPTR q = list;
        while (q->next != list) {
            q = q->next;
        }
        q->next = p;
        p->next = list;
    }
    return list;
}

// Display the polynomial
void displayPolynomial(NODEPTR list) {
    if (list == NULL) {
        printf("Polynomial is empty.\n");
```

```
        return;
    }
    NODEPTR p= list;
    do {
        printf("%+dx^%dy^%dz^%d ", p->coefficient, p->x_power, p->y_power, p->z_power);
        p = p->next;
    } while (p != list);
    printf("\n");
}

// Evaluate the polynomial for given x, y, and z values
double evaluatePolynomial(NODEPTR list, double x, double y, double z)
{
    double result = 0.0;
    if (list == NULL)
 {
        return result;
    }
NODEPTR q = list;
    do {
        result = result += q->coefficient * pow(x, q->x_power) * pow(y, q->y_power) * pow(z,
q->z_power);
        q = q->next;
    } while (q != list);
    return result;
}

// Main function
int main() {
    NODEPTR poly = NULL;
```

```
    // Represent the polynomial P(x,y,z) = 6x^2y^2z - 4yz^5 + 3x^3yz + 2xy^5z - 2xyz^3
    poly= insertEnd(poly, 6, 2, 2, 1);
    poly = insertEnd(poly, -4, 0, 1, 5);
//    poly = insertEnd(poly, 3, 3, 1, 1);
//  poly = insertEnd(poly, 2, 1, 5, 1);
//  poly= insertEnd(poly, -2, 1, 1, 3);

    // Display the polynomial
    printf("Polynomial: ");
    displayPolynomial(poly);

    // Evaluate the polynomial
    double x, y, z;
    printf("\nEnter values for x, y, z: ");
    scanf("%lf %lf %lf", &x, &y, &z);

    double result = evaluatePolynomial(poly, x, y, z);
    printf("Result of evaluation: %.2lf\n", result);
    return 0;
}
```

Smple output

Enter values for x, y, z: 1 2 3

Polynomial: +6x^2y^2z^1 -4x^0y^1z^5 +3x^3y^1z^1 +2x^1y^5z^1 -2x^1y^1z^3

Result of evaluation: ------

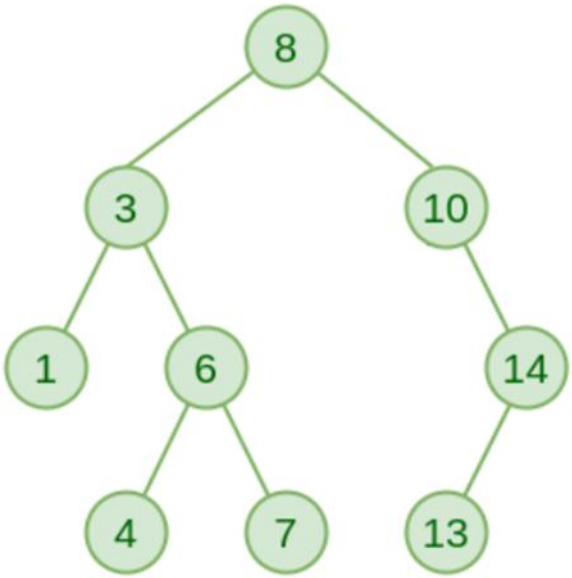
## Program 10 theory

Binary Search Tree is a node-based binary tree data structure which has the following properties:

The left subtree of a node contains only nodes with keys lesser than the node’s key.

The right subtree of a node contains only nodes with keys greater than the node's key.

The left and right subtree each must also be a binary search tree.




**Basic Operations:**

1. Insertion in Binary Search Tree
2. Searching in Binary Search Tree
3. Deletion in Binary Search Tree
4. Binary Search Tree (BST) Traversals – Inorder, Preorder, Post Order
5. Convert a normal BST to Balanced BST

10. Dictionary can be implemented using binary search tree. A binary search tree is a binary tree such that each node stores a key of a dictionary. Key 'k' of a node is always greater than the keys present in its left sub tree. Similarly, key 'k' of a node is always lesser than the keys present in its right sub tree.

Design, Develop and Implement a menu driven Program in C to perform the following operations using Binary Search Tree (BST).
a.Create a dictionary of words
b.Traverse the dictionary in Inorder, Preorder and Post Order
c.Search the dictionary for a given word (KEY) and display the appropriate message

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

// Structure definition for a node in the Binary Search Tree (BST)
struct node {
  char info[20];          // Stores the key (word) in the node
          struct node *left;      // Pointer to the left subtree
         struct node *right;     // Pointer to the right subtree
```

```
};

        // Typedef for convenience in working with node pointers
        typedef struct node *NODEPTR;

        // Function declarations
        NODEPTR maketree(char word[]);          // Create a new tree with a single node
    NODEPTR createtree(char word[]);        // Create a tree (unused function in this code)
        void setleft(NODEPTR p, char word[]);    // Attach a node as the left child
        void setright(NODEPTR p, char word[]);   // Attach a node as the right child
        void intrav(NODEPTR p);                  // Perform in-order traversal
        void pretrav(NODEPTR p);                 // Perform pre-order traversal
        void posttrav(NODEPTR p);                // Perform post-order traversal
        void search(NODEPTR p, char key[]);      // Search for a key in the tree

        // Main function to implement the menu-driven program
    void main() {
       NODEPTR ptree; // Root of the binary search tree
       NODEPTR p, q;  // Temporary pointers for traversal and insertion
       char word[20], key[20]; // Variables to hold input words and keys for searching
       int opt;       // User's menu choice

       do {
          // Display menu options
          printf("\n1->CREATE DICTIONARY  2->TRAVERSE 3->SEARCH  4->EXIT ");
          printf("\nEnter your option:");
          scanf("%d", &opt);

          switch (opt) {
             case 1: // Create dictionary (build BST)
                    printf("\nEnter a word :");
                    scanf("%s", word);
                    ptree = maketree(word); // Initialize the tree with the first word

                    // Keep adding words to the BST until "END" is entered
                    while (strcmp(word, "END") != 0) {
                       printf("\nEnter a word(Type END to stop):");
                       scanf("%s", word);
                       if (strcmp(word, "END") == 0)
                          break; // Stop word entry if "END" is entered

                       p = q = ptree; // Start from the root of the tree

                       // Find the appropriate position for the new word
                       while ((strcmp(word, p->info) != 0) && q != NULL) {
                          p = q; // Keep track of the parent node
                          if (strcmp(word, p->info) < 0) // If word is smaller, move left
                             q = p->left;
                          else                        // If word is larger, move right
                             q = p->right;
```

```
                }

                // Attach the new word as a left or right child
                if (strcmp(word, p->info) < 0)
                    setleft(p, word);
              else if (strcmp(word, p->info) >= 0)
                    setright(p, word);
                }
            printf("\n DICTIONARY CREATED SUCCESSFULLY");
            break;

             case 2: // Traverse the dictionary in different orders
                printf("\n PREORDER TRAVERSAL OF THE DICTIONARY IS:");
                pretrav(ptree); // Pre-order traversal

                printf("\n INORDER TRAVERSAL OF THE DICTIONARY IS:");
                intrav(ptree);  // In-order traversal

                printf("\n POSTORDER TRAVERSAL OF THE DICTIONARY IS:");
                posttrav(ptree); // Post-order traversal
            break;

             case 3: // Search for a word in the dictionary
            printf("\n Enter the key to search in the dictionary:");
            scanf("%s", key);
            search(ptree, key); // Call search function
            break;

         case 4: // Exit the program
            printf("\nEXITING BINARY TREE");
            exit(1); // Exit the program
      }
   } while (opt != 4); // Repeat until the user chooses to exit
}

// Function to create a new tree with a single node
NODEPTR maketree(char w[]) {
        NODEPTR t;
        t = (NODEPTR)malloc(sizeof(struct node)); // Allocate memory for the new node
        if (t == NULL) { // Check for memory allocation failure
            printf("\n Node allocation failed");
            exit(0);
   }
        strcpy(t->info, w); // Copy the word into the node
   t->left = NULL;     // Initialize left and right children as NULL
        t->right = NULL;
   return t;          // Return the created node
    }

    // Attach a node as the left child of the given parent node
```

```
    void setleft(NODEPTR p, char w[]) {
   if (p == NULL) // Check for invalid parent node
            printf("Void Insertion");
        else if (p->left != NULL) // Check if left child already exists
            printf("Invalid Insertion");
        else
            p->left = maketree(w); // Attach the new node as the left child
}

// Attach a node as the right child of the given parent node
    void setright(NODEPTR p, char w[]) {
   if (p == NULL) // Check for invalid parent node
            printf("Void Insertion");
        else if (p->right != NULL) // Check if right child already exists
      printf("Invalid Insertion");
   else
      p->right = maketree(w); // Attach the new node as the right child
    }

    // Perform in-order traversal (left, root, right)
    void intrav(NODEPTR tree) {
        if (tree != NULL) {
            intrav(tree->left);          // Visit left subtree
            printf("%s\t", tree->info);  // Visit root
            intrav(tree->right);         // Visit right subtree
        }
}

    // Perform pre-order traversal (root, left, right)
void pretrav(NODEPTR tree) {
        if (tree != NULL) {
       printf("%s\t", tree->info);  // Visit root
            pretrav(tree->left);         // Visit left subtree
            pretrav(tree->right);        // Visit right subtree
        }
    }

// Perform post-order traversal (left, right, root)
void posttrav(NODEPTR tree) {
   if (tree != NULL) {
            posttrav(tree->left);        // Visit left subtree
            posttrav(tree->right);       // Visit right subtree
            printf("%s\t", tree->info);  // Visit root
   }
}

// Search for a given word (key) in the tree
void search(NODEPTR tree, char key[]) {
   NODEPTR p = tree; // Start searching from the root
```

```
   while (p != NULL && strcmp(key, p->info) != 0) { // Traverse until key is found or
tree ends
      if (strcmp(key, p->info) < 0) // If key is smaller, move to left subtree
         p = p->left;
      else // If key is larger, move to right subtree
         p = p->right;
   }
   if (p != NULL) // Key is found
      printf("\n Key %s is found in the dictionary", key);
   else // Key is not found
      printf("\n Key %s is not found in the dictionary", key);
}


/*Sample output


1->CREATE DICTIONARY  2->TRAVERSE 3->SEARCH  4->EXIT
     Enter your option:1

     Enter a word :table

     Enter a word(Type END to stop):chair

     Enter a word(Type END to stop):lamp

     Enter a word(Type END to stop):laptop

     Enter a word(Type END to stop):mouse

Enter a word(Type END to stop):keyboard

Enter a word(Type END to stop):vase

Enter a word(Type END to stop):water

Enter a word(Type END to stop):cup

Enter a word(Type END to stop):steel

Enter a word(Type END to stop):umbrella

Enter a word(Type END to stop):van

Enter a word(Type END to stop):END

 DICTIONARY CREATED SUCCESSFULLY
1->CREATE DICTIONARY  2->TRAVERSE 3->SEARCH  4->EXIT
Enter your option:2
```

PREORDER TRAVERSAL OF THE DICTIONARY IS:
table chair lamp keyboard cup laptop mouse steel vase umbrella van water

INORDER TRAVERSAL OF THE DICTIONARY IS:
chair cup keyboard lamp laptop mouse steel table umbrella van vase water

POSTORDER TRAVERSAL OF THE DICTIONARY IS:
cup keyboard steel mouse laptop lamp chair van umbrella water vase table

1->CREATE DICTIONARY 2->TRAVERSE 3->SEARCH 4->EXIT
Enter your option:3

Enter the key to search in the dictionary:mouse

Key mouse is found in the dictionary
1->CREATE DICTIONARY 2->TRAVERSE 3->SEARCH 4->EXIT
Enter your option:3

Enter the key to search in the dictionary:cable

Key cable is not found in the dictionary
1->CREATE DICTIONARY 2->TRAVERSE 3->SEARCH 4->EXIT
Enter your option:END
*/