T.Rohinth
230701267
CSE E

Implementation of Single Linked List

Aim:

To implement and execute single linked list

Algorithm:

1. Define the Node Structure

- Create a struct to represent a node with two members: data (to store the value) and next (to point to the next node).

2. Initialize the Linked List

- Create a pointer to the head node and initialize it to NULL.

3. Append a Node

- Create a new node.
- If the list is empty, set the new node as the head.
- Otherwise, traverse to the end of the list and set the next pointer of the last node to the new node.

4. Prepend a Node

- Create a new node.
- Set the next pointer of the new node to the current head.
- Update the head to point to the new node.

5. Delete a Node by Value

- If the list is empty, do nothing.
- If the head node holds the value, update the head to the next node.
- Otherwise, traverse the list to find the node with the given value and update the pointers to remove the node.

6. Search for a Value

- Traverse the list from the head to the end.

- If a node with the given value is found, return true.
- If the end of the list is reached without finding the value, return false.

7. Print the List

- Traverse the list from the head to the end.
- Print the data of each node.

Program:

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

struct node {
   int data;
   struct node *next;
 };
struct node *l=NULL;
struct node *newnode;

void insert_first(){
   newnode=(struct node*)malloc(sizeof(struct node));
   if(newnode!=NULL){
      printf("enter no");
      scanf("%d",&newnode->data);
      if (l!=NULL){
      newnode->next=l;
      l=newnode;
      }
      else
      {
         newnode->next=NULL;
         l=newnode;
      }
```

```
    }

}

void insert_last(){
    newnode=(struct node*)malloc(sizeof(struct node));
    struct node*p;
    if(newnode!=NULL){
        printf("enter no");
        scanf("%d",&newnode->data);
        newnode->next=NULL;
        if (l!=NULL){
            p=l;
            while(p->next!=NULL){
                p=p->next;
            }
            p->next=newnode;

        }
        else
        {
            l=newnode;
        }

    }

}

void insert_afterp(){
    newnode=(struct node*)malloc(sizeof(struct node));
    struct node *p=l;
    int pos,c=1;
    if(newnode!=NULL){
        printf("enter the element");
        scanf("%d",&newnode->data);
        printf("enter the positon");
        scanf("%d",&pos);
```

```
        while(c<pos-1){
            p=p->next;
            c++;
        }
        newnode->next=p->next;
        p->next=newnode;
    }
}

void delete_first(){
    struct node*p=l;
    l=l->next;
    free(p);
}

void delete_last(){
    struct node*p=l;
    struct node*temp;
    while(p->next->next!=NULL){
        p=p->next;
    }
    temp=p->next;
    p->next=NULL;
    free(temp);

}

void delete_afterp(){
    struct node*p=l;
    struct node *temp;
    int pos,c=1;
    printf("enter position ");
    scanf("%d",&pos);
    while(c<pos-1){
        p=p->next;
        c++;
    }
```

```
   temp=p->next;
   p->next=temp->next;
   free(temp);
}

void find(){
   struct node *p=l;
   int pos,c=1;
   printf("enter position ");
   scanf("%d",&pos);
   while(c<pos){
      p=p->next;
      c++;
   }
   printf("The element present at the position %d is %d",pos,p->data);
}

void find_next(){
   struct node *p=l;
   int pos,c=1;
   printf("enter position ");
   scanf("%d",&pos);
   while(c<pos){
      p=p->next;
      c++;
   }
   printf("The element present next to the position %d is %d",pos,p->next->data);
}

void find_previous(){
   struct node *p=l;
   int pos,c=1;
   printf("enter position ");
   scanf("%d",&pos);
   while(c<pos-1){
      p=p->next;
      c++;
```

```
    }
    printf("The element present previous to the position %d is %d",pos,p->data);
}

void isempty(){
    if(l!=NULL)
        printf("List is not empty");
    else
        printf("List is empty");
}

void delete_list(){
    struct node *p=l;
    struct node *temp;
    while (p->next!=NULL){
        temp=p->next;
        free(p);
        p=temp;
    }
    l=NULL;

}



void display(){
    if (l!=NULL){
        struct node*p;
        p=l;
        while(p!=NULL){
            printf("%d",p->data);
            p=p->next;
        }
    }
    else
        printf("List is empty");
}
```

```
void main(){
   int n;
   struct node *l=NULL;
   printf("options\n1.Enter at first\n2.enter at last\n3.insert after p\n4.delete first
element\n5.delete last\n6.delete after p\n7.find\n8.find next\n9.find
previous\n10.is the list empty\n11.delete the list\n12.display\n13.exit\n");
   do{
      printf("\nEnter your option:");
      scanf("%d",&n);
      switch(n){
         case 1:
            insert_first();
            break;
         case 2:
            insert_last();
            break;
         case 3:
            insert_afterp();
            break;
         case 4:
            delete_first();
            break;
         case 5:
            delete_last();
            break;
         case 6:
            delete_afterp();
            break;
         case 7:
            find();
            break;
         case 8:
            find_next();
            break;
```

```
        case 9:
           find_previous();
           break;
        case 10:
           isempty();
           break;
        case 11:
           delete_list();
           break;
        case 12:
           display();
           break;
     }
  }while(n!=13);
}
```

Output:

```
/tmp/JqNNsAOWrb.o
options
1.Enter at first
2.enter at last
3.insert after p
4.delete first element
5.delete last
6.delete after p
7.find
8.find next
9.find previous
10.is the list empty
11.delete the list
12.display
13.exit

Enter your option:2
enter no:10

Enter your option:2
enter no:20
```

Enter your option:2
enter no:30

Enter your option:2
enter no:40

Enter your option:1
enter no:0

Enter your option:3
enter the element:32
enter the positon:2

Enter your option:4

Enter your option:12
10 20 30 40

Enter your option:7
enter position 3
The element present at the position 3 is 30

Enter your option:11

Enter your option:12
List is empty
Enter your option:13

Result:

Thus the code implemented and executed successfully.

IMPLEMENTATION OF DOUBLY LINKED LIST

Aim:

To perform the doubly linked list operation(insertion ,deletion ,searching, display)

Algorithm:

1. Start
2. Create a structure and functions for each operations
3. Declare the variables
4. Create a do-while loop to display the menu and execute operations based on your input until the user chooses to exit
5. Inside the loop display the menu options
6. Prompt the user to enter their choice
7. Use switch statement to perform different operations based on the user's choice and display it.
8. Repeat the loop until the user chooses to exit
9. Exit

Program:

```
#include<stdio.h>
#include<stdlib.h>
// Define a node structure for doubly linked list
struct Node {
int data;
struct Node* prev;
struct Node* next;
};
// Function to create a new node
struct Node* createNode(int data)
{
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = data;
newNode->prev = NULL;
newNode->next = NULL;
return newNode;
}
// Function to insert a node at the beginning of the list
void insertAtBeginning(struct Node** head_ref, int data)
{
```

```
 struct Node* newNode = createNode(data);
 if (*head_ref == NULL)
 {
 *head_ref = newNode;
 return;
}
newNode->next = *head_ref;
 (*head_ref)->prev = newNode;
 *head_ref = newNode;
 }
 // Function to insert a node at the end of the list
 void insertAtEnd(struct Node** head_ref, int data)
{
struct Node* newNode = createNode(data);
 struct Node* temp = *head_ref;
 if (*head_ref == NULL)
 {
 *head_ref = newNode;
 return;
}
 while (temp->next != NULL)
{
 temp = temp->next;
}
 temp->next = newNode;
 newNode->prev = temp;
 }
 // Function to insert a node at a specific position
 void insertAtPosition(struct Node** head_ref, int data, int position)
 {
 if (position < 1)
 {
 printf("Invalid position\n");
 return;
```

```
 }
 if (position == 1)
 {
insertAtBeginning(head_ref, data);
 return;
 }
 struct Node* newNode = createNode(data);
 struct Node* temp = *head_ref;
 for (int i = 1; i < position - 1 && temp != NULL; i++)
{
 temp = temp->next;
}
 if (temp == NULL)
 {
 printf("Position out of range\n");
 return;
 }
 newNode->next = temp->next;
 if (temp->next != NULL)
{
 temp->next->prev = newNode;
 }
 temp->next = newNode;
 newNode->prev = temp;
 }
 // Function to delete the first node
void deleteFirstNode(struct Node** head_ref)
{
if (*head_ref == NULL)
 {
 printf("List is empty\n");
 return;
}
struct Node* temp = *head_ref;
```

```
 *head_ref = temp->next;
 if (*head_ref != NULL)
{
(*head_ref)->prev = NULL;
 }
 free(temp);
 }
 // Function to delete the last node
 void deleteLastNode(struct Node** head_ref)
 {
 if (*head_ref == NULL)
 {
 printf("List is empty\n");
 return;
}
 struct Node* temp = *head_ref;
 while (temp->next != NULL)
 { temp = temp->next;
 }
 if (temp->prev != NULL)
{
 temp->prev->next = NULL;
 }
 else
{
 *head_ref = NULL;
 }
 free(temp);
 }
 // Function to delete a node at a specific position
void deleteAtPosition(struct Node** head_ref, int position)
 {
 if (*head_ref == NULL)
{
```

```
 printf("List is empty\n");
 return;
}
 if (position < 1)
 { printf("Invalid position\n");
return;
 }
 if (position == 1)
 {
deleteFirstNode(head_ref);
 return;
 }
 struct Node* temp = *head_ref;
 for (int i = 1; i < position && temp != NULL; i++)
{
temp = temp->next;
 }
 if (temp == NULL)
{
 printf("Position out of range\n");
 return;
 }
 if (temp->next != NULL)
{
 temp->next->prev = temp->prev;
 }
 temp->prev->next = temp->next;
free(temp);
 }
 // Function to search for a node with a given value
struct Node* searchNode(struct Node* head, int key)
 {
 struct Node* temp = head;
while (temp != NULL)
```

```
{
 if (temp->data == key)
 {
 return temp;
}
 temp = temp->next;
}
 return NULL;
 }
 // Function to display the doubly linked list
void displayList(struct Node* head)
{
 struct Node* temp = head;
 while (temp != NULL)
 {
 printf("%d ", temp->data);
 temp = temp->next;
 }
 printf("\n");
 }
 int main()
{
 struct Node* head = NULL;
int choice, data, position, key;
do
 {
 printf("\n1. Insert at Beginning\n");
 printf("2. Insert at End\n");
 printf("3. Insert at Position\n");
 printf("4. Delete First Node\n");
printf("5. Delete Last Node\n");
 printf("6. Delete at Position\n");
 printf("7. Search for a Node\n");
 printf("8. Display List\n");
```

```
 printf("9. Exit\n");
 printf("Enter your choice: ");
 scanf("%d", &choice);
 switch(choice)
{
case 1:
            printf("Enter data to insert at beginning: ");
            scanf("%d", &data);
           insertAtBeginning(&head, data); break;
case 2:
            printf("Enter data to insert at end: ");
            scanf("%d", &data);
            insertAtEnd(&head, data);
           break;
case 3:
            printf("Enter data to insert: ");
            scanf("%d", &data);
            printf("Enter position to insert at: ");
           scanf("%d", &position);
            insertAtPosition(&head, data, position);
           break;
case 4:
           deleteFirstNode(&head);
            break;
case 5:
            deleteLastNode(&head);
           break;
 case 6:
            printf("Enter position to delete: ");
            scanf("%d", &position);
            deleteAtPosition(&head, position);
            break;
case 7:
            printf("Enter value to search: ");
```

```
            scanf("%d", &key);
            struct Node* result = searchNode(head, key);
            if (result != NULL)
            {
            printf("%d found in the list.\n", key);
            }
            else
            {
            printf("%d not found in the list.\n", key);
            }
            break;
case 8:
            printf("Doubly linked list: ");
            displayList(head);
            break;
case 9:
            printf("Exiting...\n");
            break;
default:
 printf("Invalid choice\n");
 }
 } while (choice != 9);
return 0;
 }
```

Output:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete First Node
5. Delete Last Node
6. Delete at Position
7. Search for a Node
8. Display List
9. Exit

Enter your choice: 2

Enter data to insert at end: 1

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete First Node
5. Delete Last Node
6. Delete at Position
7. Search for a Node
8. Display List
9. Exit

Enter your choice: 2

Enter data to insert at end: 2

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete First Node
5. Delete Last Node
6. Delete at Position
7. Search for a Node
8. Display List
9. Exit

Enter your choice: 2

Enter data to insert at end: 3

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete First Node
5. Delete Last Node
6. Delete at Position
7. Search for a Node

8. Display List

9. Exit

Enter your choice: 8

Doubly linked list: 1 2 3

1. Insert at Beginning

2. Insert at End

3. Insert at Position

4. Delete First Node

5. Delete Last Node

6. Delete at Position

7. Search for a Node

8. Display List

9. Exit

Enter your choice: 6

Enter position to delete: 3

1. Insert at Beginning

2. Insert at End

3. Insert at Position

4. Delete First Node

5. Delete Last Node

6. Delete at Position

7. Search for a Node

8. Display List

9. Exit

Enter your choice: 8

Doubly linked list: 1 2

1. Insert at Beginning

2. Insert at End

3. Insert at Position

4. Delete First Node

5. Delete Last Node

6. Delete at Position

7. Search for a Node

8. Display List

9. Exit

Enter your choice: 9

Exiting...

=== Code Execution Successful ===

Result:

Thus , the implementation of doubly linked list is executed successfully..

EX NO:3 APPLICATION OF SINGLY LINKED LIST(POLYNOMIAL)

AIM:

To write a C program to perform addition, subtraction, and multiplication of two polynomials using linked lists.

ALGORITHM:

Step 1: Start.

Step 2: Define a structure for the polynomial node, containing the coefficient, power, and a pointer to the next node.

Step 3: Implement a function to create a new node for the polynomial with the given coefficient and power.

Step 4: Implement a function to create a polynomial list with 'n' nodes by repeatedly calling the node creation function.

Step 5: Implement a function to display the polynomial.

Step 6: Implement a function to add two polynomials.

Step 7: Implement a function to subtract two polynomials.

Step 8: Implement a function to multiply two polynomials.

Step 9: Implement a function to combine like terms in a polynomial (used after multiplication).

Step 10: In the main function, create two polynomials by reading the number of terms and their coefficients and powers.

Step 11: Provide a menu to perform addition, subtraction, and multiplication of the polynomials.

Step 12: Display the result of the chosen operation.

Step 13: Stop.

PROGRAM:

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

struct node {
    int coeff;
    int pow;
    struct node *next;
};

struct node* create_node() {
    struct node *p = malloc(sizeof(struct node));
    if (p != NULL) {
        printf("Enter the coefficient: ");
        scanf("%d", &p->coeff);
        printf("Enter the power of x: ");
        scanf("%d", &p->pow);
        p->next = NULL;
    }
    return p;
}

struct node* create_list(int n) {
    struct node *temp, *L1, *new;
```

```
    L1 = malloc(sizeof(struct node));
    temp = L1;
    while (n--) {
        new = create_node();
        temp->next = new;
        temp = new;
    }
    temp->next = NULL;
    return L1;
}

void display(struct node *L) {
    struct node *p = L->next;
    while (p != NULL) {
        printf("%dx^%d", p->coeff, p->pow);
        if (p->next != NULL) {
            printf(" + ");
        }
        p = p->next;
    }
    printf("\n");
}

struct node* addition(struct node *L1, struct node *L2) {
    struct node *p1 = L1->next, *p2 = L2->next, *new, *L3, *temp;
```

```
L3 = malloc(sizeof(struct node));
L3->next = NULL;
temp = L3;
while (p1 != NULL && p2 != NULL) {
    new = malloc(sizeof(struct node));
    if (p1->pow == p2->pow) {
        new->coeff = p1->coeff + p2->coeff;
        new->pow = p1->pow;
        p1 = p1->next;
        p2 = p2->next;
    } else if (p1->pow > p2->pow) {
        new->coeff = p1->coeff;
        new->pow = p1->pow;
        p1 = p1->next;
    } else {
        new->coeff = p2->coeff;
        new->pow = p2->pow;
        p2 = p2->next;
    }
    temp->next = new;
    new->next = NULL;
    temp = temp->next;
}
while (p1 != NULL) {
    new = malloc(sizeof(struct node));
```

```
        new->coeff = p1->coeff;
        new->pow = p1->pow;
        temp->next = new;
        new->next = NULL;
        temp = temp->next;
        p1 = p1->next;
    }
    while (p2 != NULL) {
        new = malloc(sizeof(struct node));
        new->coeff = p2->coeff;
        new->pow = p2->pow;
        temp->next = new;
        new->next = NULL;
        temp = temp->next;
        p2 = p2->next;
    }
    return L3;
}

struct node* subtraction(struct node *L1, struct node *L2) {
    struct node *p1 = L1->next, *p2 = L2->next, *new, *L3, *temp;
    L3 = malloc(sizeof(struct node));
    L3->next = NULL;
    temp = L3;
    while (p1 != NULL && p2 != NULL) {
```

```
    new = malloc(sizeof(struct node));
    if (p1->pow == p2->pow) {
        new->coeff = p1->coeff - p2->coeff;
        new->pow = p1->pow;
        p1 = p1->next;
        p2 = p2->next;
    } else if (p1->pow > p2->pow) {
        new->coeff = p1->coeff;
        new->pow = p1->pow;
        p1 = p1->next;
    } else {
        new->coeff = -p2->coeff;
        new->pow = p2->pow;
        p2 = p2->next;
    }
    temp->next = new;
    new->next = NULL;
    temp = temp->next;
}
while (p1 != NULL) {
    new = malloc(sizeof(struct node));
    new->coeff = p1->coeff;
    new->pow = p1->pow;
    temp->next = new;
    new->next = NULL;
```

```
        temp = temp->next;
        p1 = p1->next;
    }
    while (p2 != NULL) {
        new = malloc(sizeof(struct node));
        new->coeff = -p2->coeff;
        new->pow = p2->pow;
        temp->next = new;
        new->next = NULL;
        temp = temp->next;
        p2 = p2->next;
    }
    return L3;
}

struct node* multiplication(struct node *L1, struct node *L2) {
    struct node *p1 = L1->next, *p2, *new, *L3, *temp;
    L3 = malloc(sizeof(struct node));
    L3->next = NULL;
    temp = L3;
    while (p1 != NULL) {
        p2 = L2->next;
        while (p2 != NULL) {
            new = malloc(sizeof(struct node));
            new->coeff = p1->coeff * p2->coeff;
```

```
            new->pow = p1->pow + p2->pow;
            temp->next = new;
            new->next = NULL;
            temp = temp->next;
            p2 = p2->next;
        }
        p1 = p1->next;
    }
    return L3;
}

struct node* add(struct node *L3) {
    struct node *temp = L3->next, *p, *ref;
    while (temp != NULL) {
        p = temp->next;
        ref = temp;
        while (p != NULL) {
            if (temp->pow == p->pow) {
                temp->coeff += p->coeff;
                ref->next = p->next;
                free(p);
                p = ref->next;
            } else {
                ref = ref->next;
                p = p->next;
```

```
            }
        }
        temp = temp->next;
    }
    return L3;
}

void free_list(struct node *L) {
    struct node *temp;
    while (L != NULL) {
        temp = L;
        L = L->next;
        free(temp);
    }
}

int main() {
    struct node *l1, *l2, *la, *ls, *lm;
    int n, option;
    printf("Enter the polynomials in Descending order only\n");
    printf("NO. of terms in polynomial 1: ");
    scanf("%d", &n);
    l1 = create_list(n);
    printf("NO. of terms in polynomial 2: ");
    scanf("%d", &n);
```

```
l2 = create_list(n);

do {
    printf("OPERATIONS:\n");
    printf("1. Addition\n");
    printf("2. Subtraction\n");
    printf("3. Multiplication\n");
    printf("4. Exit\n");
    printf("Enter your option: ");
    scanf("%d", &option);

    switch (option) {
        case 1:
            la = addition(l1, l2);
            printf("The Sum of the two polynomials is: ");
            display(la);
            free_list(la);
            break;

        case 2:
            ls = subtraction(l1, l2);
            printf("The difference of the two polynomials is: ");
            display(ls);
            free_list(ls);
            break;
```

```
            case 3:
                lm = multiplication(l1, l2);
                lm = add(lm);
                printf("The product of the two polynomials is: ");
                display(lm);
                free_list(lm);
                break;

            case 4:
                printf("Program ended.\n");
                break;

            default:
                printf("Invalid option. Please try again.\n");
                break;
        }
    } while (option != 4);

    free_list(l1);
    free_list(l2);

    return 0;
}
```

OUTPUT:

Enter the polynomials in Descending order only

NO. of terms in polynomial 1: 3

Enter the coefficient: 4

Enter the power of x: 3

Enter the coefficient: 3

Enter the power of x: 2

Enter the coefficient: 2

Enter the power of x: 1

NO. of terms in polynomial 2: 3

Enter the coefficient: 1

Enter the power of x: 3

Enter the coefficient: 2

Enter the power of x: 2

Enter the coefficient: 3

Enter the power of x: 1

OPERATIONS:

1. Addition

2. Subtraction

3. Multiplication

4. Exit

Enter your option: 1

The Sum of the two polynomials is: 5x^3 + 5x^2 + 5x^1

OPERATIONS:

1. Addition

2. Subtraction

3. Multiplication

4. Exit

Enter your option: 2

The difference of the two polynomials is: 3x^3 + 1x^2 - 1x^1

OPERATIONS:

1. Addition

2. Subtraction

3. Multiplication

4. Exit

Enter your option: 3

The product of the two polynomials is: 4x^6 + 14x^5 + 20x^4 + 11x^3 + 6x^2

OPERATIONS:

1. Addition

2. Subtraction

3. Multiplication

4. Exit

Enter your option: 4

Program ended.

RESULT:

Thus, the C program to perform infix to postfix conversion in stack was executed successfully.

Implementation of stack using array and linked lists

Aim:

The aim of the provided code is to implement a stack using Array and Linked List implementation and execute operations on stack.

Algorithm:

1.Start
2. Create a structure and functions for the given operations
3.Initialize stack array with capacity and top=-1
4.To push an element into a stack read the data to be pushed. If the top is equal to capacity-1 display stack overflow. Otherwise increment the top and push the data onto stack at index top
5.To pop an element from a stack if the top is equal to -1 display as stack underflow. Otherwise pop data from stack at index top the decrement the top and display the popped data
6. To return the top most element from a stack if the top is equal to -1 display stack is empty. Otherwise display data at index top
7.After these operations display all elements in stack from top to 0
8.End

Program:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
 int data;
 struct Node* next;
};
struct StackLL {
 struct Node* top;
};
struct StackArray {
 int* array;
 int top;
 int capacity;
};
struct StackLL* createStackLL() {
 struct StackLL* stack = (struct StackLL*)malloc(sizeof(struct StackLL));
 stack->top = NULL;
 return stack;
}
struct StackArray* createStackArray(int capacity) {
 struct StackArray* stack = (struct StackArray*)malloc(sizeof(struct StackArray));
 stack->capacity = capacity;
 stack->top = -1;
 stack->array = (int*)malloc(stack->capacity * sizeof(int));
 return stack;
}
```

```
int isEmptyLL(struct StackLL* stack) {
 return stack->top == NULL;
}
int isEmptyArray(struct StackArray* stack) {
 return stack->top == -1;
}
void pushLL(struct StackLL* stack, int data) {
 struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
 newNode->data = data;
 newNode->next = stack->top;
 stack->top = newNode;
}
void pushArray(struct StackArray* stack, int data) {
 if (stack->top == stack->capacity - 1) {
 printf("Stack Overflow\n");
 return;
 }
 stack->array[++stack->top] = data;
}
int popLL(struct StackLL* stack) {
 if (isEmptyLL(stack)) {
 printf("Stack Underflow\n");
 return -1;
 }
 struct Node* temp = stack->top;
 int data = temp->data;
 stack->top = stack->top->next;
 free(temp);
 return data;
}
int popArray(struct StackArray* stack) {
 if (isEmptyArray(stack)) {
 printf("Stack Underflow\n");
 return -1;
 }
 return stack->array[stack->top--];
}
int peekLL(struct StackLL* stack) {
 if (isEmptyLL(stack)) {
 printf("Stack is empty\n");
 return -1;
 }
 return stack->top->data;
}
int peekArray(struct StackArray* stack) {
 if (isEmptyArray(stack)) {
 printf("Stack is empty\n");
 return -1;
 }
```

```
 return stack->array[stack->top];
}
void displayLL(struct StackLL* stack) {
 if (isEmptyLL(stack)) {
 printf("Stack is empty\n");
 return;
 }
 struct Node* temp = stack->top;
 printf("Elements in stack: ");
 while (temp != NULL) {
 printf("%d ", temp->data);
 temp = temp->next;
 }
 printf("\n");
}
void displayArray(struct StackArray* stack) {
 if (isEmptyArray(stack)) {
 printf("Stack is empty\n");
 return;
 }
 printf("Elements in stack: ");
 for (int i = stack->top; i >= 0; i--) {
 printf("%d ", stack->array[i]);
 }
 printf("\n");
}
int main() {
 struct StackLL* stackLL = createStackLL();
 pushLL(stackLL, 1);
 pushLL(stackLL, 2);
 pushLL(stackLL, 3);
 displayLL(stackLL);
 printf("Top element: %d\n", peekLL(stackLL));
 printf("Popped element: %d\n", popLL(stackLL));
 displayLL(stackLL);
 struct StackArray* stackArray = createStackArray(5);
 pushArray(stackArray, 4);
 pushArray(stackArray, 5);
 pushArray(stackArray, 6);
 displayArray(stackArray);
 printf("Top element: %d\n", peekArray(stackArray));
 printf("Popped element: %d\n", popArray(stackArray));
 displayArray(stackArray);
 return 0;
}
```

Output:

Elements in stack: 3 2 1

Top element: 3
Popped element: 3
Elements in stack: 2 1
Elements in stack: 6 5 4
Top element: 6
Popped element: 6
Elements in stack: 5 4

Result:
The output is verified successfully for the above program.

EX NO:05 APPLICATIONS OF STACK

INFIX TO POSTFIX CONVERSION

AIM:

To write a C program to perform infix to postfix conversion using stack.

ALGORITHM:

Step 1: Start

Step 2: Create an empty stack.

Step 3: Get a string(expression) from the user.

Step 4: Traverse the string.

Step 5: If the element is an operand, place it in the output string.

Step 6: If the element is an operator and has higher precedence than the operator in the stack, push it into the stack.

Step 7: If the operator has lower or equal precedence than the operator in stack, pop until stack becomes empty.

Step 8: If it is a left parenthesis, push it into the stack.

Step 9: If it is a right parenthesis, pop all the operators from the stack till it encounters the left parenthesis.

Step 10: After the infix expression has been read, pop all operators from the stack and append them to the output string.

Step 11: Print the output string which is the postfix expression of the given infix expression.

Step 10: Stop

PROGRAM:

```
#include <stdio.h>
#include<string.h>
#define SIZE 500

int precedence(char c)
{
   if(c=='^')
   return 3;
   else if (c=='*'||c=='/'||c=='%')
   return 2;
   else if(c=='+'||c=='-')
   return 1;
   else
   return 0;
}

char associativity(char c)
{
   if(c=='^')
   return 'r';
   else
   return 'l';
}

int valid(char c)
```

```
{
    if(c=='+'||c=='-'||c=='*'||c=='/'||c=='^'||'%')
    return 1;
    else
    return 0;
}

void main() {
    int len,prec1,prec2,top=-1,index=-1;
    int flag=0;
    char exp[SIZE];
    char stack[SIZE];
    char postfix[SIZE];
    printf("enter an expresion:\n");
    scanf("%[^\n]s",exp);
    len=strlen(exp);
    for(int i=0;i<len;i++)
    {

        if((exp[i]>='a' && exp[i]<='z')||(exp[i]>='A' && exp[i]<='Z')||(exp[i]>='0'
&& exp[i]<='9'))
        {
            postfix[++index]=exp[i];
        }
         else if(exp[i]=='(')
        {
            stack[++top]=exp[i];
        }
```

```
else if(exp[i]==')')
{
   while(top>=0&&stack[top]!='(')
   {
      postfix[++index]=stack[top--];
   }
   top--;
}
else if(exp[i]==' ')
{
   continue;
}
else if(valid(exp[i]))
{
   prec1=precedence(exp[i]);
   prec2=precedence(stack[top]);
      while(top>=0 && ((prec1<=prec2) && associativity(exp[i])=='l'))
      {
         postfix[++index]=stack[top--];
      }
   stack[++top]=exp[i];
}
else
{
   printf("invalid expression");
   flag=1;
   break;
```

```
        }

    }

   while(top>=0)
    {
       postfix[++index]=stack[top--];
    }

    if (flag==0)
    {
         printf("\npostfix expression: ");
         printf("%s\n",postfix);
    }

}
```

OUTPUT:

enter an expression:

a+(b*c)-d

postfix expression: abc*+d-

RESULT:

Thus, the C program to perform infix to postfix conversion using stack was executed successfully.

EVALUATING ARITHMETIC EXPRESSION USING STACK

Aim:

To evaluate arithmetic expression using stack .

Algorithm:

1. Start
2. Create an empty stack to hold operands.
3. Initialize a variable top to -1 which represents the top of the stack
4. Read the input from user
5. Iterate through each character in the expression

6. If the character is a digit, convert the character to its integer value and push the

integer into stack.

7. if the character is an operator, pop the top two operands from the stack and

perform the corresponding operation.

8. Get the result and display it

9.End.

Program:

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

```
#include <ctype.h>
#include <string.h>

#define MAX 100

typedef struct Stack {
    int top;
    int items[MAX];
} Stack;

void initStack(Stack* s) {
    s->top = -1;
}

int isFull(Stack* s) {
```

```
    return s->top == MAX - 1;
}

int isEmpty(Stack* s) {
    return s->top == -1;
}

void push(Stack* s, int item) {
    if (isFull(s)) {
        printf("Stack overflow\n");
        return;
    }
    s->items[++s->top] = item;
}
```

```
int pop(Stack* s) {
    if (isEmpty(s)) {
        printf("Stack underflow\n");
        exit(1);
    }
    return s->items[s->top--];
}

int peek(Stack* s) {
    if (isEmpty(s)) {
        printf("Stack is empty\n");
        exit(1);
    }
    return s->items[s->top];
}
```

```
int precedence(char op) {
    switch (op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
    }
    return 0;
}
```

```
int isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^';
}

void infixToPostfix(char* infix, char* postfix) {
    Stack s;
    initStack(&s);
    int i = 0, k = 0;
    while (infix[i]) {
        if (isdigit(infix[i])) {
            postfix[k++] = infix[i];
        } else if (infix[i] == '(') {
            push(&s, infix[i]);
```

```
        } else if (infix[i] == ')') {
            while (!isEmpty(&s) &&
peek(&s) != '(') {
                postfix[k++] = pop(&s);
            }
            pop(&s); // pop '('
        } else if (isOperator(infix[i])) {
            while (!isEmpty(&s) &&
precedence(peek(&s)) >=
precedence(infix[i])) {
                postfix[k++] = pop(&s);
            }
            push(&s, infix[i]);
        }
        i++;
```

```
    }
    while (!isEmpty(&s)) {
        postfix[k++] = pop(&s);
    }
    postfix[k] = '\0';
}

int evaluatePostfix(char* postfix) {
    Stack s;
    initStack(&s);
    int i = 0;
    while (postfix[i]) {
        if (isdigit(postfix[i])) {
            push(&s, postfix[i] - '0');
```

```
        } else if
(isOperator(postfix[i])) {
            int val1 = pop(&s);
            int val2 = pop(&s);
            switch (postfix[i]) {
                case '+':
                    push(&s, val2 + val1);
                    break;
                case '-':
                    push(&s, val2 - val1);
                    break;
                case '*':
                    push(&s, val2 * val1);
                    break;
                case '/':
```

```
                push(&s, val2 / val1);
                break;
            case '^':
                push(&s,
(int)pow(val2, val1));
                break;
        }
    }
    i++;
  }
  return pop(&s);
}

int main() {
  char infix[MAX], postfix[MAX];
```

```
    printf("Enter an infix
expression: ");
    scanf("%s", infix);

    infixToPostfix(infix, postfix);
    printf("Postfix expression:
%s\n", postfix);

    int result =
evaluatePostfix(postfix);
    printf("Result of evaluation:
%d\n", result);

    return 0;
}
```

Output:

```
Enter the postfix expression :
abc+*d*
Enter the value of a : 2
Enter the value of b : 3
Enter the value of c : 4
Enter the value of d : 5
The result is 70
```

Result:

Thus, the C program to evaluate an arithmetic expression using stack was executed successfully.

AIM:

To Implement a queue using an array and linked list in C programming language.

ALGORITHM:

1. Start

2. Define a structure Node with a data field and a pointer to the next node

3. Declare variables front, rear, and l to represent the front and rear pointers, and the size of the queue respectively

4. Initialize front and rear to NULL

5. Display the menu with options and prompt the user to enter the size of the queue 6. Repeat until the user chooses to exit:

a. Prompt the user to select an option

b. If option is 1 (Enqueue):

i. Allocate memory for a new node

ii. Prompt the user to enter a value to be added

iii. If the queue is empty, set both front and rear to the new node iv. Otherwise, add the new node to the rear of the queue and update rear

c. If option is 2 (Dequeue):

i. Check if the queue is empty, if yes, display "Queue is empty"

ii. Otherwise, display the value of the front node

iii. Move front to the next node and free the memory of the previous front node iv. If front becomes NULL after deletion, set rear to NULL as well

d. If option is 3 (Display):

i. Traverse the queue from front to rear and display the data of each node

e. If option is 4 (Exit), terminate the loop

6. End

PROGRAM:

ARRAY:

```
#include<stdio.h>

int q[100];
int front = -1, rear = -1, l;

void enqueue() {
    int a;
    if (rear >= l - 1)
        printf("Queue Overflow\n");
    else {
        printf("Enter the value to be added : ");
        scanf("%d", &a);
        rear++;
        q[rear] = a;
        if (front == -1) {
            front++;
        }
    }
}

void dequeue() {
    if (front == -1) {
        printf("Queue Underflow\n");
    } else {
        printf("The deleted element is %d\n", q[front]);
        if (front == rear) {
```

```
            front = -1;
            rear = -1;
        } else {
            front++;
        }
    }
}

void display() {
    for (int i = front; i < rear + 1; i++) {
        printf("%d ", q[i]);
    }
    printf("\n");
}

int main() {
    int n;
    printf("1.Enqueue\n2.Dequeue\n3.Display\n4.Exit\n");
    printf("Enter the size of the queue : ");
    scanf("%d", &l);
    do {
        printf("Enter your option: ");
        scanf("%d", &n);
        switch (n) {
            case 1:
                enqueue();
                break;
            case 2:
                dequeue();
                break;
```

```
            case 3:
                display();
                break;
        }
    } while (n != 4);
    return 0;
}
```

LINKED LIST:

```
#include<stdio.h>
#include<stdlib.h>
struct node {
    int data;
    struct node *next;
};
struct node *rear = NULL,*front=NULL;
struct node *p,*newnode;
void enqueue() {
    newnode = (struct node*)malloc(sizeof(struct node));
    if(newnode != NULL) {
        printf("Enter the value to be added : ");
        scanf("%d", &newnode->data);
        if (rear==NULL){
            front=newnode;
            rear=newnode;
            front->next=NULL;
            rear->next=NULL;
        }
        else{
```

```
            newnode->next=NULL;
            rear->next=newnode;
            rear=rear->next;
        }
    }
}
void dequeue() {
    if(front != NULL) {
        struct node *temp = front;
        front = front->next;
        printf("The deleted element is %d\n",temp->data);
        free(temp);
    } else {
        printf("Queue Underflow\n");
    }
}
void display() {
    if (front != NULL) {
        p = front;
        printf("Queue elements: ");
        while(p != NULL) {
            printf("%d ", p->data);
            p = p->next;
        }
        printf("\n");
    } else {
        printf("Stack is empty\n");
    }
}
void main() {
```

```
    int n;
    printf("1.Enqueue\n2.Dequeue\n3.Display\n");
    do {
        printf("Enter your option: ");
        scanf("%d", &n);
        switch(n) {
            case 1:
                enqueue();
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
        }
    } while(n != 4);
}
```

OUTPUT:

```
Enter the size of the queue: 5

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your option: 1
Enter the value to enqueue: 10
```

Element 10 enqueued successfully.

Menu:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your option: 1

Enter the value to enqueue: 20

Element 20 enqueued successfully.

Menu:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your option: 3

Queue elements: 10 20

Menu:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your option: 2

Dequeued element: 10

Menu:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your option: 3

Queue elements: 20

Menu:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your option: 4

Exiting...

RESULT:

This output reflects the complete interaction with the queue program, including user input prompts and corresponding output messages.

Aim:

To Implement Binary Search Tree and Perform Insert , Delete , Search , Display Operations.

Algorithm:

1. Initialize Tree: Start with an empty tree (tree = NULL).

2. Insert Node: Add a new node to the tree, placing it in the correct position based on its value.

3. Delete Node: Remove a node, adjusting the tree to maintain order, replacing it with the appropriate child or subtree.

4. Find Minimum: Traverse the leftmost path to find the smallest value in the tree.

5. Find Maximum: Traverse the rightmost path to find the largest value in the tree.

6. Find Node: Search the tree recursively to locate a node with a specific value.

7. Inorder Traversal: Visit left subtree, current node, and right subtree in sequence.

8. Preorder Traversal: Visit current node, then left and right subtrees.

9. Postorder Traversal: Visit left and right subtrees, then current node.

10. User Interface: Present a menu to the user for performing these operations and process their choices in a loop.

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *left;
   struct node *right;
};
typedef struct node Node;
Node *insert(Node *tree,int e);
Node *delete(Node *tree,int e);
Node *find_min(Node *tree);
Node *find_max(Node *tree);
Node *find(Node *tree,int e);
void inorder(Node *tree);
void preorder(Node *tree);
void postorder(Node *tree);
int main()
{
   Node *tree=NULL;
   int ch;
```

```
   do{
      printf("1.Insert\n2.Delete\n3.Find\n4.Find Min\n5.Find
Max\n6.Inorder\n7.Preorder\n8.Postorder\n9.Exit\n");
      printf("Enter your choice:");
      scanf("%d",&ch);
      switch(ch)
      {
         case 1:
            int e;
            printf("Enter the element to insert:");
            scanf("%d",&e);
            tree=insert(tree,e);
            break;
         case 2:
            int d;
            printf("Enter the element to delete:");
            scanf("%d",&d);
            tree=delete(tree,d);
            break ;
         case 3:
            int f;
            printf("Enter element to find:");
            scanf("%d",&f);
            Node *p = find(tree, f);
            if (p!= NULL)
               printf("Element is found - %d\n", p->data);
            else
               printf("Element not found in the tree\n");
            break;
         case 4:
            printf("To find Minimum element in the tree\n");
            Node *q = find_min(tree);
            if (q != NULL)
               printf("Minimum value: %d\n", q->data);
            else
               printf("Tree is empty.\n");
```

```
            break;
        case 5:
            printf("To find Maximum element in the tree\n");
            Node *r=find_max(tree);
            if (r != NULL)
                printf("Mamimum value: %d\n", r->data);
            else
                printf("Tree is empty.\n");
            break;
        case 6:
            printf("The inorder traversal is:");
            inorder(tree);
            printf("\n");
            break;
        case 7:
            printf("The preorder traversal is:");
            preorder(tree);
            printf("\n");
            break;
        case 8:
            printf("The postorder traversal is:");
            postorder(tree);
            printf("\n");
            break;
        case 9:
            printf("Exit\n");
            exit(0);
            break;
      }
   }while(ch<=9 && ch>=1);

}
Node *insert(Node *tree,int e)
{
   Node*newnode=malloc(sizeof(Node));
   if(tree==NULL)
```

```
    {
        newnode->data=e;
        newnode->left=NULL;
        newnode->right=NULL;
        tree=newnode;
    }
    else if(e<tree->data)
    {
        tree->left=insert(tree->left,e);
    }
    else if(e>tree->data)
    {
        tree->right=insert(tree->right,e);
    }
    return tree;
}
Node *delete(Node *tree,int e)
{
    Node *tempnode=malloc(sizeof(Node));
    if(e<tree->data)
    {
        tree->left=delete(tree->left,e);
    }
    else if(e>tree->data)
    {
        tree->right=delete(tree->right,e);
    }
    else if(tree->left && tree->right)
    {
        tempnode=find_min(tree->right);
        tree->data=tempnode->data;
        tree->right=delete(tree->right,tree->data);
    }
    else
    {
        tempnode=tree;
```

```
        if(tree->left==NULL)
        {
            tree=tree->right;
        }
        else if(tree->right==NULL)
        {
            tree=tree->left;
        }
        free(tempnode);
    }
    return tree;
}
Node *find_min(Node *tree)
{
    if(tree!=NULL)
    {
        if(tree->left==NULL)
            return tree;
        else
            return find_min(tree->left);
    }
    return NULL;
}
Node *find_max(Node *tree)
{
    if(tree!=NULL)
    {
        if(tree->right==NULL)
        {
            return tree;
        }
        else
            return find_max(tree->right);
    }
    return NULL;
}
```

```
Node *find(Node *tree,int e)
{
   if(tree==NULL)
      return NULL;
   else if(e<tree->data)
      return find(tree->left,e);
   else if(e>tree->data)
      return find(tree->right,e);
   else
      return tree;
}
void inorder(Node *tree)
{
   if(tree!=NULL)
   {
      inorder(tree->left);
      printf("%d",tree->data);
      inorder(tree->right);
   }
}
void preorder(Node *tree)
{
   if(tree!=NULL)
   {
      printf("%d",tree->data);
      preorder(tree->left);
      preorder(tree->right);
   }
}
void postorder(Node *tree)
{
   if(tree!=NULL)
   {
      postorder(tree->left);
      postorder(tree->right);
      printf("%d",tree->data);
```

```
    }
}
```

Output:

```
1.Insert
2.Delete
3.Find
4.Find Min
5.Find Max
6.Inorder
7.Preorder
8.Postorder
9.Exit
Enter your choice: 1
Enter the element to insert: 10

1.Insert
2.Delete
3.Find
4.Find Min
5.Find Max
6.Inorder
7.Preorder
8.Postorder
9.Exit
Enter your choice: 1
Enter the element to insert: 5

1.Insert
2.Delete
3.Find
4.Find Min
5.Find Max
6.Inorder
7.Preorder
8.Postorder
9.Exit
```

Enter your choice: 1

Enter the element to insert: 20

1.Insert

2.Delete

3.Find

4.Find Min

5.Find Max

6.Inorder

7.Preorder

8.Postorder

9.Exit

Enter your choice: 6

The inorder traversal is: 5 10 20

1.Insert

2.Delete

3.Find

4.Find Min

5.Find Max

6.Inorder

7.Preorder

8.Postorder

9.Exit

Enter your choice: 4

To find Minimum element in the tree

Minimum value: 5

1.Insert

2.Delete

3.Find

4.Find Min

5.Find Max

6.Inorder

7.Preorder

8.Postorder

9.Exit

Enter your choice: 5

To find Maximum element in the tree

Maximum value: 20

1.Insert

2.Delete

3.Find

4.Find Min

5.Find Max

6.Inorder

7.Preorder

8.Postorder

9.Exit

Enter your choice: 2

Enter the element to delete: 10

1.Insert

2.Delete

3.Find

4.Find Min

5.Find Max

6.Inorder

7.Preorder

8.Postorder

9.Exit

Enter your choice: 6

The inorder traversal is: 5 20

1.Insert

2.Delete

3.Find

4.Find Min

5.Find Max

6.Inorder

7.Preorder

8.Postorder

9.Exit

Enter your choice: 9

Exit

Result:

Thus the Binary Search Tree Program and all operations are Successfully Implemented and Executed.

Performing Tree traversal

AIM:

To perform the tree traversal techniques

ALGORITHM:

1. Start
2. Create a node which contains data, left, right of the element
3. Create 3 different types of functions to traversal in 3 different ways:
Inorder, Preorder, Postorder.
4. Call each function and display the output
5. End

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
struct node *left;
int element;
struct node *right;
};
typedef struct node Node;
Node *Insert(Node *Tree, int e);
void Inorder(Node *Tree);
void Preorder(Node *Tree);
void Postorder(Node *Tree);
int main()
{
Node *Tree = NULL;
int n, i, e, ch;
```

```
printf("Enter number of nodes in the tree : ");
scanf("%d", &n);
printf("Enter the elements :\n");
for (i = 1; i <= n; i++)
{
scanf("%d", &e);
Tree = Insert(Tree, e);
}do
{
printf("1. Inorder \n2. Preorder \n3. Postorder \n4. Exit\n");
printf("Enter your choice : ");
scanf("%d", &ch);
switch (ch)
{
case 1:
Inorder(Tree);
printf("\n");
break;
case 2:
Preorder(Tree);
printf("\n");
break;
case 3:
Postorder(Tree);
printf("\n");
break;
}
} while (ch <= 3);
return 0;
}
Node *Insert(Node *Tree, int e)
{
Node *NewNode = malloc(sizeof(Node));
if (Tree == NULL)
```

```
{
NewNode->element = e;
NewNode->left = NULL;
NewNode->right = NULL;
Tree = NewNode;
}
else if (e < Tree->element)
{
Tree->left = Insert(Tree->left, e);
}
else if (e > Tree->element)
{
Tree->right = Insert(Tree->right, e);
}
return Tree;
}
void Inorder(Node *Tree)
{
if (Tree != NULL)
{
Inorder(Tree->left);
printf("%d\t", Tree->element);
Inorder(Tree->right);
}
}
void Preorder(Node *Tree)
{
if (Tree != NULL)
{
printf("%d\t", Tree->element);
Preorder(Tree->left);
Preorder(Tree->right);
}
}
```

```
void Postorder(Node *Tree)
{
if (Tree != NULL)
{
Postorder(Tree->left);
Postorder(Tree->right);
printf("%d\t", Tree->element);
}
}
```

OUTPUT:

```
Enter number of nodes in the tree : 4
Enter the elements :
4 6 9 10
1. Inorder
2. Preorder
3. Postorder
4. Exit
Enter your choice : 1
4       6       9       10
1. Inorder
2. Preorder
3. Postorder
4. Exit
Enter your choice : 2
4       6       9       10
1. Inorder
2. Preorder
3. Postorder
4. Exit
Enter your choice : 3
10      9       6       4
1. Inorder
```

2. Preorder

3. Postorder

4. Exit

Enter your choice : 4

RESULT:

The tree traversal code is implemented and executed successfully

Implementation of AVL Tree

Aim:

To implement the basic operations of AVL tree like insert and delete.

Algorithm:

Step 1: Define Node Structure

1. Define a structure named AVLNode with attributes data,left,right and height.

Step 3: Insertion

1. Start insertion at the root.
2. Compare the key with the current node's data.
3. If the key is smaller, go left; if larger, go right.
4. Recursively insert the key into the appropriate subtree.
5. Update the height of the current node.
6. Check the balance factor of the node.
7. If unbalanced, perform rotations to rebalance the tree.

Step 4: Deletion

1. Start deletion at the root.
2. Search for the node with the given key.
3. Once found, handle three cases:
    - Node has no children or just one child: Simply remove it.
    - Node has two children: Find its in-order successor, copy its value, and delete the successor node.
4. Update the height of the current node and check the balance factor.
5. If unbalanced, perform rotations to rebalance the tree.

Step 5:Rotation functions

Right rotate:

1. Accept a pointer to the current node (denoted as `y`).
2. Set a temporary pointer `x` to the left child of `y`.
3. Set the left child of `y` to the right child of `x`.
4. Set the right child of `x` to `y`.
5. Update the heights of `y` and `x`.
6. Return `x`, the new root of the rotated subtree.

Left rotate:

1. Accept a pointer to the current node (denoted as `x`).
2. Set a temporary pointer `y` to the right child of `x`.
3. Set the right child of `x` to the left child of `y`.
4. Set the left child of `y` to `x`.
5. Update the heights of `x` and `y`.
6. Return `y`, the new root of the rotated subtree.

Step 6: In-order Traversal

1. Recursively traverse the tree in-order (left, root, right).
2. Print the data of each node in sorted order.

Step 7: Main Function

1. Create an interactive menu allowing the user to:
    - Insert a node
    - Delete a node
    - Perform in-order traversal
    - Exit

Step 8: Freeing Memory

1. Recursively free the memory allocated for the entire AVL tree.

Program:

```
#include <stdio.h>

#include <stdlib.h>


struct AVLNode {

   int data;

   struct AVLNode* left;

   struct AVLNode* right;

   int height;

};


int height(struct AVLNode* node) {

   if (node == NULL)

      return 0;

   return node->height;

}


int max(int a, int b) {

   return (a > b) ? a : b;
```

```
}

struct AVLNode* createNode(int key) {
    struct AVLNode* newNode = (struct AVLNode*)malloc(sizeof(struct AVLNode));
    if (newNode != NULL) {
        newNode->data = key;
        newNode->left = NULL;
        newNode->right = NULL;
        newNode->height = 1;
    }
    return newNode;
}

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

struct AVLNode* leftRotate(struct AVLNode* x) {
    struct AVLNode* y = x->right;
    struct AVLNode* 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(struct AVLNode* node) {
   if (node == NULL)
      return 0;
   return height(node->left) - height(node->right);
}

struct AVLNode* insert(struct AVLNode* root, int key) {
   if (root == NULL)
      return createNode(key);

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

   root->height = 1 + max(height(root->left), height(root->right));
   int balance = getBalance(root);

   if (balance > 1 && key < root->left->data)
      return rightRotate(root);
   if (balance < -1 && key > root->right->data)
      return leftRotate(root);
   if (balance > 1 && key > root->left->data) {
      root->left = leftRotate(root->left);
      return rightRotate(root);
   }
   if (balance < -1 && key < root->right->data) {
```

```
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }
    return root;
}

struct AVLNode* minValueNode(struct AVLNode* node) {
    struct AVLNode* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}

struct AVLNode* deleteNode(struct AVLNode* root, int key) {
    if (root == NULL)
        return root;

    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        if ((root->left == NULL) || (root->right == NULL)) {
            struct AVLNode* temp = root->left ? root->left : root->right;
            if (temp == NULL) {
                temp = root;
                root = NULL;
            } else
                *root = *temp;
            free(temp);
        } else {
```

```
            struct AVLNode* temp = minValueNode(root->right);
            root->data = temp->data;
            root->right = deleteNode(root->right, temp->data);
        }
    }

    if (root == NULL)
        return root;

    root->height = 1 + max(height(root->left), height(root->right));
    int balance = getBalance(root);

    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);
    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);
    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

void inOrderTraversal(struct AVLNode* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
```

```
        printf("%d ", root->data);
        inOrderTraversal(root->right);
    }
}

void freeAVLTree(struct AVLNode* root) {
    if (root != NULL) {
        freeAVLTree(root->left);
        freeAVLTree(root->right);
        free(root);
    }
}

int main() {
    struct AVLNode* root = NULL;
    int choice, key;

    do {
        printf("\nAVL Tree Implementation:\n");
        printf("1. Insert a node\n");
        printf("2. Delete a node\n");
        printf("3. In-order Traversal\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the key to insert: ");
                scanf("%d", &key);
                root = insert(root, key);
```

```
            break;
        case 2:
            printf("Enter the key to delete: ");
            scanf("%d", &key);
            root = deleteNode(root, key);
            break;
        case 3:
            printf("In-order Traversal: ");
            inOrderTraversal(root);
            printf("\n");
            break;
        case 4:
            freeAVLTree(root);
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice! Please enter a valid option.\n");
    }

  } while (choice != 4);

  return 0;
}
```

Output:

```
AVL Tree Operations:
1. Insert a node
2. Delete a node
3. In-order Traversal
4. Exit

Enter your choice: 1
```

Implementation of AVL Tree

Enter the key to insert: 5

AVL Tree Operations:

1. Insert a node
2. Delete a node
3. In-order Traversal
4. Exit

Enter your choice: 3

In-order Traversal: 5

AVL Tree Operations:

1. Insert a node
2. Delete a node
3. In-order Traversal
4. Exit

Enter your choice: 1

Enter the key to insert: 3

AVL Tree Operations:

1. Insert a node
2. Delete a node
3. In-order Traversal
4. Exit

Enter your choice: 1

Enter the key to insert: 1

AVL Tree Operations:

1. Insert a node

2. Delete a node

3. In-order Traversal

4. Exit

Enter your choice: 3

In-order Traversal: 1 3 5

AVL Tree Operations:

1. Insert a node

2. Delete a node

3. In-order Traversal

4. Exit

Enter your choice: 2

Enter the key to delete: 3

AVL Tree Operations:

1. Insert a node

2. Delete a node

3. In-order Traversal

4. Exit

Enter your choice: 3

In-order Traversal: 1 5

AVL Tree Operations:

1. Insert a node

2. Delete a node

3. In-order Traversal

4. Exit

Enter your choice: 4

Exiting...

Result:

Thus the AVL tree program and the operations(insert,delete)were successfully implemented.

Topological Sorting

Aim: To create a graph and perform topological sorting.

Algorithm:

Step 1: Start

Step 2: Create a node which contains vertex and next as their members.

Step 3: Allocates memory dynamically for nodes and the graph structure using malloc().

Step 4: Create a graph with a specified number of vertices and initialize adjacency lists.

Step 5: Create a function to add the edges between the vertices

Step 6: Find the indegree for every vertex.

Step 7: Place the vertices whose indegree is 0 on the empty queue.

Step 8: Dequeue the vertex v and decrement the indegree of all its adjacent vertices.

Step 9: Enqueue the vertex on the queue if its indegree falls to zero.

Step 10: Repeat from step 3 until the queue becomes empty.

Step 11: The topological ordering is the order in which the vertices dequeue.

Program:

```
#include <stdio.h>
#include <stdlib.h>
int ver ,in[10],out[10];
struct node
{
```

```
    int data;
    struct node*next;
};
struct list
{
    struct node*head;
};
struct list*adj[10]={0};
void addnode(int s, int d)
{
    out[s]++;
    in[d]++;
    struct node*newn, *temp;
    newn=(struct node*)malloc (sizeof(struct node));
        newn->data=d;
        newn->next=NULL;

    if(adj[s]->head==NULL)
    {
         adj[s]->head=newn;
    }
    else
    {

    temp=adj[s]->head;
```

```
    while(temp->next)
    {
        temp=temp->next;
    }
    temp->next=newn;
    }


}

void top_sort()
{
    int que[10]={0},front=-1,rear=-1,w,v, cur, count=0;
    struct node*temp;
    for (int i=1;i<=ver;i++)
    {
        if(in[i]==0)
        {
            que[++rear]=i;
        }
    }
        while(front!=rear)
        {
            cur=que[++front];
            printf ("%d ",cur);
```

```
            count++;
            temp=adj[cur]->head;
            while(temp)
            {
               w=temp->data;
               if(--in[w]==0)
               {
                  que[++rear]=w;
               }
               temp=temp->next;
            }

         }

      if (count!=ver)
      {
         printf ("CYCLIC GRAPH");
      }
}

void display()
{

   for(int i=1;i<=ver;i++)
   {
```

```
        struct node*temp=adj[i]->head;
        printf ("%d----",i);
        while(temp)
       {
          printf("%d ",temp->data);
           temp=temp->next;
       }
       printf ("\n");
    }
}
int main()
{

   printf ("Enter vertex: ");
   scanf ("%d",&ver);
   for(int i=1;i<=ver;i++)
   {
      in[i]=0;out[i]=0;
      adj[i]=(struct list*)malloc (sizeof(struct list));
      adj[i]->head=NULL;
   }
   int s, d, ch,start;
   do
   {
   printf ("Enter a vertex and its adjaceny: ");
```

```
  scanf ("%d %d",&s,&d);
  addnode(s,d);
  printf ("Continue : ");
  scanf ("%d",&ch);
  }while (ch);
  printf("GRAPH:\n");
  display ();
  printf("\n");
  printf("AFTER SORTING:\n");
  top_sort();

}
```

Output:

GRAPH:

1----2 3 4

2----4 5

3----6

4----3 6

5----4 7

6----

7----6

AFTER SORTING:

1 2 5 4 7 3 6

Result:

Thus the program successfully implemented and executed.

Implementation of BFS and DFS

Aim:

To implement and execute BFS and DFS on a graph and to record the output.

Algorithm:

1. Constants and Arrays:

- Define constants max for maximum size of arrays and arrays s[], q[] for stack and queue respectively.
- Variables front, rear, top to keep track of indices in the queue, stack, and top of the stack respectively.
- Arrays visited[] to mark visited vertices and adj[][] for the adjacency matrix of the graph.

2. BFS Function:

- Initialize visited[] to mark vertices as unvisited.
- Enqueue the starting vertex v into the queue and mark it as visited.
- While the queue is not empty:
    1. Dequeue a vertex a.
    2. Print a.
    3. Visit all adjacent vertices of a that have not been visited, mark them as visited, and enqueue them.

3. DFS Function:

- Initialize visited[] to mark vertices as unvisited and top to keep track of the top of the stack.
- Push the starting vertex v onto the stack and mark it as visited.
- While the stack is not empty:
    1. Pop a vertex a.
    2. Print a.
    3. Visit all adjacent vertices of a that have not been visited, mark them as visited, and push them onto the stack.

4. Create Graph Function:

- Initialize the adjacency matrix with zeros.
- Parse the input array representing edges and set corresponding entries in the adjacency matrix to 1 (for undirected graphs, set both adj[x][y] and adj[y][x]).

5. Print Graph Function:

- Print the adjacency matrix of the graph.

6. Main Function:

- Initialize the graph with a fixed number of vertices n and an array of edges a[][].
- Call the create graph function to build the adjacency matrix.
- Print the adjacency matrix.
- Prompt the user to enter the starting vertex for BFS and DFS.
- Call BFS and DFS functions with the starting vertex.

Program:

```
#include<stdio.h>
#define max 20
int s[max],q[max],front=0,rear=-1,top=-1,n,m,adj[max][max],visited[max];

void bfs(int v){
   visited[v]=1;
   q[++rear]=v;
   int a;
   while(front<=rear){
      a=q[front++];
      printf("%d ",a);
      for (int i=1;i<n+1;i++){
         if (adj[a][i] && !visited[i]){
            visited[i]=1;
            q[++rear]=i;
         }
```

```
        }
    }
}

void dfs(int v){
    visited[v]=1;
    s[++top]=v;
    int a;
    while(top>=0){
        a=s[top--];
        visited[a]=1;
        for (int i=1;i<n+1;i++){
            if (adj[a][i] && !visited[i]){
                s[++top]=i;
            }
        }
        printf("%d ",a);
    }
}

void creategraph(int arr[][2]){
    for (int i=1;i<n+1;i++){
        for (int j=1;j<n+1;j++){
            adj[i][j]=0;
        }
    }
    for (int i=0;i<m;i++){
        int x=arr[i][0];
        int y=arr[i][1];
        adj[x][y]=1;
        adj[y][x]=1;
    }
}

void printgraph(){
    for (int i=1;i<n+1;i++){
        for (int j=1;j<n+1;j++){
```

```
            printf("%d",adj[i][j]);
        }
        printf("\n");
    }

}

int main(){
    n=4;
    int a[][2]={{1,2},{1,3},{2,4}};
    m=sizeof(a)/sizeof(a[0]);
    creategraph(a);
    printf("Graph:");
    printgraph();
    int v;
    printf("Enter the starting point : ");
    scanf("%d",&v);
    printf("BFS : ");
    bfs(v);
    for (int i=1;i<n+1;i++){
        visited[i]=0;
    }
    printf("\nEnter the starting point : ");
    scanf("%d",&v);
    printf("DFS : ");
    dfs(v);

    return 0;
}
```

Output:

```
Graph:
0110
1001
1000
0100
Enter the starting point : 1
```

BFS : 1 2 3 4
Enter the starting point : 1
DFS : 1 3 2 4

Result:

Thus the code implemented and executed successfully.

Implementation of Prim's Algorithm

Aim:

To find minimum spanning tree using prim's algorithm

Algorithm:

1.Start with any vertex: Choose a starting vertex in the graph.

2.Grow the tree one edge at a time: Add the edge with the lowest weight connecting an existing tree vertex to a non-tree vertex.

3.Repeat step 2: Keep adding the lowest weight edge until all vertices are connected in the tree.

4.Avoid cycles: Don't add edges that create cycles in the tree.

5.Minimum spanning tree: The resulting tree is the minimum spanning tree, containing all vertices with minimal total edge weight.

Code:

```
#include <stdio.h>

#include<stdbool.h>


#define INF 999999

int minKey(int key[], bool mstSet[],int n)

{

 int min = INF;

 int min_index;

 for (int v = 0; v <n; v++)

 {

  if (mstSet[v] == false && key[v] < min)

   {

      min = key[v];

      min_index = v;

   }

 }

  return min_index;

}


void printMST(int n,int parent[], int graph[][n])

{
```

```
  printf("Edge \tWeight\n");
  for (int i = 1; i < n; i++)
    printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}

void primMST(int n,int graph[][n]) {

  bool mstSet[n];
  int parent[n];
  int key[n];

  for (int i = 0; i < n; i++)
  {
    key[i] = INF;
    mstSet[i] = false;
  }

  key[0] = 0;
  parent[0] = -1;

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

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

```
  }
 printMST(n,parent, graph);
}

int main() {

  int n;
  printf("Enter the number of vertices: ");
  scanf("%d", &n);
  int graph[n][n];

  printf("\nEnter the adjacency matrix:\n");
  for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
   {
      scanf("%d", &graph[i][j]);
    }
  }
 primMST(n,graph);
 return 0;
}
```

Output:

Enter the adjacency matrix: 0 2 3 0 0

2 0 0 1 4

3 0 0 0 2

0 1 0 0 3

0 4 2 3 0

Edge	Weight
1 - 2	2
0 - 3	3

2 - 4 4

3 - 0 3

1 - 3 1

Result:

Thus we implemented prim's algorithm successfully

Dijkstra's Algorithm Implementation in C

AIM:

The aim of this assignment is to understand and implement Dijkstra's algorithm in C. Dijkstra's algorithm is a popular algorithm for finding the shortest path between nodes in a graph, particularly for non-negative edge weights.

Algorithm:

1. **Initialization:**
    - Set the initial distance of the source node to 0 and all other nodes to infinity.
    - Mark all nodes as unvisited.

2. **Iteration:**
    - Select the unvisited node with the smallest distance, this node is considered visited.
    - Update the distances of its neighbouring nodes by adding the weight of the edge from the selected node to each neighbouring node, if the new distance is smaller than the current distance.

3. **Termination:**
    - Repeat step 2 until all nodes are visited or the destination node is reached.
    - The shortest path from the source to each node can be reconstructed from the final distances.

PROGRAM:

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

#define V 9

int minDistance(int dist[], bool sptSet[]) {
   int min = INT_MAX, min_index;

   for (int v = 0; v < V; v++)
      if (sptSet[v] == false && dist[v] <= min)
         min = dist[v], min_index = v;

   return min_index;
}

void printSolution(int dist[]) {
   printf("Vertex \t\t Distance from Source\n");
   for (int i = 0; i < V; i++)
      printf("%d \t\t %d\n", i, dist[i]);
```

```
}

void dijkstra(int graph[V][V], int src) {
   int dist[V];
   bool sptSet[V];

   for (int i = 0; i < V; i++)
      dist[i] = INT_MAX, sptSet[i] = false;

   dist[src] = 0;

   for (int count = 0; count < V - 1; count++) {
      int u = minDistance(dist, sptSet);

      sptSet[u] = true;

      for (int v = 0; v < V; v++)
         if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v])
            dist[v] = dist[u] + graph[u][v];
   }

   printSolution(dist);
}
```

```
int main() {
   int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                      {4, 0, 8, 0, 0, 0, 0, 11, 0},
                      {0, 8, 0, 7, 0, 4, 0, 0, 2},
                      {0, 0, 7, 0, 9, 14, 0, 0, 0},
                      {0, 0, 0, 9, 0, 10, 0, 0, 0},
                      {0, 0, 4, 14, 10, 0, 2, 0, 0},
                      {0, 0, 0, 0, 0, 2, 0, 1, 6},
                      {8, 11, 0, 0, 0, 0, 1, 0, 7},
                      {0, 0, 2, 0, 0, 0, 6, 7, 0}};
   dijkstra(graph, 0);
   return 0;
}
```

PROGRAM OUTPUT:

```
Vertex      Distance from Source
0                0
1                4
2                12
```

3	19
4	21
5	11
6	9
7	8
8	14

RESULT:

The program successfully implements Dijkstra's algorithm to find the shortest paths from a given source vertex to all other vertices in a weighted graph.

AIM:

To understand the working and implement merge sort and quick sort in C.

ALGORITHM:

QUICK SORT:

1. **Base Case**: If the array has 0 or 1 element, it is already sorted.
2. **Pivot Selection**: Select the pivot element (commonly the middle element).
3. **Partitioning**: Partition the array into three parts:
    - Elements less than the pivot.
    - Elements equal to the pivot.
    - Elements greater than the pivot.
4. **Recursion**: Recursively apply the same logic to the left and right sub-arrays.
5. **Concatenation**: Combine the sorted sub-arrays and the pivot to get the final sorted array.

MERGE SORT:

1. **Base Case**: If the array has 0 or 1 element, it is already sorted.
2. **Divide**: Split the array into two halves.
3. **Recursion**: Recursively apply the same logic to both halves.
4. **Merge**: Merge the two sorted halves to produce the sorted array:
    - Compare the elements of the two halves one by one and build a new sorted array.

These algorithms both have their own advantages and typical use cases. Quick Sort is generally faster in practice with an average time complexity of O(nlog⁡n)O(n \log n)O(nlogn), but it can degrade to O(n2)O(n^2)O(n2) in the worst case. Merge Sort guarantees O(nlog⁡n)O(n \log n)O(nlogn) time complexity and is stable, making it suitable for data that requires stability.

CODING

```
//merge sort

#include<stdio.h>

#include<stdlib.h>

void merge(int arr[],int left,int mid,int right)

{

  int n1=mid-left+1;
```

```
int n2=right-mid;

int l[n1],r[n2];

for(int i=0;i<n1;i++)
l[i]=arr[left+i];
for(int j=0;j<n2;j++)
r[j]=arr[mid+j+1];

int i=0,j=0,k=left;

while(i<n1 && j<n2)
{
   if(l[i]<=r[j])
   {
      arr[k]=l[i];
      i++;
   }
   else
   {
      arr[k]=r[j];
      j++;
   }
   k++;
}

while(i<n1)
{
   arr[k]=l[i];
```

```
        i++;
        k++;
    }

    while(j<n2)
    {
        arr[k]=r[j];
        j++;
        k++;
    }
}

void mergesort(int arr[],int left,int right)
{
    if(left<right)
    {
        int mid= left+(right-left)/2;
        mergesort(arr,left,mid);
        mergesort(arr,mid+1,right);
        merge(arr,left,mid,right);
    }
}

void print(int arr[],int size)
{
    for(int i=0;i<size;i++)
    printf("%d ",arr[i]);
}
```

```
int main()
{
   int arr[]={7,9,1,3,2};
   int size= sizeof(arr)/sizeof(arr[0]);

   mergesort(arr,0,size-1);
   print(arr,size);
}
//quicksort
#include<stdio.h>
#include<stdlib.h>
void swap(int *a,int *b)
{
   int temp=*a;
   *a=*b;
   *b= temp;
}

int partition(int arr[],int low,int high)
{
   int pivot=arr[high];
   int i=(low-1);
   for(int j=low;j<high;j++)
   {
      if(arr[j]<pivot)
      {
         i++;
         swap(&arr[i],&arr[j]);
      }
```

```
    }
    swap(&arr[i+1],&arr[high]);
    return (i+1);
}

void quicksort(int arr[],int low,int high)
{
    if(low<high)
    {
    int pi=partition(arr,low,high);
    quicksort(arr,low,pi-1);
    quicksort(arr,pi+1,high);
    }
}

void print(int arr[],int size)
{
    for(int i=0;i<size;i++)
    printf("%d ",arr[i]);
}
int main()
{
    int arr[]={2,5,1,0,4};
    int size=sizeof(arr)/sizeof(arr[0]);
    quicksort(arr,0,size-1);
    print(arr,size);
}
```

OUTPUT FOR MERGE SORT:

1 2 3 7 9

OUTPUT FOR QUICK SORT:

0 1 2 4 5

RESULT:

Thus Quick Sort And Merge Sort Is Implemented Successfully.

Write a C program to create a hash table and perform collision resolution using the following techniques:

1) Open Addressing
2) Closed Addressing
3) Rehashing

Algorithm:

Open Addressing (Probing):

- Algorithm: When a collision occurs (key's hash position is occupied), it explores alternative locations in the hash table to place the key-value pair.
- Uses a probe sequence: A series of steps to generate alternative positions based on the initial hash position and the number of probes made so far. Common examples include linear probing, quadratic probing, and double hashing.
- Focuses on finding an empty slot within the existing hash table to insert the element.

Closed Addressing (Separate Chaining):

- Algorithm: Instead of searching for an empty slot within the hash table, it stores elements in linked lists associated with each hash position (bucket).
- Collisions are resolved by adding the new element to the linked list at the corresponding bucket.
- Easier to handle deletions as elements can be removed from the linked list.
- May experience performance issues if many elements hash to the same bucket, causing long linked lists.

Rehashing:

- Technique: Used to address performance issues in open addressing when the load factor (occupied slots / table size) becomes too high.

- Process: Creates a new, larger hash table and redistributes all existing elements using their hash function in the new table.
- Reduces the number of probes needed for search and insertion operations by increasing the available space.
- May require re-implementing the hash function if it depends on the table size.

Code:

Open Addressing:

```
#include <stdio.h>

#include <stdlib.h>


#define EMPTY -1


unsigned int hash_function (int key, int table_size) {

    return key % table_size;

}


void linear_probing_insert (int *hash_table, int table_size, int key) {

    int index = hash_function (key, table_size);

    int original_index = index;

    while (hash_table[index] ! = EMPTY) { // Assuming -1 indicates an
empty slot
```

```
        index = (index + 1) % table_size;
        if (index == original_index) {
            printf ("Hash table is full\n");
            return;
        }
    }
    hash_table[index] = key;
}

int linear_probing_search (int *hash_table, int table_size, int key) {
    int index = hash_function (key, table_size);
    int original_index = index;
    while (hash_table[index] != EMPTY) { // Assuming -1 indicates an
empty slot
        if (hash_table[index] == key) {
            return index;
        }
        index = (index + 1) % table_size;
        if (index == original_index) {
            break;
        }
    }
    return -1; // Key not found
}
```

```
void linear_probing_delete (int *hash_table, int table_size, int key) {
    int index = linear_probing_search (hash_table, table_size, key);
    if (index != -1) {
        hash_table[index] = EMPTY; // Mark as deleted
    } else {
        Printf ("Key not found\n");
    }
}

int main() {
    int table_size = 10;
    int *hash_table = (int *) malloc (table_size * sizeof(int));
    for (int i = 0; i < table_size; i++) {
        hash_table[i] = EMPTY; // Initialize hash table slots to -1 (indicating
empty)
    }

    linear_probing_insert (hash_table, table_size, 5);
    linear_probing_insert (hash_table, table_size, 15);
    linear_probing_insert (hash_table, table_size, 25);

    int index = linear_probing_search(hash_table, table_size, 15);
    if (index! =  -1) {
```

```
        printf ("Key 15 found at index %d\n", index);
    } else {
        Printf ("Key 15 not found\n");
    }

    linear_probing_delete (hash_table, table_size, 15);
    index = linear_probing_search (hash_table, table_size, 15);
    if (index!= -1) {
        printf ("Key 15 found at index %d\n", index);
    } else {
        Printf ("Key 15 not found\n");
    }

    free(hash_table);
    return 0;
}
```

Output:

Key 15 found at index 6

Key 15 not found

Closed Addressing:

Code:

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

// Structure for a node in the linked list
typedef struct Node {
  int key;
  struct Node* next;
} Node;

// Structure for the hash table
typedef struct HashTable {
  int table_size;
  Node** chains;  // Array of pointers to linked list heads
} HashTable;

// Function to create a new node
Node* createNode(int key) {
  Node* newNode = (Node*)malloc(sizeof(Node));
  newNode->key = key;
  newNode->next = NULL;
  return newNode;
}

// Function to create a new hash table
HashTable* createHashTable(int table_size) {
  HashTable* hashTable = (HashTable*)malloc(sizeof(HashTable));
  hashTable->table_size = table_size;
  hashTable->chains = (Node**)malloc(table_size * sizeof(Node*));
  for (int i = 0; i < table_size; i++) {
    hashTable->chains[i] = NULL;  // Initialize all chains to NULL
  }
  return hashTable;
}

// Hash function (replace with your own implementation)
int hashFunction(int key) {
  return key % table_size;  // Simple modulo function
}


void insert(HashTable* hashTable, int key) {
  int index = hashFunction(key);
  Node* newNode = createNode(key);
  newNode->next = hashTable->chains[index];
  hashTable->chains[index] = newNode;
}
```

```
int search(HashTable* hashTable, int key) {
  int index = hashFunction(key);
  Node* head = hashTable->chains[index];
  while (head != NULL) {
    if (head->key == key) {
      return 1;  // Key found
    }
    head = head->next;
  }
  return 0;
}

int main() {
  int table_size = 7;
  HashTable* hashTable = createHashTable(table_size);

  insert(hashTable, 10);
  insert(hashTable, 13);
  insert(hashTable, 5);

  if (search(hashTable, 10)) {
    printf("Key 10 found\n");
  } else {
    printf("Key 10 not found\n");
  }

  if (search(hashTable, 13)) {
    printf("Key 13 found\n");
  } else {
    printf("Key 13 not found\n");
  }

  return 0;
}
```

Output:

Key 10 found

Key 13 found

Linear Probing:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_FILL 0.75
typedef struct Entry {
  int key;
  int value;
  int isDeleted;
} Entry;

// Structure for the hash table
typedef struct HashTable {
  int table_size;
  Entry* table;
  int filledEntries;
} HashTable;

// Function to create a new node (entry)
Entry* createEntry (int key, int value) {
  Entry* entry = (Entry*) malloc (sizeof (Entry));
  entry->key = key;
```

```
  entry->value = value;
  entry->isDeleted = 0;
  return entry;
}

// Function to create a new hash table
HashTable* createHashTable (int table_size) {
  HashTable* hashTable = (HashTable*) malloc (sizeof (HashTable));
  hashTable->table_size = table_size;
  hashTable->table = (Entry*) malloc (table_size * sizeof (Entry));
  hashTable->filledEntries = 0;
  for (int i = 0; i < table_size; i++) {
    hashTable->table[i]. isDeleted = 1;  // Initialize all entries as deleted
  }
  return hashTable;
}

// Hash function (replace with your own implementation)
int hashFunction (int key) {
  return key % table_size;
}
int probe (HashTable* hashTable, int index, int i) {
  return (index + i) % hashTable->table_size;
```

```
}

// Function to insert a key-value pair into the hash table
void insert (HashTable* hashTable, int key, int value) {
  // Check if load factor exceeds threshold (consider rehashing if needed)
  if ((float)hashTable->filledEntries / hashTable->table_size >=
MAX_FILL) {
    // Handle full table (e.g., print error or resize)
    Printf ("Hash table is full\n");
    return;
  }

  int index = hashFunction(key);
  int i = 0;
  while (hashTable->table [probe (hashTable, index, i)].isDeleted != 1) {
    // Linear probing until an empty or deleted slot is found
    i++;
  }
  hashTable->table[probe(hashTable, index, i)] = *createEntry(key,
value);
  hashTable->filledEntries++;
}

// Function to search for a key in the hash table
```

```
int search (HashTable* hashTable, int key) {
  int index = hashFunction(key);
  int i = 0;
  while (i < hashTable->table_size) {
    int probedIndex = probe (hashTable, index, i);
    if (hashTable->table[probedIndex].isDeleted  == 1) {
      // Skip deleted entries
      i++;
    } else if (hashTable->table[probedIndex].key == key) {
      return 1;  // Key found
    } else {
      // Continue probing
      i++;
    }
  }
  return 0;  // Key not found
}

int main() {
  // Example usage
  int table_size = 10;
  HashTable* hashTable = createHashTable(table_size);
```

```
  Insert (hashTable, 7, 2);
  insert (hashTable, 10, 5);
  insert (hashTable, 13, 8);

  if (search (hashTable, 7)) {
    printf ("Key 7 found\n");
  } else {
    Printf ("Key 7 not found\n");
  }

  if (search (hashTable, 13)) {
    printf ("Key 13 found\n");
  } else {
    Printf ("Key 13 not found\n");
  }

  return 0;
}
```

Output:

Key 7 found

Key 13 found

Quadratic Probing:

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

#define TABLE_SIZE 10
typedef struct HashTable {
    int *table;
    int size;
} HashTable;
HashTable* createTable (int size) {
    HashTable *newTable = (HashTable*) malloc (sizeof (HashTable));
    newTable->size = size;
    newTable->table = (int*) malloc (size * sizeof(int));

    for(int i = 0; i < size; i++) {
        newTable->table[i] = -1;  // -1 indicates an empty slot
    }
    return newTable;
}
```

```
int hashFunction (int key, int size) {
    return key % size;
}
void insert (HashTable *table, int key) {
    int index = hashFunction (key, table->size);
    int i = 0;

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

    table->table[(index + i * i) % table->size] = key;
    printf ("Inserted key %d at index %d\n", key, (index + i * i) % table-
>size);
}
int search (HashTable *table, int key) {
    int index = hashFunction (key, table->size);
    int i = 0;

    while (table->table [(index + i * i) % table->size] != key) {
```

```
        if (table->table [(index + i * i) % table->size] == -1 || i == table->size)
{
            return -1;
        }
        i++;
    }
    return (index + i * i) % table->size;
}

// Function to display the hash table
void displayTable (HashTable *table) {
    for (int i = 0; i < table->size; i++) {
        if (table->table[i] != -1) {
            printf ("Index %d: %d\n", i, table->table[i]);
        } else {
            Printf ("Index %d: Empty\n", i);
        }
    }
}

int main() {
    HashTable *hashTable = createTable (TABLE_SIZE);

    // Inserting values
```

```
Insert (hashTable, 5);
insert (hashTable, 25);
insert (hashTable, 15);
insert (hashTable, 35);
insert (hashTable, 95);

// Display the hash table
Printf ("\nHash Table:\n");
displayTable(hashTable);

// Searching values
int keysToSearch [] = {5, 15, 95, 99};
for (int i = 0; i < 4; i++) {
    int index = search (hashTable, keysToSearch[i]);
    if (index ! = -1) {
        printf ("Key %d found at index %d\n", keysToSearch[i], index);
    } else {
        Printf ("Key %d not found in the hash table\n", keysToSearch[i]);
    }
}

// Clean up
free(hashTable->table);
```

```
    free(hashTable);

    return 0;
}
```

Output:

```
Inserted key 5 at index 5
Inserted key 25 at index 6
Inserted key 15 at index 9
Inserted key 35 at index 4
Inserted key 95 at index 1

Hash Table:
Index 0: Empty
Index 1: 95
Index 2: Empty
Index 3: Empty
Index 4: 35
Index 5: 5
Index 6: 25
Index 7: Empty
Index 8: Empty
Index 9: 15
```

Key 5 found at index 5

Key 15 found at index 9

Key 95 found at index 1

Key 99 not found in the hash table

Rehashing:

Code:

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

#define INITIAL_TABLE_SIZE 10
#define LOAD_FACTOR_THRESHOLD 0.7

typedef struct HashTable {
    int *table;
    int size;
    int count;
} HashTable;

// Function to create a new hash table
```

```
HashTable* createTable (int size) {
    HashTable *newTable = (HashTable*) malloc(sizeof(HashTable));
    newTable->size = size;
    newTable->count = 0;
    newTable->table = (int*) malloc(size * sizeof(int));
    for(int i = 0; i < size; i++) {
        newTable->table[i] = -1;  // -1 indicates an empty slot
    }
    return newTable;
}

// Hash function
int hashFunction(int key, int size) {
    return key % size;
}

// Function to insert a value into the hash table using quadratic probing
void insert(HashTable *table, int key);

// Function to rehash the table
void rehash(HashTable *table) {
    int oldSize = table->size;
    int *oldTable = table->table;
```

```
    table->size *= 2;
    table->count = 0;
    table->table = (int*) malloc(table->size * sizeof(int));
    for(int i = 0; i < table->size; i++) {
        table->table[i] = -1;
    }

    for(int i = 0; i < oldSize; i++) {
        if (oldTable[i] != -1) {
            insert(table, oldTable[i]);
        }
    }

    free(oldTable);
}

// Function to insert a value into the hash table using quadratic probing
void insert(HashTable *table, int key) {
    if ((float)table->count / table->size > LOAD_FACTOR_THRESHOLD) {
        printf("Load factor exceeded. Rehashing...\n");
        rehash(table);
    }
```

```
    int index = hashFunction (key, table->size);
    int i = 0;

    while(table->table [(index + i * i) % table->size] != -1) {
        i++;
    }

    table->table [(index + i * i) % table->size] = key;
    table->count++;
    printf("Inserted key %d at index %d\n", key, (index + i * i) % table-
>size);
}

// Function to display the hash table
void displayTable (HashTable *table) {
    for (int i = 0; i < table->size; i++) {
        if (table->table[i] != -1) {
            printf ("Index %d: %d\n", i, table->table[i]);
        } else {
            printf("Index %d: Empty\n", i);
        }
    }
}
```

```
int main () {
    HashTable *hashTable = createTable (INITIAL_TABLE_SIZE);

    // Inserting values
    insert (hashTable, 5);
    insert (hashTable, 25);
    insert (hashTable, 15);
    insert hashTable, 35);
    insert (hashTable, 95);
    insert (hashTable, 85);
    insert (hashTable, 75);
    insert (hashTable, 65);
    insert (hashTable, 55);

    // Display the hash table
    Printf ("\n Hash Table:\n");
    displayTable(hashTable);

    // Clean up
    free(hashTable->table);
    free(hashTable);
```

```
    return 0;
}
```

Output:

Index 16: 35

Index 17: Empty

Index 18: Empty

Index 19: 75