RAJALAKSHMI ENGINEERING COLLEGE

An AUTONOMOUS Institution
Affiliated to ANNA UNIVERSITY, Chennai

25 years OF ACADEMIC EXCELLENCE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Laboratory Manual

REGULATION 2023

CS23231 – DATA STRUCTURES

RAJALAKSHMI ENGINEERING COLLEGE

An Autonomous Institution, Affiliated to Anna University
Rajalakshmi Nagar, Thandalam – 602 105

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS23231 – DATA STRUCTURES
(*Regulation 2023*)

LAB MANUAL

Name : ..... ROONEY BALA .I.....

Register No. : ..... 230701269 .....

Year / Branch / Section: ..... **Ist YEAR – CSE E** .....

Semester : ..... **II** .....

Academic Year : ..... **2023 - 2024** .....

LESSON PLAN

Course Code	Course Title (Laboratory Integrated Theory Course)	L	T	P	C
CS23231	Data Structures	1	0	6	4

LIST OF EXPERIMENTS	
Sl. No	**Name of the experiment**
Week 1	Implementation of Single Linked List (Insertion, Deletion and Display)
Week 2	Implementation of Doubly Linked List (Insertion, Deletion and Display)
Week 3	Applications of Singly Linked List (Polynomial Manipulation)
Week 4	Implementation of Stack using Array and Linked List implementation
Week 5	Applications of Stack (Infix to Postfix)
Week 6	Applications of Stack (Evaluating Arithmetic Expression)
Week 7	Implementation of Queue using Array and Linked List implementation
Week 8	Implementation of Binary Search Tree
Week 9	Performing Tree Traversal Techniques
Week 10	Implementation of AVL Tree
Week 11	Performing Topological Sorting
Week 12	Implementation of BFS, DFS
Week 13	Implementation of Prim's Algorithm
Week 14	Implementation of Dijkstra's Algorithm
Week 15	Program to perform Sorting
Week 16	Implementation of Open Addressing (Linear Probing and Quadratic Probing)
Week 17	Implementation of Rehashing

INDEX

Note: Students have to write the Algorithms at left side of each problem statements.

Ex. No.: 1	Implementation of Single Linked List	Date:04.05.2024

Write a C program to implement the following operations on Singly Linked List.

(i) Insert a node in the beginning of a list.

(ii) Insert a node after P

(iii) Insert a node at the end of a list

(iv) Find an element in a list

(v) FindNext

(vi) FindPrevious

(vii) isLast

(viii) isEmpty

(ix) Delete a node in the beginning of a list.

(x) Delete a node after P

(xi) Delete a node at the end of a list

(xii) Delete the List

Aim:

To implement Insert, Delete and Find Operations on a Singly Linked List.

Algorithm:

Insert at Beginning:

- Create a new node.
- Set new node's next to the current head.
- Update head to the new node.

Insert at End:

- Create a new node.
- If the list is empty, set head to the new node.
- Else, traverse to the last node and set its next to the new node.

Insert in Middle:

- Traverse to the node before the desired position.
- Create a new node.
- Set new node's next to the next node.
- Update previous node's next to the new node.

Delete First Node:

- If list is not empty, update head to the next node.

Delete Last Node:

- Traverse to the second last node.
- Set its next to null.

Delete Node at Position:

- If position is first, delete the first node.
- Else, traverse to the node before the desired position.
- Update previous node's next to skip the node to be deleted.

Search Node:

- Traverse through the list to find the node with the given data.
- Return the position or indicate if not found.

Check if List is Empty:

- Return true if head is null, otherwise false.

Check if Node is Last:

- Traverse to the node at the given position.
- Check if its next is null.

Display List:

- Traverse through the list.
- Print each node's data.

Program:

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

struct node
{
   int data;
   struct node *link;
} *first = NULL;

void insert_beg(int);
void insert_end(int);
void insert_mid(int, int);
void del_first();
void del_last();
void del_anypos(int);
void display();
void del_all();
void isLast(int);
void isEmpty();
void findnext(int);
void findprev(int);
int count();
void search(int);

void insert_beg(int roll)
{
   struct node *newnode;
   newnode = (struct node *)malloc(sizeof(struct node));
   newnode->data = roll;
   if (first == NULL)
   {
      newnode->link = NULL;
      first = newnode;
   }
```

```
else
   {
      newnode->link = first;
      first = newnode;
   }
   printf("Data inserted\n");
}

void insert_end(int roll)
{
   struct node *newnode, *temp;
   temp = first;
   newnode = (struct node *)malloc(sizeof(struct node));
   newnode->data = roll;
   if (first == NULL)
   {
      newnode->link = NULL;
      first = newnode;
   }
   else
   {
      while (temp->link != NULL)
      {
         temp = temp->link;
      }
      newnode->link = NULL;
      temp->link = newnode;
   }
   printf("Data Inserted\n");
}
```

```
void display()
{
   struct node *temp = first;
   if (temp != NULL)
   {
      while (temp != NULL)
      {
         printf("%d ", temp->data);
         temp = temp->link;
      }
   }
   else
   {
      printf("\nNo data inside");
   }
}

void insert_mid(int loc, int roll)
{
   struct node *newnode, *temp = NULL;
   temp = first;
   int i = 1;
   newnode = (struct node *)malloc(sizeof(struct node));
   newnode->data = roll;
   int t = count();
   if (loc == 0)
      insert_beg(roll);
   else if (loc < t)
   {
      while (i < loc)
      {
         temp = temp->link;
         i++;
      }
```

```
newnode->link = temp->link;
      temp->link = newnode;
      printf("Data Inserted\n");
   }
   else if (loc == t)
   {
      insert_end(roll);
   }
   else if (loc > t + 1)
   {
      printf("Out of bounds");
   }
}

int count()
{
   struct node *temp = first;
   int count = 0;
   while (temp != NULL)
   {
      temp = temp->link;
      count++;
   }
   return count;
}

void del_first()
{
   struct node *temp = NULL;
   temp = first;
   if (first == NULL)
   {
      printf("INVALID OPERATION");
   }
```

```
    else
    {
        first = temp->link;
        free(temp);
    }
    printf("Data deleted\n");
}

void del_last()
{
    struct node *temp = NULL, *temp1 = NULL;
    temp = first;
    while (temp->link != NULL)
    {
        temp1 = temp;
        temp = temp->link;
    }
    free(temp);
    temp1->link = NULL;
    printf("Data Deleted\n");
}

void del_anypos(int pos)
{
    struct node *temp = NULL, *temp1 = NULL;
    temp = first;
    if (pos == 0)
    {
        del_first();
    }
```

```
    else
    {
        for (int i = 1; i <= pos; i++)
        {
            if (temp == NULL)
            {
                printf("INVALID");
                break;
            }
            else
            {
                temp1 = temp;
                temp = temp->link;
            }
        }
        if (temp->link != NULL)
            temp1->link = temp->link->link;
        else
            temp1->link = temp->link;
        free(temp);
    }
    printf("Data Deleted\n");
}

void del_all()
{
    struct node *temp = first, *temp1 = NULL;
    while (temp != NULL)
    {
        temp1 = temp;
```

```
        temp = temp->link;
        free(temp1);
        first = NULL;
    }
    printf("\nAll data deleted successfully");
}

void isEmpty()
{
    if (first == NULL)
        printf("\nThe list is empty\n");
    else
        printf("\nThe list is not empty\n");
}

void isLast(int pos)
{
    struct node *temp = first;
    int i = 1;
    while (i < pos)
    {
        temp = temp->link;
        i++;
    }
    if (temp->link == NULL)
        printf("\nIt is the last node");
    else
        printf("\nIt is not the last node");
}
```

```
void search(int data)
{
   int c = 1;
   struct node *temp = first;
   if (first == NULL)
      printf("\nThe list is empty\n");
   else
   {
      while (temp != NULL && temp->data != data)
      {
         temp = temp->link;
         c++;
         if (c > count())
            printf("No data in list\n");
         else
            continue;
      }
      printf("\n%d is the position of data\n", c);
   }
}

void findnext(int data)
{
   int c = 1;
   struct node *temp = first;
   if (first == NULL)
   {
      printf("\nThe list is empty\n");
   }
```

```
    else
    {
        while (temp != NULL && temp->data != data)
        {
            temp = temp->link;
            c++;
            if (c > count())
                printf("No data in list\n");
            else
                continue;
        }
        printf("\n%d is the position of data\n", c + 1);
    }
}

void findprev(int data)
{
    int c = 1;
    struct node *temp = first;
    if (first == NULL)     printf("\nThe list is empty\n");
    else
    {
        while (temp != NULL && temp->data != data)
        {
            temp = temp->link;
            c++;
            if (c > count())
                printf("No data in list\n");
            else
                continue;
        }
        printf("\n%d is the position of data\n", c - 1);
    }
}
```

```
int main()
{
   int n, ch, pos;
   printf("MENU DRIVEN PROGRAM:\n");
   printf("0. Exit\n");
   printf("1. Insert a node at the beginning\n");
   printf("2. Insert a node at the end\n");
   printf("3. Insert a node after P\n");
   printf("4. Search an element\n");
   printf("5. Find next\n");
   printf("6. Find previous\n");
   printf("7. isLast\n");
   printf("8. isEmpty\n");
   printf("9. Delete at beg\n");
   printf("10. Delete after P\n");
   printf("11. Delete at end\n");
   printf("12. Delete list\n");
   printf("13. Display\n");

   while (1)
   {
      printf("\nEnter your choice: ");
      scanf("%d", &ch);
      switch (ch)
      {
         case 1:
            printf("\nEnter roll to insert at beginning: ");
            scanf("%d", &n);
            insert_beg(n);
            break;
```

```
case 2:
   printf("\nEnter roll to insert at end: ");
   scanf("%d", &n);
   insert_end(n);
   break;

case 3:
   printf("Enter P: ");
   scanf("%d", &pos);
   printf("\nEnter roll to insert after P: ");
   scanf("%d", &n);
   insert_mid(pos, n);
   break;

case 4:
   printf("\nEnter data to search: ");
   scanf("%d", &n);
   search(n);
   break;

case 5:
   printf("\nEnter data to find next: ");
   scanf("%d", &n);
   findnext(n);
   break;

case 6:
   printf("\nEnter data to find previous: ");
   scanf("%d", &n);
   findprev(n);
   break;
```

```
case 7:
   printf("\nEnter position to check last: ");
   scanf("%d", &pos);
   isLast(pos);
   break;

case 8:
   isEmpty();
   break;

case 9:
   del_first();
   break;

case 10:
   printf("\nEnter position to delete after P: ");
   scanf("%d", &pos);
   del_anypos(pos);
   break;

case 11:
   del_last();
   break;

case 12:
   del_all();
   break;

case 13:
   display();
   break;
```

```
            default:
                printf("\nMENU EXITED");
                break;
        }
        if (ch == 0)
        {
            break;
        }
        else
        {
            continue;
        }
    }
    return 0;
}
```

Output:

MENU DRIVEN PROGRAM:

0. Exit
1. Insert a node at the beginning
2. Insert a node at the end
3. Insert a node after P
4. Search an element
5. Find next
6. Find previous
7. isLast
8. isEmpty
9. Delete at beg
10. Delete after P
11. Delete at end
12. Delete list
13. Display

Enter your choice : 1

Enter roll to insert at beginning : 2
Data inserted

Enter your choice : 1

Enter roll to insert at beginning : 4
Data inserted

Enter your choice : 1

Enter roll to insert at beginning : 5
Data inserted

Enter your choice : 1

Enter roll to insert at beginning : 6
Data inserted

Enter your choice : 13
6 5 4 2
Enter your choice : 7

Enter position to check last : 2
It is not the last node

```
Enter your choice :  4
Enter data to search : 5
2 is the position of data

Enter your choice :  0

MENU EXITED
```

Result:

This code is verified and implemented successfully.

Ex. No.: 2	Implementation of Doubly Linked List	Date:04.05.2024

Write a C program to implement the following operations on Doubly Linked List.

(i) Insertion

(ii) Deletion

(iii) Search

(iv) Display

Aim:

To implement Insert, Delete and Find Operations on a Singly Linked List.

Algorithm:

- Define a struct `node` with integer data and pointers to the next and previous nodes.
- Initialize a global pointer `head` to `NULL`.
- For insertion at the beginning:
    - Allocate memory for a new node.
    - Set its data and pointers accordingly.
    - Update `head` if list is not empty, else set `head` to the new node.
- For insertion at the end:
    - Allocate memory for a new node.
    - Traverse the list to the last node.
    - Update pointers to include the new node.
- For insertion after a given node `prev_node`:
    - Allocate memory for a new node.
    - Update pointers of the new node and its neighbors.
- For deletion from the beginning:
    - If the list is empty, return.
    - Update `head` to point to the next node and free the old head.
- For deletion from the end:
    - If the list is empty or has only one node, delete that node and set `head` to `NULL`.
    - Traverse to the second last node, update pointers, and free the last node.

- For deletion after a given node `prev_node`:
    - If `prev_node` is `NULL` or the next node is `NULL`, return.
    - Update pointers to skip the node to be deleted and free it.
- For display:
    - Start from `head` and print each node's data while traversing.
- Repeat steps 3-9 based on user input until exit.

Program:

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

struct node {
   int data;
   struct node *next;
   struct node *prev;
};

struct node *l=NULL;
struct node *newnode;

void insert_first(){
   newnode=(struct node*)malloc(sizeof(struct node));
   if(newnode!=NULL){
      printf("Enter the element : ");
      scanf("%d",&newnode->data);
      if (l!=NULL){
         newnode->next=l;
         newnode->prev=NULL;
         l->prev=newnode;
         l=newnode;
      }
      else
      {
         newnode->next=NULL;
         newnode->prev=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;
         }
         newnode->prev=p;
         p->next=newnode;

      }
      else
      {
         newnode->prev=NULL;
         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;
       newnode->prev=p;
       p->next=newnode;
       p->next->prev=newnode;
    }
}

void delete_first(){
    struct node*p=l;
    l=l->next;
    l->prev=NULL;
    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){
        p=p->next;
        c++;
    }
    temp=p->next;
    if (temp->next==NULL){
        free(temp);
        p->next=NULL;
    }
    else{
        p->next=temp->next;
        temp->next->prev=p;
        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 the 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){
        p=p->next;
        c++;
    }
    printf("The element present previous to the position %d is %d",pos,p->prev->data);
}


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.display\n11.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:
               display();
               break;
         }
      } while(n!=11);
   }
```

Output:

```
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.display
11.exit

Enter your option:1
Enter the element : 1

Enter your option:1
Enter the element : 2

Enter your option:2
enter no 3

Enter your option:2
enter no 4

Enter your option:10
2 1 3 4

Enter your option:7
enter position 1
The element present at the position 1 is 2

Enter your option:5

Enter your option:10
2 1 3

Enter your option:11
```

Result:

This code is verified and implemented successfully.

Ex. No.: 3	Polynomial Manipulation	Date:04.05.2024

Write a C program to implement the following operations on Singly Linked List.

(i) Polynomial Addition

(ii) Polynomial Subtraction

(iii) Polynomial Multiplication

Aim:

To write a C program to implement the following operations on singly linked list
(i) Polynomial Addition
(ii) Polynomial Subtraction
(iii) Polynomial Multiplication

Algorithm:

1. Start
2. Define structure
3. Create term functions
4. Insert term into the polynomial and add subtract and multiplication these polynomial and display it.
5. End

Program:

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

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

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

```
      temp->next = new;
      temp = new;
    }
    temp->next=NULL;
    return L1;
}

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

struct node* addition(struct node *L1,struct node *L2)
 {
   struct node *p1,*p2,*new, *L3,*temp;

   L3 = (struct node*)malloc(sizeof(struct node));
   L3->next=NULL;
   temp = L3;
   p1=L1->next;
   p2 = L2->next;
```

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

if(p1==NULL)
{
   while(p2!=NULL)
   {
      new = (struct node*)malloc(sizeof(struct node));
      new->coeff = p2->coeff;
```

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

struct node* subtraction(struct node *L1,struct node *L2)
 {
    struct node *p1,*p2,*new, *L3,*temp;

    L3 = (struct node*)malloc(sizeof(struct node));
    L3->next=NULL;
    temp = L3;
    p1=L1->next;
    p2 = L2->next;
```

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

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

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

struct node* multiplication(struct node *L1,struct node *L2)
{
    struct node *p1,*p2,*new, *L3,*temp;

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

```
        while(p2!=NULL)
        {
         new = (struct node*)malloc(sizeof(struct node));
         new->coeff = p2->coeff*p1->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,*p,*ref;
    temp = l3->next;
    while(temp!=NULL)
    {
        p = temp->next;
        ref =temp;
        while(p!=NULL)
        {
            if(temp->pow==p->pow)
            {
                temp->coeff = 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;
}

int main()
{
   struct node *l1 , *l2,*la,*ls,*lm;
   int n;

   printf("Enter the polynomials in Descending order only");
   printf("\nNO. of terms in polynomial 1:-");
   scanf("%d",&n);
   l1 = create_list(n);

   printf("\nNO. of terms in polynomial 2:-");
   scanf("%d",&n);
   l2 = create_list(n);

   printf("L1=");
   display(l1);
   printf("L2=");
   display(l2);

   printf("\nThe Sum of the two polynomial is=");
   la=addition(l1,l2);
   display(la);
```

```
    printf("The difference of the two polynomial is=");
    ls=subtraction(l1,l2);
    display(ls);

    printf("The product of the two polynomial is=");
    lm=multiplication(l1,l2);
    lm=add(lm);
    display(lm);
}
```

Output:

```
Enter the polynomials in Descending order only
NO. of terms in polynomial 1: 3
Enter the coefficient:- 2
Enter the power of x:- 3
Enter the coefficient:- 3
Enter the power of x:- 2
Enter the coefficient:- 4
Enter the power of x:- 1

NO. of terms in polynomial 2: 4
Enter the coefficient:- 1
Enter the power of x:- 4
Enter the coefficient:- 2
Enter the power of x:- 3
Enter the coefficient:- 5
Enter the power of x:- 2
Enter the coefficient:- 1
Enter the power of x:- 1

L1=2x^3+3x^2+4x^1
L2=1x^4+2x^3+5x^2+1x^1

The Sum of the two polynomial is=1x^4+4x^3+8x^2+5x^1
The difference of the two polynomial is=-1x^4+0x^3+-2x^2+3x^1
The product of the two polynomial is=2x^7+7x^6+20x^5+25x^4+23x^3+4x^2
```

Result:

This code is verified and implemented successfully.

Ex. No.: 4	Implementation of Stack using Array and Linked List Implementation	Date:11.05.2024

Write a C program to implement a stack using Array and linked List implementation and execute the following operation on stack.

(i) Push an element into a stack

(ii) Pop an element from a stack

(iii) Return the Top most element from a stack

(iv) Display the elements in a stack

Aim:

To write a c program to implement a stack using array and linked list implementation and execute the following operation on stack.

(i)Push an element into a stack

(ii) Pop an element from a stack

(iii) Return the Top most element from a stack

(iv) Display the elements in a 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 Itu data to be pushed. If the top is equal to capacity-1 display stack over flow otherwise increment the top and push the data onto Stack at index top.

5. To pop an element from a Stack if the is equal to-I display as stack underflow. Alter wise pop delta from Stack at index fop the decrement the top and display the popped data.

6. To return the pop most element from a Stack it the top is equal to-1 display Stack is empty. Otherwise display all elements in stack from top too.

7. After these operations display all elements in Stack from top too.

8. End.

STACK USING ARRAY

Program:

```
#include <stdio.h>
#define size 100

int stack[size];
int top = -1, i;

void push() {
   int a;
   printf("Enter the data to insert in stack : ");
   scanf("%d", &a);
   if (top == size - 1) {
      printf("Stack is full\n");
   } else {
      top = top + 1;
      stack[top] = a;
   }
}

void pop() {
   int a;
   if (top == -1) {
      printf("The stack is empty\n");
   } else {
      a = stack[top];
      printf("Item popped is : %d\n", a);
      top--;
   }
}
```

```
void peek() {
   if (top == -1) {
      printf("The stack is empty\n");
   } else {
      printf("The top most element in the stack is : %d\n", stack[top]);
   }
}

void display() {
   if (top == -1) {
      printf("Stack is empty\n");
   } else {
      printf("Items in the stack are : ");
      for (i = top; i >= 0; i--)
         printf("%d ", stack[i]);
      printf("\n");
   }
}

int main() {
   int choice;
   printf("STACK IMPLEMENTATION USING ARRAY\n");
   printf("1.Push\n2.Pop\n3.Peek\n4.Display\n5.Exit\n");
   do {
      printf("Enter Your Choice : ");
      scanf("%d", &choice);
      switch (choice) {
         case 1:
            push();
            break;
         case 2:
            pop();
            break;
```

```
            case 3:
                peek();
                break;
            case 4:
                display();
                break;
            case 5:
                printf("Exiting!!\n");
                break;
            default:
                printf("Invalid choice\n");
        }
    } while (choice != 5);
    return 0;
}
```

Output:

Operations:

1.Push

2.Pop

3.Peek

4.Display

5.Exit

Enter Your Choice : 1

Enter the data to insert in stack : 25

Enter Your Choice : 1

Enter the data to insert in stack : 45

Enter Your Choice : 1

Enter the data to insert in stack : 53

Enter Your Choice : 1

Enter the data to insert in stack : 24

Enter Your Choice : 4

Items in the stack are : 24 53 45 25

Enter Your Choice : 2

Item popped is : 24

Enter Your Choice : 3

The top most element in the stack is : 53

\

Enter Your Choice : 4

Items in the stack are : 53 45 25

Enter Your Choice : 5

Exiting!!

STACK USING LINKED LIST:

Program:

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

struct Node {
   int data;
   struct Node* next;
};

struct Node* top = NULL;

void push(int x) {
   struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
   if (temp == NULL) {
      printf("Stack Overflow\n");
      return;
   }
   temp->data = x;
   temp->next = top;
   top = temp;
}

void pop() {
   if (top == NULL) {
      printf("Stack Underflow\n");
      return;
   }
   struct Node* temp = top;
   top = top->next;
   free(temp);
}
```

```
int peek() {
   if (top == NULL) {
      printf("Stack is empty\n");
      return -1;
   }
   return top->data;
}

void display() {
   struct Node* temp = top;
   printf("Stack: ");
   while (temp != NULL) {
      printf("%d ", temp->data);
      temp = temp->next;
   }
   printf("\n");
}

int main() {
   int choice, item;
   printf("STACK IMPLEMENTATION USING LINKED LIST\n");
   printf("1. Push\n2. Pop\n3. Peek\n4. Display\n5. Exit\n");
   do {
      printf("Enter your choice: ");
      scanf("%d", &choice);
      switch (choice) {
         case 1:
            printf("Enter element to push: ");
            scanf("%d", &item);
            push(item);
            break;
         case 2:
            pop();
            break;
         case 3:
            printf("Top element: %d\n", peek());
            break;
```

```
            case 4:
                display();
                break;
            case 5:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice\n");
        }
    } while (choice != 5);
    return 0;
}
```

Output:

Operations:

1. Push
2. Pop
3. Peek
4. Display
5. Exit

Enter your choice: 1

Enter element to push: 43

Enter your choice: 1

Enter element to push: 32

Enter your choice: 4

Stack: 32 43

Enter your choice: 1

Enter element to push: 124

Enter your choice: 4

Stack: 124 32 43

Enter your choice: 2

Enter your choice: 3

Top element: 32

Enter your choice: 5

Exiting...

Result:

This code is verified and implemented successfully

Ex. No.: 5	Infix to Postfix Conversion	Date:11.05.2024

Write a C program to perform infix to postfix conversion using stack.

Aim:

To perform infix to postfix conversion using stack.

Algorithm:

1. Start

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

3. Define a function to determine the priority of operators.

4. Define a function to push a symbol onto the stack.

5. Define a function to pop a symbol from the stack. - Check if the stack is empty. If so, print "Stack Underflow" and exit.

6. Define a function to peek at the top symbol in the stack. - If the stack is empty, return null character '\0'.

7. Define a function to display the stack. - If the stack is empty, print "The stack is empty".

8. Define a function to check the precedence and process operators accordingly.

9. In the main function:
   a. Define variables for input string, symbol, and a stack. - Initialize the stack as NULL.
   b. Prompt the user to enter an expression. - Read the expression from the user.
   c. Iterate through each character in the expression:
      i. If it's an alphabet character, print it
      ii. If it's an operator, call the check_precedence function.
      iii. If it's a space, continue to the next character.
      iv. If it's an opening parenthesis '(', push it onto the stack.
      v. If it's a closing parenthesis ')', pop and print symbols from the stack until an opening parenthesis is encountered.
      vi. If it's any other character, print "Invalid expression" and break the loop.
   d. After the loop, pop and print any remaining symbols from the stack.

10. End.

Program:

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

struct Node
{
   char sign;
   struct Node* next;
};

int priority(char sym)
{
   if(sym=='*'||sym=='/'||sym=='%')
   return 2;
   else if(sym=='+'||sym=='-')
   return 1;
   else
   return 0;
}

void push(struct Node* L,char sym)
{
   struct Node*n = (struct Node*) malloc(sizeof(struct Node));
   if(n!=NULL)
   {
      n->sign=sym;
      if(L->next!=NULL)
      {
         n->next=L->next;
         L->next=n;
      }
```

```
        else
        L->next=n;
    }
}

void pop(struct Node* L)
{
    if(L->next==NULL)
    printf("\n---Stack Underflow---\n");
    else
    {
        struct Node* temp=L->next;
        L->next = L->next->next;
        free(temp);
    }
}

char peek(struct Node* L)
{
    if(L->next==NULL)
    return '\0';
    else
    return L->next->sign;
}

void display(struct Node* L)
{
    if(L->next==NULL)
    printf("\nThe stack is empty\n");
    else
    {
        struct Node* temp=L->next;
```

```
        while(temp!=NULL)
        {
            printf(" | %c |\n",temp->sign);
            temp=temp->next;
        }
    }
}

void check_precedence(struct Node*L,char in_exp)
{
    char in_stack=peek(L);

    if(priority(in_stack) >= priority(in_exp))
    {
        printf("%c",in_stack);
        pop(L);
        check_precedence(L,in_exp);
    }

    else
    {
        push(L,in_exp);
    }
}

int main() {
    char str[100],sym;
    struct Node list;
    list.next=NULL;
    printf("Enter expression containing lowercase alphabets and operators (+,-,*,/,%)\n");
    scanf("%[^\n]s",str);
```

```
for(int i=0;str[i]!='\0';i++)
{
   if((str[i]>='a'&& str[i]<='z')||(str[i]>='A' && str[i]<='Z'))
   {
      printf("%c",str[i]);
   }

   else if(str[i]=='+'||str[i]=='-'||str[i]=='*'||str[i]=='/')
   {
      check_precedence(&list,str[i]);
   }

   else if(str[i]==' ')
   continue;

   else if(str[i]=='(')
   push(&list,str[i]);

   else if(str[i]==')')
   {
      while(peek(&list)!='(')
      {
         printf("%c",peek(&list));
         pop(&list);
      }
      pop(&list);
   }
   else
   {
      printf("Invalid expression ");
      break;
   }
}
```

```
    while(list.next)
    {
        printf("%c",peek(&list));
        pop(&list);
    }
    return 0;
}
```

Output:

```
Enter expression containing lowercase alphabets and operators (+,-,*,/,%)
a+b*c/d-e
abc*d/+e-
```

Result:

Hence the code is implemented and executed successfully.

Ex. No.: 6	Evaluating Arithmetic Expression	Date:11.05.2024

Write a C program to evaluate Arithmetic expression using stack.

Aim:

To evaluate arithmetic expression using stack

Algorithm:

1.Start

2.Define an array `stack` to hold operands, and a character array `str` to store the input expression.

3.Define a function `push` to push an operand onto the stack.

4.Define a function `pop` to pop an operand from the stack.

5.Define a function `evaluate` to perform arithmetic operations on two operands.

6.In the `main` function:

- Declare variables `temp` and `i`.
- Prompt the user to enter an equation.
- Read the equation into the `str` array.
- Iterate through each character in `str`:
- If the character is a digit, update `temp` by multiplying its current value by 10 and adding the digit.
- If the character is a space, push the accumulated number `temp` onto the stack and reset `temp` to 0.
- If the character is an operator ('+', '-', '*', '/', '%'):
- Pop two operands (`num2` and `num1`) from the stack.
- Evaluate the expression (`num1 op num2`) using the operator and push the result onto the stack.
- Increment `i` to skip the next character (since it's an operator).
- Check if the last character in the expression is a digit or if there's only one element left in the stack. If not, print "Invalid expression" and exit.
- Print the final result, which is the only element left in the stack.
-

7.End.

Program:

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

int stack[30];
char str[30];
int top=-1;

void push(int num)
{
   top++;
   stack[top]=num;
}

int pop()
{
   int last=stack[top];
   top--;
   if(top!=-2)    return last;
   else
   {
      printf("Invalid expression...");
      exit(-1);
   }
}

int evaluate(int num1,int num2,char op)
{
   switch(op)
   {
      case '+':
      return num1+num2;
      break;
```

```
        case '-':
        return num1-num2;
        break;

        case '*':
        return num1*num2;
        break;

        case '/':
        return num1/num2;
        break;

        case '%':
        return num1%num2;
        break;
    }
}

void main()
{
    int temp=0,i;
    printf("Enter equation: ");
    scanf("%[^\n]s",str);
    for(i=0;str[i]!='\0';i++)
    {
        if(str[i]>='0'&& str[i]<='9')
        temp=temp*10+((int)(str[i]-'0'));

        else if(str[i]==' ')
        {
            push(temp);
            temp=0;
        }
```

```
        else if(str[i]=='+'||str[i]=='-'||str[i]=='*'||str[i]=='/'||str[i]=='%')
        {
            int num2=pop();
            int num1=pop();
            int result=evaluate(num1,num2,str[i]);
            push(result);
            i++;
        }
    }

    if(str[i-1]>='0'&& str[i+1]<='9' || top!=0)
    {
        printf("Invalid expression...");
        exit(-1);
    }

    printf("Result = %d",pop());
}
```

Output:

```
Enter equation: 5 3 * 8 -
Result = 7
```

Result:

Hence the code is implemented and executed successfully.

Ex. No.: 7	Implementation of Queue using Array and Linked List Implementation	Date:11.05.2024

Write a C program to implement a Queue using Array and linked List implementation and execute the following operation on stack.

(i) Enqueue

(ii) Dequeue

(iii) Display the elements in a Queue

Aim:

To implement queue using array and linked list and perform basic operations of queue.

Algorithm:

1. Linked List implementation

* Define a structure for node which contains two parts: data and a pointer to the next node.

* Create a new node and put the data in the data part of new node. New node should be added to the end of previous node.

* To remove an element, first check if the queue is empty or not, if it is not empty, remove the first element and move the pointer pointing to the first node to the next node.

2. Array Implementation

*Create an array of any size and new variables: front and rear with initial value -1.

* To insert the data, first check if rear is equal to max size of array or not. If not equal, increment rear and add the new value to array.

* To delete the data, first check if queue is empty or not. If it is not empty, print the value and increment the value of front.

Linked List implementation:

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

typedef struct node{
	int data;
	struct node* next;
}Node;

void enqueue(Node** queue,int val){
	Node* newnode=malloc(sizeof(Node));
	Node* cur;
	if(newnode){

		newnode->data =val;
		newnode->next = NULL;
		if(*queue==NULL){
			*queue = newnode;
			cur =*queue;
		}
		else{
			cur->next= newnode;
			cur = newnode;
		}
	}
	else{
		printf("\nMemory allocation failed\n");
		exit(1);
	}
}

void peek(Node* queue){
	if(queue){
		printf("\nThe first element in the queue is %d",queue->data);
	}
	else{
		printf("\nThe queue is empty");
		return;
	}
}

int dequeue(Node** queue){
	if(*queue){
		Node* temp = *queue;
		(*queue) = (*queue)->next;
		int x = temp->data;
		free(temp);
		return x;
	}
```

```
        else{
                printf("\nQueue is empty");
                return -1;
        }
}


void display(Node* queue){
        if(queue){
                while(queue){
                        printf("%d\t",queue->data);
                        queue = queue->next;
                }
        }
        else{
                printf("\nThe queue is empty");
                return;
        }
}

int main(){
        Node* queue= NULL;
        int n,choice;
        do{
                printf("\n1. ENQUEUE\n2. DEQUEUE \n3. PEEK \n4. DISPLAY \n5.EXIT");
                printf("\nENTER YOUR CHOICE: ");
                scanf("%d",&choice);
                switch(choice){
                        case 1:{
                                printf("\nEnter the element to enqueue: ");
                                scanf("%d",&n);
                                enqueue(&queue,n);
                                break;
                        }
                        case 2:{
                                n = dequeue(&queue);
                                if(n!=-1){
                                        printf("\nThe dequeued element is %d",n);
                                        break;
                                }
                                }
                        case 3: peek(queue); break;
                        case 4: display(queue); break;
                }

        }while(choice<=4);

        return 0;
}
```

Output:

```
1. ENQUEUE
2. DEQUEUE
3. PEEK
4. DISPLAY
5.EXIT
ENTER YOUR CHOICE: 1

Enter the element to enqueue: 2

1. ENQUEUE
2. DEQUEUE
3. PEEK
4. DISPLAY
5.EXIT
ENTER YOUR CHOICE: 1

Enter the element to enqueue: 4

1. ENQUEUE
2. DEQUEUE
3. PEEK
4. DISPLAY
5.EXIT
ENTER YOUR CHOICE: 1

Enter the element to enqueue: 5

1. ENQUEUE
2. DEQUEUE
3. PEEK
4. DISPLAY
5.EXIT
ENTER YOUR CHOICE: 1

Enter the element to enqueue: 6

1. ENQUEUE
2. DEQUEUE
3. PEEK
4. DISPLAY
5.EXIT
ENTER YOUR CHOICE: 2

The dequeued element is 2
1. ENQUEUE
2. DEQUEUE
3. PEEK
4. DISPLAY
5.EXIT
ENTER YOUR CHOICE: 3
```

```
The first element in the queue is 4
1. ENQUEUE
2. DEQUEUE
3. PEEK
4. DISPLAY
5.EXIT
ENTER YOUR CHOICE: 4
4       5       6

1. ENQUEUE
2. DEQUEUE
3. PEEK
4. DISPLAY
5.EXIT
ENTER YOUR CHOICE: 5
```

Array Implementation:

```
#include<stdio.h>
#define MAX 10
int queue[MAX],front=-1,rear = -1;

void enqueue(int v){
	if(front == MAX-1){
		printf("\nQueue is full");
		return;
	}
	queue[++rear] = v;
	if(front == -1){
		front++;
	}
}

int dequeue(){
	if(front ==-1){
		printf("\nQueue is empty");
		return -1;
	}
	int temp = queue[front];
	if(front == rear){
		front = -1;
		rear = -1;
	}
	front++;
	return temp;
}

void peek(){
	if(front ==-1){
		printf("\nQueue is empty");
		return;
	}
	printf("\nThe first element in the queue is %d",queue[front]);

}

void display(){
	if(front ==-1){
		printf("\nQueue is empty");
		return;
	}
	printf("\nGiven queue is ");
	for (int i =front;i<=rear;i++)
		printf("%d\t",queue[i]);
	printf("\n");

}
```

```
int main(){
	int n,choice;
	do{
		printf("\n1. ENQUEUE\n2. DEQUEUE \n3. PEEK \n4. DISPLAY \n5.EXIT");
		printf("\nENTER YOUR CHOICE: ");
		scanf("%d",&choice);
		switch(choice){
			case 1:{
				printf("\nEnter the element to enqueue: ");
				scanf("%d",&n);
				enqueue(n);
				break;
			}
			case 2:{
				n = dequeue();
				if(n!=-1){
					printf("\nThe dequeued element is %d",n);
					break;
				}
				}
			case 3: peek(); break;
			case 4: display(); break;
		}

	}while(choice<=4);
}
```

OUTPUT:

```
1. ENQUEUE
2. DEQUEUE
3. PEEK
4. DISPLAY
5.EXIT
ENTER YOUR CHOICE: 1

Enter the element to enqueue: 2

1. ENQUEUE
2. DEQUEUE
3. PEEK
4. DISPLAY
5.EXIT
ENTER YOUR CHOICE: 1

Enter the element to enqueue: 4

1. ENQUEUE
2. DEQUEUE
3. PEEK
4. DISPLAY
5.EXIT
ENTER YOUR CHOICE: 1

Enter the element to enqueue: 6

1. ENQUEUE
2. DEQUEUE
3. PEEK
4. DISPLAY
5.EXIT
ENTER YOUR CHOICE: 4

Given queue is 2        4        6

1. ENQUEUE
2. DEQUEUE
3. PEEK
4. DISPLAY
5.EXIT
ENTER YOUR CHOICE: 2
The dequeued element is 2

1. ENQUEUE
2. DEQUEUE
3. PEEK
4. DISPLAY
5.EXIT
ENTER YOUR CHOICE: 3
The first element in the queue is 4
```

1. ENQUEUE
2. DEQUEUE
3. PEEK
4. DISPLAY
5.EXIT
ENTER YOUR CHOICE: 4

Given queue is 4 6

1. ENQUEUE
2. DEQUEUE
3. PEEK
4. DISPLAY
5.EXIT
ENTER YOUR CHOICE: 5

Result:

Hence, the implementation of queue using linked list and array is done and verified.

Ex. No.: 8	Tree Traversal	Date:18.05.2024

Write a C program to implement a Binary tree and perform the following tree traversal operation.

(i) Inorder Traversal

(ii) Preorder Traversal

(iii) Postorder Traversal

Aim:

Performing tree traversal techniques

Algorithm:

1. Start
2. Create a tree node
3. **In-order**

   Traverse the left subtree, visit the root node, and then traverse the right subtree.
4. **Pre-Order Traversal**

   Visit the root node, traverse the left subtree, and then traverse the right subtree.
5. **Post-Order Traversal**

   Traverse the left subtree, traverse the right subtree, and then visit the root node.
6. Than build a tree.
7. Int main() function perform the tree traversal techniques.
8. Stop

Program:

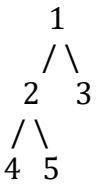
```
#include <stdio.h>
#include <stdlib.h>

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

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

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

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

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

struct TreeNode* buildTree() {
    int data;
    struct TreeNode *root = NULL;

    printf("Enter data (Enter -1 to stop): ");
    scanf("%d", &data);

    if (data == -1) return NULL;
    root = createNode(data);

    printf("Enter left child of %d:\n", data);
    root->left = buildTree();

    printf("Enter right child of %d:\n", data);
    root->right = buildTree();

    return root;
}

int main() {
    printf("Enter the binary tree (Enter -1 to stop):\n");
    struct TreeNode* root = buildTree();

    printf("\nIn-order traversal: ");
    inorderTraversal(root);
    printf("\n");
```

```
    printf("Pre-order traversal: ");
    preorderTraversal(root);
    printf("\n");

    printf("Post-order traversal: ");
    postorderTraversal(root);
    printf("\n");

    return 0;
}
```

Output:

```
Enter the binary tree (Enter -1 to stop):
Enter data (Enter -1 to stop): 1
Enter left child of 1:
Enter data (Enter -1 to stop): 2
Enter left child of 2:
Enter data (Enter -1 to stop): 4
Enter left child of 4:
Enter data (Enter -1 to stop): -1
Enter right child of 4:
Enter data (Enter -1 to stop): -1
Enter right child of 2:
Enter data (Enter -1 to stop): 5
Enter left child of 5:
Enter data (Enter -1 to stop): -1
Enter right child of 5:
Enter data (Enter -1 to stop): -1
Enter right child of 1:
Enter data (Enter -1 to stop): 3
Enter left child of 3:
Enter data (Enter -1 to stop): -1
Enter right child of 3:
Enter data (Enter -1 to stop): -1


In-Order Traversal: 4 2 5 1 3
Pre-Order Traversal: 1 2 4 5 3
Post-Order Traversal: 4 5 2 3 1
```



Result:

Hence, the tree traversal code is implemented and executed successfully.

Ex. No.: 9	Implementation of Binary Search tree	Date:18.05.2024

Write a C program to implement a Binary Search Tree and perform the following operations.

(i) Insert

(ii) Delete

(iii) Search

(iv) Display

Aim:

Implement a binary search tree to perform insertion,deletion,search and display operation.

Algorithm:

1. Start: Begin.
2. Define Node Structure: Define node structure.
3. Create New Node: Function to create a new node.
4. Insert Node: Insert a new node while maintaining BST property.
5. Delete Node: Delete a node while preserving BST property.
6. Find Minimum Value: Find the minimum value in a subtree.
7. Search Node: Recursive search function.
8. Inorder Traversal: Print nodes in sorted order.
9. End: Finish.

Program:

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

struct node {
   int data;
   struct node *leftChild, *rightChild;
};
struct node *root = NULL;

struct node *newNode(int item) {
   struct node *temp = (struct node *)malloc(sizeof(struct node));
   if (!temp) {
      printf("Memory allocation failed\n");
      exit(1);
   }
   temp->data = item;
   temp->leftChild = temp->rightChild = NULL;
   return temp;
}

void insert(int data) {
   struct node *tempNode = newNode(data);
   struct node *current;
   struct node *parent;
   if (root == NULL) {
      root = tempNode;
      return;
   }
   current = root;
   parent = NULL;
```

```
   while (1) {
      parent = current;
      if (data < parent->data) {
         current = current->leftChild;
         if (current == NULL) {
            parent->leftChild = tempNode;
            return;
         }
      }
      else {
         current = current->rightChild;
         if (current == NULL) {
            parent->rightChild = tempNode;
            return;
         }
      }
   }
}

struct node* search(int data) {
   struct node *current = root;
   while (current != NULL && current->data != data) {
      if (data < current->data) {
         current = current->leftChild;
      } else {
         current = current->rightChild;
      }
   }
   return current;
}
```

```
void printTree(struct node* Node) {
   if (Node == NULL)      return;
   printTree(Node->leftChild);
   printf(" %d", Node->data);
   printTree(Node->rightChild);
}

struct node* findMin(struct node* node) {
   struct node* current = node;
   while (current && current->leftChild != NULL)
      current = current->leftChild;
   return current;
}

struct node* deleteNode(struct node* root, int data) {
   if (root == NULL) return root;
   if (data < root->data)
      root->leftChild = deleteNode(root->leftChild, data);
   else if (data > root->data)
      root->rightChild = deleteNode(root->rightChild, data);
   else {
      if (root->leftChild == NULL) {
         struct node *temp = root->rightChild;
         free(root);
         return temp;
      } else if (root->rightChild == NULL) {
         struct node *temp = root->leftChild;
         free(root);
         return temp;
      }
      struct node* temp = findMin(root->rightChild);
      root->data = temp->data;
      root->rightChild = deleteNode(root->rightChild, temp->data);
   }
   return root;
}
```

```
int main() {
    int choice, data;
    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert\n");
        printf("2. Search\n");
        printf("3. Delete\n");
        printf("4. Print In-order\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                insert(data);
                printf("Insertion done\n");
                break;
            case 2:
                printf("Enter data to search: ");
                scanf("%d", &data);
                struct node *k;
                k = search(data);
                if (k != NULL)
                    printf("Element %d found\n", k->data);
                else
                    printf("Element not found\n");
                break;
            case 3:
                printf("Enter data to delete: ");
                scanf("%d", &data);
                root = deleteNode(root, data);
                printf("Deletion done\n");
                break;
```

```
            case 4:
                printf("BST in-order:\n");
                printTree(root);
                printf("\n");
                break;

            case 5:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }
    return 0;
}
```

Output:

```
Menu:
1. Insert
2. Search
3. Delete
4. Print In-order
5. Exit
Enter your choice: 1
Enter data to insert: 2
Insertion done

Menu:
1. Insert
2. Search
3. Delete
4. Print In-order
5. Exit
Enter your choice: 1
Enter data to insert: 3
Insertion done

Menu:
1. Insert
2. Search
3. Delete
4. Print In-order
5. Exit
Enter your choice: 1
Enter data to insert: 4
Insertion done

Menu:
1. Insert
2. Search
3. Delete
4. Print In-order
5. Exit
Enter your choice: 1
Enter data to insert: 1
Insertion done

Menu:
1. Insert
2. Search
3. Delete
4. Print In-order
5. Exit
Enter your choice: 4
BST in-order:
 1 2 3 4
```

```
Menu:
1. Insert
2. Search
3. Delete
4. Print In-order
5. Exit
Enter your choice: 2
Enter data to search: 2
Element 2 found at address 0x1820ac0

Menu:
1. Insert
2. Search
3. Delete
4. Print In-order
5. Exit
Enter your choice: 3
Enter data to delete: 3
Deletion done

Menu:
1. Insert
2. Search
3. Delete
4. Print In-order
5. Exit
Enter your choice: 4
BST in-order:
 1 2 4

Menu:
1. Insert
2. Search
3. Delete
4. Print In-order
5. Exit
Enter your choice: 5
```

Result:

Thus the code is implemented and executed successfully.

Ex. No.: 10	Implementation of AVL Tree	Date:18.05.2024

Write a function in C program to insert a new node with a given value into an AVL tree. Ensure that the tree remains balanced after insertion by performing rotations if necessary. Repeat the above operation to delete a node from AVL tree.

Aim:

Design and implement functions in C to insert and delete nodes in an AVL tree, ensuring the tree remains balanced by performing necessary rotations.

Algorithm:

1. Define the structure for an AVL tree node.
2. Implement a function to calculate the height of a node.
3. Implement a function to calculate the maximum of two integers.
4. Implement a function to create a new node with a given key.
5. Implement a function to perform a right rotation on a subtree.
6. Implement a function to perform a left rotation on a subtree.
7. Implement a function to get the balance factor of a node.
8. Implement a function to insert a new key into the AVL tree:
   a. Recursively find the correct position for the new key.
   b. Insert the node and update the height.
   c. Check the balance factor and perform necessary rotations.
9. Implement a function to delete a key from the AVL tree:
   a. Recursively find the node to be deleted.
   b. Handle cases with one or no children.
   c. Use in-order successor for nodes with two children.
   d. Update heights and balance the tree by performing necessary rotations.
10. Implement a function to print the tree in pre-order traversal to display the structure.

Program:

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

struct Node {
   int key;
   struct Node *left;
   struct Node *right;
   int height;
};

int height(struct Node *N) {
   if (N == NULL)
      return 0;
   return N->height;
}

int max(int a, int b) {
   return (a > b) ? a : b;
}

struct Node* newNode(int key) {
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
   node->key = key;
   node->left = NULL;
   node->right = NULL;
   node->height = 1;
   return(node);
}

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

struct Node* insert(struct Node* node, int key) {
   if (node == NULL)
      return(newNode(key));
   if (key < node->key)
      node->left = insert(node->left, key);
   else if (key > node->key)
      node->right = insert(node->right, key);
   else
      return node;
   node->height = 1 + max(height(node->left), height(node->right));
   int balance = getBalance(node);
   if (balance > 1 && key < node->left->key)
      return rightRotate(node);
   if (balance < -1 && key > node->right->key)
      return leftRotate(node);
```

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

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

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

```
      struct Node *temp = minValueNode(root->right);
      root->key = temp->key;
      root->right = deleteNode(root->right, temp->key);
    }
    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 preOrder(struct Node *root) {
    if (root != NULL) {
      printf("%d ", root->key);
      preOrder(root->left);
      preOrder(root->right);
    }
}
```

```
int main() {
   struct Node *root = NULL;
   int value, c, a;
   while (1) {
      printf("\nOperations:\n");
      printf("1. Insert element\n");
      printf("2. Delete element\n");
      printf("3. Display tree\n");
      printf("4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &c);

     switch (c) {
         case 1:
            printf("Enter numbers to insert into the tree (enter -1 to stop): ");
            while (1) {
               scanf("%d", &value);
               if (value == -1)
                  break;
               root = insert(root, value);
            }
            break;
         case 2:
            printf("Enter the number to delete: ");
            scanf("%d", &a);
            root = deleteNode(root, a);
            break;
         case 3:
            printf("Pre-order traversal of the constructed AVL tree: ");
            preOrder(root);
            printf("\n");
            break;
         case 4:
            printf("Exiting...\n");
            exit(0);
```

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

Output:

```
Operations:
1. Insert element
2. Delete element
3. Display tree
4. Exit
Enter your choice: 1
Enter numbers to insert into the tree (enter -1 to stop): 10 5 15 20 -1

Operations:
1. Insert element
2. Delete element
3. Display tree
4. Exit
Enter your choice: 3
Pre-order traversal of the constructed AVL tree: 10 5 15 20

Operations:
1. Insert element
2. Delete element
3. Display tree
4. Exit
Enter your choice: 2
Enter the number to delete: 5

Operations:
1. Insert element
2. Delete element
3. Display tree
4. Exit
Enter your choice: 3
Pre-order traversal of the constructed AVL tree: 15 10 20

Operations:
1. Insert element
2. Delete element
3. Display tree
4. Exit
Enter your choice: 2
Enter the number to delete: 10
```

Operations:
1. Insert element
2. Delete element
3. Display tree
4. Exit
Enter your choice: 3
Pre-order traversal of the constructed AVL tree: 15 20

Operations:
1. Insert element
2. Delete element
3. Display tree
4. Exit
Enter your choice: 4
Exiting...

Result:

Thus the code is implemented and executed successfully.

Ex. No.: 11	Graph Traversal	Date:25.05.2024

Write a C program to create a graph and perform a Breadth First Search and Depth First Search.

Aim:

To create a graph and perform a Breadth First Search and Depth First Search.

Algorithm:

1. Start
2. Create a node which contains vertex and next as their members.
3. Allocates memory dynamically for nodes and the graph structure using malloc().
4. Create a graph with a specified number of vertices and initialize adjacency lists.
5. Create a function to add the edges between the vertices
6. Performs BFS traversal starting from a given vertex using a queue data structure to maintain order.
7. Conducts DFS traversal starting from a specified vertex, employing recursion to exploregraph branches.
8. Displays the adjacency list representation of the graph and the traversal sequences. for both BFS and DFS.
9. End

Program:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

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

struct Node* createNode(int v);

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

struct Graph* createGraph(int vertices);

void addEdge(struct Graph* graph, int src, int dest);
void printGraph(struct Graph* graph);
void BFS(struct Graph* graph, int startVertex);
void DFS(struct Graph* graph, int startVertex);
void DFSUtil(struct Graph* graph, int vertex);

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

```
    printf("Graph:\n");
    printGraph(graph);

    printf("\nBFS Traversal:\n");
    BFS(graph, 2);

    printf("\nDFS Traversal:\n");
    DFS(graph, 2);
    return 0;
}

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

struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->adjLists = (struct Node**)malloc(vertices * sizeof(struct Node*));
    graph->visited = (int*)malloc(vertices * sizeof(int));

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

void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
```

```
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

void printGraph(struct Graph* graph) {
    for (int v = 0; v < graph->numVertices; v++) {
        struct Node* temp = graph->adjLists[v];
        printf("Vertex %d: ", v);
        while (temp) {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}

void BFS(struct Graph* graph, int startVertex) {
    int queue[MAX];
    int front = 0, rear = 0;

    graph->visited[startVertex] = 1;
    queue[rear++] = startVertex;

    while (front < rear) {
        int currentVertex = queue[front++];
        printf("Visited %d\n", currentVertex);

        struct Node* temp = graph->adjLists[currentVertex];
        while (temp) {
            int adjVertex = temp->vertex;
            if (graph->visited[adjVertex] == 0) {
                graph->visited[adjVertex] = 1;
                queue[rear++] = adjVertex;
            }
```

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

void DFSUtil(struct Graph* graph, int vertex) {
    struct Node* temp = graph->adjLists[vertex];
    graph->visited[vertex] = 1;
    printf("Visited %d\n", vertex);

    while (temp) {
        int adjVertex = temp->vertex;
        if (graph->visited[adjVertex] == 0) {
            DFSUtil(graph, adjVertex);
        }
        temp = temp->next;
    }
}

void DFS(struct Graph* graph, int startVertex) {
    for (int i = 0; i < graph->numVertices; i++) {
        graph->visited[i] = 0;
    }
    DFSUtil(graph, startVertex);
}
```

Output:

Graph:

Vertex 0: 2->2->1-> NULL

Vertex 1: 2->0-> NULL

Vertex 2: 3->0->1->0-> NULL

Vertex 3: 3->3->2 -> NULL

BFS Traversal:

Visited 2

Visited 3

Visited 0

Visited 1

DFS Traversal:

Visited 2

Visited 3

Visited 0

Visited 1

Result:

Thus the program successfully implemented and executed.

Ex. No.: 12	Topological Sorting	Date:25.05.2024

Write a C program to create a graph and display the ordering of vertices.

Aim:

Write a C program to create a graph and display the ordering of vertices (TOPOLOGICAL SORT)

Algorithm:

Step 1: Find the indegree for every vertex.

Step 2: Place the vertices whose indegree is 0 on the empty queue.

Step 3: Dequeue the vertex v and decrement the indegree of all its adjacent vertices.

Step 4: Enqueue the vertex on the queue if its indegree falls to zero.

Step 5: Repeat from step 3 until the queue becomes empty.

Step 6: The topological ordering is the order in which the vertices dequeue.

Program:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node Node;
typedef struct Node* pos;
typedef struct Node* Vert;
typedef struct Graph_parts* Graph;
typedef int Queue;

int front=-1,rear=-1;
Queue Q[50];

void enqueue(int n)
{
   if(front==-1)
      front++;
   rear++;
   Q[rear]=n;
}

int dequeue()
{
   int p=Q[front];
   if(front==rear)
   {
      front=-1;
      rear=-1;
   }
   else
      front++;
   return p;
}
```

```
struct Node
{
   int data;
   pos next;
};

struct Graph_parts
{
   int vertices;
   int edges;
   Vert* List;
};

void Link(Graph G, int v1, int v2)
{
   Vert n = (Vert)malloc(sizeof(Node));
   if(n!=NULL)
   {
      n->data = v2;
      pos p=G->List[v1];
      n->next=p->next;
      p->next=n;
   }
}

void Get_edges(Graph G)
{
   int v1,v2,cost;
   printf("Enter the edges:\n");
   for(int i=0;i<G->edges;i++)
   {
      printf("=> ");
      scanf("%d %d", &v1, &v2);
      Link(G,v1,v2);
   }
}
```

```
Graph Create_graph()
{
   Graph G = (Graph)malloc(sizeof(struct Graph_parts));
   if(G!=NULL)
   {
      printf("Enter the number of vertices in graph: ");
      scanf("%d", &G->vertices);
      printf("Vertices are from 0 to %d\n",G->vertices-1);

      printf("\nEnter the number of edges: ");
      scanf("%d",&G->edges);

      G->List = (Vert*)malloc(sizeof(Node) * G->vertices);
      if(G->List!=NULL)
      {
         for (int i = 0; i < G->vertices; i++)
         {
            G->List[i] = (Vert)malloc(sizeof(struct Node));
         }
      }
      Get_edges(G);
      return G;
   }
}

void Display(Graph G)
{
   printf("\nDisplaying the Graph using List\n\n");
   for (int i = 0; i < G->vertices; i++)
   {
      struct Node* p = G->List[i];
      printf("%d => ", i);
      p=p->next;
```

```
      while (p != NULL)
      {
         printf("%d ", p->data);
         p = p->next;
      }
      printf("\n");
   }
}

void mark_all_zeros(int v[],int in[],int size)
{
   for(int i=0;i<size;i++)
   {
      in[i]=0;
      v[i]=0;
   }
}

void find_indegree(Graph G,int in_deg[])
{
   pos p;
   for(int i=0;i<G->vertices;i++)
   {
      p=G->List[i]->next;
      while(p!=NULL)
      {
         in_deg[p->data]++;
         p=p->next;
      }
   }
}
```

```
void Topo_sort(Graph G,int vis[],int in_deg[])
{
   int arr[G->vertices];
   int s=0;
   int i,j=0;
   while(j<G->vertices)
   {
      for(i=0;i<G->vertices;i++)
      {
         if(in_deg[i]==0 && vis[i]==0)
         {
            enqueue(i);
            vis[i]=1;
         }
      }
      if(front!=-1)
      {
         int n=dequeue();
         arr[s]=n;
         s++;
         pos p=G->List[n]->next;
         while(p!=NULL)
         {
            in_deg[p->data]--;
            p=p->next;
         }
      }
      else
         break;
      j++;
   }
   if(j==G->vertices)
   {
      printf("\nTopological sort for the given graph is:\n");
```

```
        for(int k=0;k<G->vertices;k++)
        {
           printf("%d ",arr[k]);
        }
    }
    else
    {
       printf("Graph has a cycle");
    }
}

void main()
{
   Graph G=Create_graph();
   Display(G);
   int visited[G->vertices];
   int indegree[G->vertices];
   mark_all_zeros(visited,indegree,G->vertices);
   find_indegree(G,indegree);
   Topo_sort(G,visited,indegree);
}
```

Output:

Enter the number of vertices in graph: 3

Vertices are from 0 to 2

Enter the number of edges: 3

Enter the edges:

=> 0 1

=> 0 2

=> 1 2

Displaying the Graph using List

0 => 2 1

1 => 2

2 =>

Topological sort for the given graph is:

0 1 2

Result:

Thus the program successfully implemented and executed.

Ex. No.: 13	Graph Traversal	Date:25.05.24

Write a C program to create a graph and find a minimum spanning tree using prims algorithm.

Aim:

To write a c program to create a graph and find a minimum spanning tree using prim's algorithm.

Algorithm:

1. Initialize `key[]` to `INT_MAX`, `key[0]` to 0, `mstSet[]` to `false`, and `parent[]` to `-1`.
2. For count` from 0 to `numvertex 1:
   - Select vertex `u` with minimum `key[u]` that is not `mstSet`.
   - Set `mstSet[u]` to `true`.
   - For each adjacent vertex `v` of `u`:

If `weight(u, v) < key [v]` and `v` is not in `mstset`, update `key[v]` and set `parent[v]` to `u`.

3. Repeat until all vertices are processed.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define infinity 10000;

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

struct graph {
   int numvertex;
   struct node **adjlist;
};

struct node* create_node(int vertex, int w) {
   struct node *new = (struct node*)malloc(sizeof(struct node));
   if (!new) {
      printf("Memory allocation error\n");
      exit(EXIT_FAILURE);
   }
   new->vertex = vertex;
   new->w = w;
   new->next = NULL;
   return new;
}

struct graph* create_graph(int no) {
   struct graph *new = (struct graph*)malloc(sizeof(struct graph));
   if (!new) {
      printf("Memory allocation error\n");
      exit(EXIT_FAILURE);
   }
```

```
    new->numvertex = no;
    new->adjlist = (struct node**)malloc(no * sizeof(struct node*));
    if (!new->adjlist) {
        printf("Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < no; i++) {
        new->adjlist[i] = NULL;
    }
    return new;
}

void add_edge(struct graph *g, int s, int d, int w) {
    struct node *new = create_node(d, w);
    new->next = g->adjlist[s];
    g->adjlist[s] = new;
    new = create_node(s, w);
    new->next = g->adjlist[d];
    g->adjlist[d] = new;
}

struct graph* input(struct graph *g) {
    int no, d, s, w;
    printf("ENTER THE NO. OF EDGES: ");
    scanf("%d", &no);
    for (int i = 0; i < no; i++) {
        printf("Enter edge (source destination weight): ");
        scanf("%d %d %d", &s, &d, &w);
        add_edge(g, s, d, w);
    }
    return g;
}

void print_graph(struct graph *g) {
    for (int i = 0; i < g->numvertex; i++) {
        struct node *temp = g->adjlist[i];
```

```
      printf("VERTEX %d\n", i);
       while (temp != NULL) {
          printf("%d(%d) --> ", temp->vertex, temp->w);
          temp = temp->next;
       }
       printf("NULL\n");
    }
}

void free_graph(struct graph *g) {
    for (int i = 0; i < g->numvertex; i++) {
       struct node *temp = g->adjlist[i];
       while (temp != NULL) {
          struct node *to_free = temp;
          temp = temp->next;
          free(to_free);
       }
    }
    free(g->adjlist);
    free(g);
}

void prim_mst(struct graph *g) {
    int numvertex = g->numvertex;
    int path[numvertex];
    int dist[numvertex];
    bool known[numvertex];
    for (int i = 0; i < numvertex; i++) {
       dist[i] = infinity;
       known[i] = false;
    }
    dist[0] = 0;
    path[0] = -1;
```

```
  for (int count = 0; count < numvertex - 1; count++) {
     int u = -1;
     int min = infinity;
     for (int v = 0; v < numvertex; v++) {
        if (!known[v] && dist[v] < min) {
           min = dist[v];
           u = v;
        }
     }
     if (u == -1) {
        printf("Graph is disconnected\n");
        return;
     }
     known[u] = true;
     struct node *temp = g->adjlist[u];
     while (temp != NULL) {
        int v = temp->vertex;
        if (!known[v] && temp->w < dist[v]) {
           path[v] = u;
           dist[v] = temp->w;
        }
        temp = temp->next;
     }
  }
  printf("Path \tDistance\n");
  for (int i = 1; i < numvertex; i++) {
     if (path[i] != -1) {
        printf("%d - %d \t%d \n", path[i], i, dist[i]);
     }
  }
}
```

```
int main() {
   int n;
   printf("ENTER THE NO. OF NODES: ");
   scanf("%d", &n);
   if (n <= 0) {
      printf("Number of nodes must be greater than 0\n");
      return -1;
   }
   struct graph *g = create_graph(n);
   input(g);
   print_graph(g);
   printf("\n\nMinimum Spanning Tree (MST) using Prim's Algorithm:\n\n");
   prim_mst(g);
   free_graph(g);
   return 0;
}
```

Output:

```
ENTER THE NO. OF NODES: 7
ENTER THE NO. OF EDGES: 9

Enter edge (source destination weight): 0 1 28
Enter edge (source destination weight): 1 2 16
Enter edge (source destination weight): 2 3 12
Enter edge (source destination weight): 3 4 22
Enter edge (source destination weight): 4 5 25
Enter edge (source destination weight): 5 0 10
Enter edge (source destination weight): 4 6 24
Enter edge (source destination weight): 2 6 18
Enter edge (source destination weight): 6 1 14

VERTEX 0
5(10) --> 1(28) --> NULL
VERTEX 1
6(14) --> 2(16) --> 0(28) --> NULL
VERTEX 2
6(18) --> 3(12) --> 1(16) --> NULL
VERTEX 3
4(22) --> 2(12) --> NULL
VERTEX 4
6(24) --> 5(25) --> 3(22) --> NULL
VERTEX 5
0(10) --> 4(25) --> NULL
VERTEX 6
1(14) --> 2(18) --> 4(24) --> NULL


Minimum Spanning Tree (MST) using Prim's Algorithm:

Path    Distance
2 - 1   16
3 - 2   12
4 - 3   22
5 - 4   25
0 - 5   10
1 - 6   14
```

Result:

Thus, the code is implemented and executed successfully.

Ex. No.: 14	Graph Traversal	Date:1.06.2024

Write a C program to create a graph and find the shortest path using Dijikstra's Algorithm.

Aim:

To write a c program to create a graph and find the shortest path using dijikstras algorithm.

Algorithm:

1. Initialize `dist[] to `INT_MAX`, `dist[src]` to 0, and `known []` to `false`.

2. For count` from 0 to `numvertex 1:

- Select vertex `u` with minimum `dist[u]` that is not `known`.
- Set `known [u]` to `true`.
- For each adjacent vertex `v` of `u`:

If`dist[u] + weight(u, v) < dist[v]`, update `dist[v]` and set `path[v]` to `u`.

3. Repeat until all vertices are processed

Program:

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

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

struct graph {
   int numvertex;
   struct node **adjlist;
};

struct node* create_node(int vertex, int w) {
   struct node *new = (struct node*)malloc(sizeof(struct node));
   if (!new) {
      fprintf(stderr, "Memory allocation error\n");
      exit(EXIT_FAILURE);
   }
   new->vertex = vertex;
   new->w = w;
   new->next = NULL;
   return new;
}

struct graph* create_graph(int no) {
   struct graph *new = (struct graph*)malloc(sizeof(struct graph));
   if (!new) {
      fprintf(stderr, "Memory allocation error\n");
      exit(EXIT_FAILURE);
   }
```

```
    new->numvertex = no;
    new->adjlist = (struct node**)malloc(no * sizeof(struct node*));
    if (!new->adjlist) {
        fprintf(stderr, "Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < no; i++) {
        new->adjlist[i] = NULL;
    }
    return new;
}

void add_edge(struct graph *g, int s, int d, int w) {
    struct node *new = create_node(d, w);
    new->next = g->adjlist[s];
    g->adjlist[s] = new;
    new = create_node(s, w);
    new->next = g->adjlist[d];
    g->adjlist[d] = new;
}

struct graph* input(struct graph *g) {
    int no, d, s, w;
    printf("ENTER THE NO. OF EDGES: ");
    scanf("%d", &no);
    for (int i = 0; i < no; i++) {
        printf("Enter edge (source destination weight): ");
        scanf("%d %d %d", &s, &d, &w);
        add_edge(g, s, d, w);
    }
    return g;
}
```

```
void print_graph(struct graph *g) {
   for (int i = 0; i < g->numvertex; i++) {
      struct node *temp = g->adjlist[i];
      printf("VERTEX %d\n", i);
      while (temp != NULL) {
         printf("%d(%d) --> ", temp->vertex, temp->w);
         temp = temp->next;
      }
      printf("NULL\n");
   }
}

void free_graph(struct graph *g) {
   for (int i = 0; i < g->numvertex; i++) {
      struct node *temp = g->adjlist[i];
      while (temp != NULL) {
         struct node *to_free = temp;
         temp = temp->next;
         free(to_free);
      }
   }
   free(g->adjlist);
   free(g);
}

void print_path(int path[], int vertex) {
   if (path[vertex] == -1) {
      printf("%d", vertex);
      return;
   }
   print_path(path, path[vertex]);
   printf(" -> %d", vertex);
}
```

```
void dijkstra(struct graph *g, int src) {
   int numvertex = g->numvertex;
   int dist[numvertex];
   bool known[numvertex];
   int path[numvertex];
   for (int i = 0; i < numvertex; i++) {
      dist[i] = INT_MAX;
      known[i] = false;
      path[i] = -1;
   }
   dist[src] = 0;
   for (int count = 0; count < numvertex - 1; count++) {
      int u = -1;
      int min = INT_MAX;
      for (int v = 0; v < numvertex; v++) {
         if (!known[v] && dist[v] < min) {
            min = dist[v];
            u = v;
         }
      }
      if (u == -1) {
         printf("Graph is disconnected\n");
         return;
      }
      known[u] = true;
      struct node *temp = g->adjlist[u];
      while (temp != NULL) {
         int v = temp->vertex;
         if (!known[v] && dist[u] != INT_MAX && dist[u] + temp->w < dist[v]) {
            dist[v] = dist[u] + temp->w;
            path[v] = u;
         }
         temp = temp->next;
      }
   }
```

```
    printf("Vertex \t Distance from Source \t Path\n");
    for (int i = 0; i < numvertex; i++) {
        if (i != src && dist[i] != INT_MAX) {
            printf("%d \t\t %d \t\t\t\t\t ", i, dist[i]);
            print_path(path, i);
            printf("\n");
        }
    }
}

int main() {
    int n;
    printf("ENTER THE NO. OF NODES: ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("Number of nodes must be greater than 0\n");
        return -1;
    }
    struct graph *g = create_graph(n);
    input(g);
    print_graph(g);
    int src;
    printf("Enter the source vertex: ");
    scanf("%d", &src);
    if (src < 0 || src >= n) {
        printf("Invalid source vertex\n");
        free_graph(g);
        return -1;
    }
    printf("Shortest Paths using Dijkstra's Algorithm:\n");
    dijkstra(g, src);
    free_graph(g);
    return 0;
}
```

Output:

```
ENTER THE NO. OF NODES: 4
ENTER THE NO. OF EDGES: 6

Enter edge (source destination weight): 0
1
10
Enter edge (source destination weight): 0
2
20
Enter edge (source destination weight): 0
3
30
Enter edge (source destination weight): 1
0
40
Enter edge (source destination weight): 1
2
50
Enter edge (source destination weight): 1
3
60

VERTEX 0
1(40) --> 3(30) --> 2(20) --> 1(10) --> NULL
VERTEX 1
3(60) --> 2(50) --> 0(40) --> 0(10) --> NULL
VERTEX 2
1(50) --> 0(20) --> NULL
VERTEX 3
1(60) --> 0(30) --> NULL

Enter the source vertex: 3

Shortest Paths using Dijkstra's Algorithm:
Vertex          Distance from Source          Path
0               30                            3 -> 0
1               40                            3 -> 0 -> 1
2               50                            3 -> 0 -> 2
```

Result:

Thus, the code is implemented and executed successfully.

Ex. No.: 15	Sorting	Date:01.06.2024

Write a C program to take n numbers and sort the numbers in ascending order. Try to implement the same using following sorting techniques.

1. **Quick Sort**
2. **Merge Sort**

Aim:

To write a C program to take n numbers and sort the numbers in ascending order. To implement the same using following sorting techniques:

1.Quick Sort
2.Merge Sort

Algorithm:

Quick Sort:

Step 1: Start the program

Step 2: Provide void swap

Step 3: Accept integer partition

Step 4: Provide void quicksort

Step 5: Accept int n = sizeof (arr) /sizeof (arr[0])

Step 6: Print original array

Step 7: Print sorted array

Step 8: Stop the program

Merge Sort:

Step 1: Start the program

Step 2: Provide void merge

Step 3: Provide void merge sort

Step 4: Provide void print array

Step 5: Print the given array

Step 6: Print the sorted array

Step 7: Stop the program

Program:

MERGE SORT:

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

void merge(int arr[], int l, int m, int r)
{
   int i, j, k;
   int n1 = m - l + 1;
   int n2 = r - m;

   int L[n1], R[n2];

   for (i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[m + 1 + j];

   i = 0;
   j = 0;
   k = l;

   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 l, int r)
{
    if (l < r) {
        // Same as (l+r)/2, but avoids
        // overflow for large l and r
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

void printArray(int A[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}
```

```
int main()
{
   int n;
   printf("Enter the size of the array: ");
   scanf("%d",&n);
   int arr[n];

   printf("\nEnter array elements:\n");
   for (int i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
   printf("Given array is \n");
   printArray(arr, n);

   mergeSort(arr, 0, n - 1);

   printf("\nSorted array is \n");
   printArray(arr, n);
   return 0;
}
```

Output:

Enter the size of the array: 6

Enter array elements:
12
11
13
5
7
6
Given array is
12 11 13 5 7 6

Sorted array is
5 6 7 11 12 13

Result:

The code is verified and implemented successfully

QUICK SORT:

```
#include <stdio.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[low];
   int i = low;
   int j = high;

   while (i < j) {
      while (arr[i] <= pivot && i <= high - 1) {
         i++;
      }
      while (arr[j] > pivot && j >= low + 1) {
         j--;
      }
      if (i < j) {
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[low], &arr[j]);
   return j;
}
```

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

int main()
{
   int n;
   printf("Enter the size of the array: ");
   scanf("%d",&n);
   int arr[n];

   printf("\nEnter array elements:\n");
   for (int i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   printf("Original array: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }

   quickSort(arr, 0, n - 1);

   // printing the sorted array
   printf("\nSorted array: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   return 0;
}
```

Output :

Enter the size of the array: 9

Enter the elements: 9
19
17
15
12
16
18
4
11
13

Original array:
19 17 15 12 16 18 4 11 13

Sorted array using quick sort:
4 11 12 13 15 16 17 18 19

Result:

The code is verified and implemented successfully

Ex. No.: 16	Hashing	Date:01.06.2024

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

(i) Open addressing

(ii) Closed Addressing

(iii) Rehashing

Aim:

To write a c program to implement various hashing techniques

Algorithm:

1. **Initialize Table:** Create a hash table with a specified size, ensuring it meets a minimum size requirement.
2. **Determine Prime Size:** Calculate the next prime number greater than or equal to the specified table size to reduce collisions.
3. **Allocate Memory:** Allocate memory for the hash table and initialize its cell array.
4. **Hash Function:** Implement a hash function to map keys to indices in the hash table.
5. **Find Position:** Search for a position in the hash table for insertion or retrieval, handling collisions using linear probing.
6. **Insertion:** Insert a key into the hash table, marking the corresponding cell as legitimate if empty.
7. **Display:** Traverse the hash table and print the contents of each cell, displaying both indices and associated data.
8. **Check Fullness:** Determine whether the hash table is full, preventing further insertions if all cells are legitimate.
9. **User Interaction:** Allow users to choose operations such as insertion or display, ensuring appropriate responses to fullness or invalid choices.
10. **Termination:** Gracefully terminate the program upon user request, indicating the end of operations.

Program: (Linear Probing)

```
#include <stdio.h>
#include<stdlib.h>
#define MINSIZE 7
typedef int index;
typedef index position;
typedef struct hash_entry cell;
typedef struct H_table* Hashtable;
enum entry { Legitimate, Empty, Deleted };

struct hash_entry
{
   int data;
   enum entry info;
};

struct H_table
{
   int table_size;
   cell* cell_array;
};
```

```
int next_prime(int table_size)
{
   int i=table_size,f;
   while(1)
   {
      f=1;
      for(int j=2;j<i;j++)
      {
         if(i%j==0)
         {
            f=0;
            break;
         }
      }
      if(f==1)
      return i;
      i++;
   }
}
index Hash(int key,int table_size)
{
   return key % table_size;
}
```

```
Hashtable Create_table(int table_size)
{
   Hashtable H;
   if(table_size < MINSIZE)
   {
      printf("Table size too small");
      return NULL;
   }
   H=(Hashtable)malloc(sizeof(struct H_table));
   if(H==NULL)
   {
      printf("Out of space");
      return NULL;
   }
   H->table_size = next_prime(table_size);
   H->cell_array = (cell*) malloc(sizeof(cell) * H->table_size);
   if(H->cell_array==NULL)
   {
      printf("Out of space");
      return NULL;
   }
   for(int i=0;i< H->table_size;i++)
   {
      H->cell_array[i].info=Empty;
   }
```

```
    return H;
}
position find(int key,Hashtable H)
{
    position cur;
    int collision=0;
    cur=Hash(key,H->table_size);
    while(H->cell_array[cur].info!=Empty && H->cell_array[cur].data != key)
    {
        cur=(cur+1) % H->table_size;
    }
    return cur;
}
void insert(int key,Hashtable H)
{
    position pos=find(key,H);
    if(H->cell_array[pos].info!=Legitimate)
    {
        H->cell_array[pos].info=Legitimate;
        H->cell_array[pos].data=key;
    }
}
```

```
void display(Hashtable H)
{
   for(int i=0;i<H->table_size;i++)
   {
      printf("%d => %d\n",i,H->cell_array[i].data);
   }
}
int isFull(Hashtable H)
{
   int count=0;
   for(int i=0;i<H->table_size;i++)
   {
      if(H->cell_array[i].info==Legitimate)
      count++;
   }
   if(count==H->table_size) return 1;
   else
      return 0;
}
void main()
{
   int size;
   printf("Enter size of the table: ");
   scanf("%d",&size);
   Hashtable H1=Create_table(size);
```

```
printf("Hash table size: %d\n",H1->table_size);
if(H1!=NULL)
{
   int d,op;
   do
   {
      printf("\nChoose operation: 1) Insert 2) Display: ");
      scanf("%d",&op);

      switch(op)
      {
         case 1:
         if(isFull(H1))
         {
            printf("Hash Table is full...\n");
            break;
         }
         printf("Enter the data to be inserted: ");
         scanf("%d",&d);
         insert(d,H1);
         break;
         case 2:
         printf("\nLinear probing hash table\n\n");
         display(H1);
         break;
```

```
            case 0:
            printf("Operation terminated");
            break;
            default:
            printf("Invalid operation...\n");
         }
      }while(op);
   }
}
```

Quadratic Probing:

```
#include <stdio.h>
#include<stdlib.h>
#define MINSIZE 7
typedef int index;
typedef index position;
typedef struct hash_entry cell;
typedef struct H_table* Hashtable;
enum entry { Legitimate, Empty, Deleted };

struct hash_entry
{
   int data;
   enum entry info;
};
```

```
struct H_table
{
   int table_size;
   cell* cell_array;
};
int next_prime(int table_size)
{
   int i=table_size,f;
   while(1)
   {
      f=1;
      for(int j=2;j<i;j++)
      {
         if(i%j==0)
         {
            f=0;
            break;
         }
      }
      if(f==1)
      return i;
      i++;
   }
}
```

```
index Hash(int key,int table_size)
{
   return key % table_size;
}
Hashtable Create_table(int table_size)
{
   Hashtable H;
   if(table_size < MINSIZE)
   {
      printf("Table size too small");
      return NULL;
   }
   H=(Hashtable)malloc(sizeof(struct H_table));
   if(H==NULL)
   {
      printf("Out of space");
      return NULL;
   }
   H->table_size = next_prime(table_size);
   H->cell_array = (cell*) malloc(sizeof(cell) * H->table_size);
   if(H->cell_array==NULL)
   {
      printf("Out of space");
      return NULL;
   }
```

```
    for(int i=0;i< H->table_size;i++)
    {
        H->cell_array[i].info=Empty;
    }
    return H;
}

position find(int key,Hashtable H)
{
    position cur;
    int collision=0;
    cur=Hash(key,H->table_size);
    while(H->cell_array[cur].info!=Empty && H->cell_array[cur].data != key)
    {
        cur=(cur+(2*(++collision)-1)) % H->table_size;
    }
    return cur;
}
void insert(int key,Hashtable H)
{
    position pos=find(key,H);
    if(H->cell_array[pos].info!=Legitimate)
    {
        H->cell_array[pos].info=Legitimate;
        H->cell_array[pos].data=key;
```

```
    }
}

void display(Hashtable H)
{
    for(int i=0;i<H->table_size;i++)
    {
        printf("%d => %d\n",i,H->cell_array[i].data);
    }
}
int isFull(Hashtable H)
{
    int count=0;
    for(int i=0;i<H->table_size;i++)
    {
        if(H->cell_array[i].info==Legitimate)
        count++;
    }
    if(count==H->table_size) return 1;
    else
        return 0;
}
```

```
void main()
{
   int size;
   printf("Enter size of the table: ");
   scanf("%d",&size);
   Hashtable H1=Create_table(size);
   printf("Hash table size: %d\n",H1->table_size);
   if(H1!=NULL)
   {
      int d,op;
      do
      {
         printf("\nChoose operation: 1) Insert 2) Display: ");
         scanf("%d",&op);
         switch(op)
         {
            case 1:
            if(isFull(H1))
            {
               printf("Hash Table is full...\n");
               break;
            }
            printf("Enter the data to be inserted: ");
            scanf("%d",&d);
            insert(d,H1);
```

```
            break;
            case 2:
            printf("\nQuadratic probing hash table\n\n");
            display(H1);
            break;
            case 0:
            printf("Operation terminated");
            break;
            default:
            printf("Invalid operation...\n");
          }
       }while(op);
    }
}
```

Rehashing:

```
#include <stdio.h>
#include<stdlib.h>
#define MINSIZE 7
typedef int index;
typedef index position;
typedef struct hash_entry cell;
typedef struct H_table* Hashtable;
enum entry { Legitimate, Empty, Deleted };
```

```
struct hash_entry
{
   int data;
   enum entry info;
};
struct H_table
{
   int table_size;
   cell* cell_array;
};
int next_prime(int table_size)
{
   int i=table_size,f;
   while(1)
   {
      f=1;
      for(int j=2;j<i;j++)
      {
         if(i%j==0)
         {
            f=0;
            break;
         }
      }
      if(f==1)
      return i;
      i++;
```

```
    }
}
int isFull(Hashtable H)
{
    int count=0;
    for(int i=0;i<H->table_size;i++)
    {
        if(H->cell_array[i].info==Legitimate)
        count++;
    }
    if(count>H->table_size*0.75) return 1;
    else
        return 0;
}
index Hash(int key,int table_size)
{
    return key % table_size;
}
Hashtable Create_table(int table_size)
{
    Hashtable H;
    if(table_size < MINSIZE)
    {
        printf("Table size too small");
        return NULL;
    }
    H=(Hashtable)malloc(sizeof(struct H_table));
```

```
    if(H==NULL)
    {
       printf("Out of space");
       return NULL;
    }
    H->table_size = next_prime(table_size);
    H->cell_array = (cell*) malloc(sizeof(cell) * H->table_size);
    if(H->cell_array==NULL)
    {
       printf("Out of space");
       return NULL;
    }
    for(int i=0;i< H->table_size;i++)
    {
       H->cell_array[i].info=Empty;
    }
    return H; }

position find(int key,Hashtable H)
{
    position cur;
    int collision=0;
    cur=Hash(key,H->table_size);
    while(H->cell_array[cur].info!=Empty && H->cell_array[cur].data != key)
    {
       cur=(cur+1) % H->table_size;
    }
```

```
    return cur;
}
void insert(int key,Hashtable H)
{
    position pos=find(key,H);
    if(H->cell_array[pos].info!=Legitimate)
    {
        H->cell_array[pos].info=Legitimate;
        H->cell_array[pos].data=key;
    }
}
Hashtable Rehash(Hashtable H1)
{
    Hashtable H2=Create_table(H1->table_size*2);
    for(int i=0;i<H1->table_size;i++)
    {
        if(H1->cell_array[i].info==Legitimate)
        insert(H1->cell_array[i].data,H2);
    }
    return H2;
}
void display(Hashtable H)
{
    for(int i=0;i<H->table_size;i++)
    {
        printf("%d => %d\n",i,H->cell_array[i].data);
    }
```

```
}
void main()
{
   int size;
   printf("Enter size of the table: ");
   scanf("%d",&size);
   Hashtable H1=Create_table(size);
   printf("Hash table size: %d\n",H1->table_size);
   if(H1!=NULL)
   {
      int d,op;
      do
      {
         printf("\nChoose operation: 1) Insert 2) Display: ");
         scanf("%d",&op);
         switch(op)
         {
            case 1:
            if(isFull(H1))
            {
               printf("\nRehashing the table from %d to",H1->table_size);
               H1=Rehash(H1);
               printf(" %d...\n\n",H1->table_size);
            }
            printf("Enter the data to be inserted: ");
            scanf("%d",&d);
            insert(d,H1);
```

```
            break;
            case 2:
            printf("\nLinear probing hash table\n\n");
            display(H1);
            break;

            case 0:
            printf("Operation terminated");
            break;
            default:
            printf("Invalid operation...\n");
          }
       }while(op);
    }
}
```

Output:

Enter size of the table: 10

Hash table size: 11

Choose operation: 1) Insert 2) Display: 1

Enter the data to be inserted: 7

Choose operation: 1) Insert 2) Display: 1

Enter the data to be inserted: 18

Choose operation: 1) Insert 2) Display: 1

Enter the data to be inserted: 45

Choose operation: 1) Insert 2) Display: 1

Enter the data to be inserted: 93

Choose operation: 1) Insert 2) Display: 1

Enter the data to be inserted: 17

Choose operation: 1) Insert 2) Display: 1

Enter the data to be inserted: 69

Choose operation: 1) Insert 2) Display: 2

Linear probing hash table

0 => 0

1 => 45

2 => 0

3 => 69

4 => 0

5 => 93

6 => 17

7 => 7

8 => 18

9 => 0

10 => 0

Result:

Thus the program has been successfully executed.

SEPARATE CHAINING:

Program:

```
#include <stdio.h>

#include<stdlib.h>

#define MINSIZE 7


typedef int index;

typedef struct Node* position;

typedef struct Node*List;

typedef struct H_table* Hashtable;


struct Node

{

   int data;

   position next;

};


struct H_table

{

   int table_size;

   List* List_array;

};
```

```
int next_prime(int table_size)
{
   int i=table_size,f;
   while(1)
   {
      f=1;
      for(int j=2;j<i;j++)
      {
         if(i%j==0)
         {
            f=0;
            break;
         }
      }
      if(f==1)
      return i;
      i++;
   }
}

index Hash(int key,int table_size)
{
   return key % table_size;
}
```

```
Hashtable Create_table(int table_size)
{
   Hashtable H;
   if(table_size < MINSIZE)
   {
      printf("Table size too small");
      return NULL;
   }
   H=(Hashtable)malloc(sizeof(struct H_table));
   if(H==NULL)
   {
      printf("Out of space");
      return NULL;
   }
   H->table_size = next_prime(table_size);
   H->List_array = (List*) malloc(sizeof(List) * H->table_size);

   if(H->List_array==NULL)
   {
      printf("Out of space");
      return NULL;
   }
   for(int i=0;i< H->table_size;i++)
   {
      H->List_array[i]=(position)malloc(sizeof(struct Node));
```

```
        if(H->List_array[i]==NULL)
        {
           printf("Out of space");
           return NULL;
        }
        else
        {
           H->List_array[i]->next==NULL;
        }
     }
     return H;
}
position find(int key,Hashtable H)
{
     position p;
     List L;
     L=H->List_array[Hash(key,H->table_size)];
     p=L->next;
     while(p!=NULL && p->data!=key)
        p=p->next;
     return p;}
```

```
void insert(int key,Hashtable H)
{
   position p,new_cell;
   List L;
   p=find(key,H);
   if(p==NULL)
   {
      new_cell=(position)malloc(sizeof(struct Node));
      if(new_cell==NULL)
      printf("Out of space");
      else
      {
         new_cell->data=key;
         L=H->List_array[Hash(key,H->table_size)];
         new_cell->next=L->next;
         L->next=new_cell;
      }
   }
}

void display(Hashtable H){
 for(int i=0;i<H->table_size;i++){
      List L=H->List_array[i];
      printf("%d => ",i);
      position p=L->next;
```

```
        while(p!=NULL)
        {
            printf("%d ",p->data);
            p=p->next;
        }
        printf("\n");
    }
}
int main()
{
    int size;
    printf("Enter size of the table: ");
    scanf("%d",&size);
    Hashtable H1=Create_table(size);
    printf("Hashtable Size: %d\n",H1->table_size);
    if(H1!=NULL)
    {
      int d,op;
       do
       {
           printf("\nChoose operation: 1) Insert 2) Display: ");
           scanf("%d",&op);
```

```
        switch(op)
        {
          case 1:
          printf("Enter the data to be inserted:");
          scanf("%d",&d);
          insert(d,H1);
          break;
           case 2:
          printf("\nSeparate Chaining hash table\n\n");
          display(H1);
          break;

          case 0:
          printf("Operation terminated");
          break;
          default:
          printf("Invalid operation\n");
        }
      } while(op);
   }
}
```

Output:

```
Enter size of the table: 10
Hashtable Size: 11

Choose operation: 1) Insert 2) Display: 1
Enter the data to be inserted: 15

Choose operation: 1) Insert 2) Display: 1
Enter the data to be inserted: 25

Choose operation: 1) Insert 2) Display: 1
Enter the data to be inserted: 35

Choose operation: 1) Insert 2) Display: 2

Separate Chaining hash table

0 =>
1 =>
2 =>
3 =>
4 =>
5 =>
6 =>
7 => 35 25 15
8 =>
9 =>
10 =>

Choose operation: 1) Insert 2) Display: 1
Enter the data to be inserted: 45

Choose operation: 1) Insert 2) Display: 2

Separate Chaining hash table

0 =>
1 =>
2 =>
3 =>
4 =>
5 =>
6 =>
7 => 45 35 25 15
8 =>
9 =>
10 =>

Choose operation: 1) Insert 2) Display: 0
Operation terminated
```

Result:

Thus the Hashing function using separate chaining method was achieved and program has been successfully executed.

Rajalakshmi Institutions

Rajalakshmi Engineering College

Rajalakshmi Nagar Thandalam, Chennai - 602 105.

Phone : +91-44-67181111, 67181112

Website : www.rajalakshmi.org