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 : MALLU KARTHICK BALAJI REDDY

Register No. : 2116231801095

Year / Branch / Section : I-YEAR-ARTIFICAL INTELLIGENCE AND DATA SCIENCE

Semester : I|-SEMESTER

Academic Year : 2023-2024

LESSON PLAN

Course Code	Course Title (Laboratory Integrated Theory Course)	L	T	P	C
CS23231	Data Structures	3	0	4	5

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.:01	Implementation of Single Linked List	Date:29/02/2024

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

Algorithm:

1) Start
2) Define a Node structure with data and a pointer to the next Node.
3) Initialize a head pointer to NULL.
4) Create functions for the following operations: a. Insert at the beginning or end:
- Create a new Node with the given data.
- Traverse the list to the last Node or update the head pointer accordingly. b. Delete from the beginning or end:
- Update the head pointer to the next Node or traverse to the second last Node and update its pointer.

c. Search for a value:
- Traverse the list and compare each Node's data with the given value.
- Return the Node if found, otherwise return NULL.

5) Test the operations by inserting, deleting, and searching for elements in the list. 6) Stop

PROGRAM:

```
#include <stdio.h>
#include<stdlib.h> void
createfnode(int ele); void
insertfront(int ele); void
insertend(int ele); void
display(); //type declaration
of a node struct node { int
data; struct node* next;
};
struct node* head = NULL;
struct node *newnode; void
insertfront(int ele)
{
newnode=(struct node*)malloc(sizeof(struct node));
if(newnode!=NULL) { newnode->data=ele;
if(head!=NULL) { newnode->next=head; head=newnode;
} else {  newnode->next=NULL; head=newnode;


}
}
}
void insertend(int ele)
{
newnode=(struct node*)malloc(sizeof(struct node)); if(newnode!=NULL)
{ newnode->data=ele;
newnode->next=NULL;
if(head!=NULL) {
struct node *t;
t=head; while(t-
>next!=NULL)
{ t=t->next; }
newnode->next=NULL;
t->next=newnode;
}




else {
head=newnode;
}
} }
int listsize()
```

```
{ int c=0;
struct node *t;
t=head;
while(t!=NULL)
{ c=c+1;
t=t->next;
}
printf("\n The size of the list is %d:\n",c);
return c; }
void insertpos(int ele,int pos)
{ int ls=0;
ls=listsize();
if(head == NULL && (pos <= 0 || pos > 1))
{
printf("\nInvalid position to insert a node\n"); return;
}
// if the list is not empty and the position is out of range if(head
!= NULL && (pos <= 0 || pos > ls))
{
printf("\nInvalid position to insert a node\n");
return; }
struct node* newnode = NULL;
newnode=(struct node*)malloc(sizeof(struct node)); if(newnode
!= NULL)
{ newnode->data=ele; struct
node* temp = head; //getting
the position-1 node int count
= 1; while(count < pos-1)
{ temp = temp ->
next; count += 1;
}
//if the position is 1 then insertion at the beginning
if(pos == 1) { newnode->next = head; head = newnode;
} else { newnode->next =
temp->next; temp->next =
newnode;

}
} }
void findnext(int s)
{ struct node
*temp; temp=head;
if(temp==NULL&&temp->next==NULL)
{
printf("No next element ");
} else
{
while(temp->data!=s)
{
temp=temp->next;
}
printf("\nNext Element of %d is %d\n",s,temp->next->data);
} }
void findprev(int s)
{ struct node
*temp; temp=head;
if(temp==NULL)
{
printf("List is empty ");
```

```
} else
{
while(temp->next->data!=s)
{
temp=temp->next;
}
printf("\n The previous ele of %d is %d\n",s,temp->data);
} }
void find(int s)
{ struct node
*temp; temp=head;
if(head==NULL)
{
printf("\n List is empty");
} else
{
while(temp->data!=s && temp->next!=NULL)
{
temp=temp->next;
}
if(temp!=NULL && temp->data==s)
{
printf("\n Searching ele %d is present in the addr of %p",temp-
>data,temp);
} else {
printf("\n Searching elem %d is not present",s);
}
} } void
isempty() {
if(head==NULL)
{
printf("\nList is empty\n");
} else
{
printf("\nList is not empty\n");
}
}
void deleteAtBeginning()
{ struct node
*t; t=head;
head=t->next; }
void deleteAtEnd()
{ struct node
*temp; temp=head;
if(head==NULL) {
printf("\n List is empty");
} else
{
while(temp->next->next!=NULL)
{
temp=temp->next;
}
temp->next=NULL;
}  } void
display() {
struct node *t;
t=head;
while(t!=NULL) {
printf("%d\t",t->data); t=t->next;
} }
```

```
void delete(int ele)
{ struct node *t;
t=head; if(t-
>data==ele)
{ head=t->next;
}
else {
while(t->next->data!=ele)
{ t=t->next;
}
t->next=t->next->next;
} } int main()
{ do { int
ch,a,pos;
printf("\n Choose any one operation that you would like to perform\n");
printf("\n 1.Insert the element at the beginning"); printf("\n 2.Insert
the element at the end"); printf("\n 3. To insert at the specified
position"); printf("\n 4. To view list"); printf("\n 5.To view list
size"); printf("\n 6.To delete first element"); printf("\n 7.To delete
last element"); printf("\n 8.To find next element"); printf("\n 9. To
find previous element"); printf("\n 10. To find search for an element");
printf("\n 11. To quit"); printf("\n Enter your choice\n");
scanf("%d",&ch); switch(ch) { case 1: printf("\n Insert an element to be
inserted at the beginning\n"); scanf("%d",&a); insertfront(a); break;
case 2: printf("\n Insert an element to be inserted at the End\n");
scanf("%d",&a); insertend(a); break; case 3: printf("\n Insert an element
and the position to insert in the list\n"); scanf("%d%d",&a,&pos);
insertpos(a,pos); break; case 4: display(); break; case 5: listsize();
break; case 6: printf("\n Delete an element to be in the beginning\n");
deleteAtBeginning(); break; case 7:
printf("\n Delete an element to be at the end\n");
deleteAtEnd(); break;
case 8: printf("\n enter the element to which you need to find next ele
in the list\n");; scanf("%d",&a); findnext(a); break; case 9:
printf("\n enter the element to which you need to find prev ele in the
list\n"); scanf("%d",&a); findprev(a); break; case 10: printf("\n enter
the element to find the address of it\n"); scanf("%d",&a); find(a);
break; case 11: printf("Ended"); exit(0); default:
printf("Invalid option is chosen so the process is quit");
} }while(1);
return 0;
}
```

OUTPUT:

```
 Choose any one operation that you would like to perform

 1.Insert the element at the beginning
 2.Insert the element at the end
 3. To insert at the specified position
 4. To view list
 5.To view list size
 6.To delete first element
 7.To delete last element
 8.To find next element
 9. To find previous element
 10. To find search for an element
 11. To quit
 Enter your choice
4
2       5       3
 Choose any one operation that you would like to perform

 1.Insert the element at the beginning
 2.Insert the element at the end
 3. To insert at the specified position
 4. To view list
 5.To view list size
 6.To delete first element
 7.To delete last element
 8.To find next element
 9. To find previous element
 10. To find search for an element
 11. To quit
 Enter your choice
5

 The size of the list is 3:

 Choose any one operation that you would like to perform

 1.Insert the element at the beginning
 2.Insert the element at the end
 3. To insert at the specified position
 4. To view list
 5.To view list size
 6.To delete first element
 7.To delete last element
 8.To find next element
 9. To find previous element
 10. To find search for an element
 11. To quit
 Enter your choice
6

 Delete an element to be in the beginning
```

```
 Choose any one operation that you would like to perform

 1.Insert the element at the beginning
 2.Insert the element at the end
 3. To insert at the specified position
 4. To view list
 5.To view list size
 6.To delete first element
 7.To delete last element
 8.To find next element
 9. To find previous element
 10. To find search for an element
 11. To quit
 Enter your choice
6

 Delete an element to be in the beginning

 Choose any one operation that you would like to perform

 1.Insert the element at the beginning
 2.Insert the element at the end
 3. To insert at the specified position
 4. To view list
 5.To view list size
 6.To delete first element
 7.To delete last element
 8.To find next element
 9. To find previous element
 10. To find search for an element
 11. To quit
 Enter your choice
7

 Delete an element to be at the end

 Choose any one operation that you would like to perform

 1.Insert the element at the beginning
 2.Insert the element at the end
 3. To insert at the specified position
 4. To view list
 5.To view list size
 6.To delete first element
 7.To delete last element
 8.To find next element
 9. To find previous element
 10. To find search for an element
 11. To quit
 Enter your choice
8

 enter the element to which you need to find next ele in the list
2
```

```
Choose any one operation that you would like to perform

1.Insert the element at the beginning
2.Insert the element at the end
3. To insert at the specified position
4. To view list
5.To view list size
6.To delete first element
7.To delete last element
8.To find next element
9. To find previous element
10. To find search for an element
11. To quit
Enter your choice
1

Insert an element to be inserted at the beginning
2

Choose any one operation that you would like to perform

1.Insert the element at the beginning
2.Insert the element at the end
3. To insert at the specified position
4. To view list
5.To view list size
6.To delete first element
7.To delete last element
8.To find next element
9. To find previous element
10. To find search for an element
11. To quit
Enter your choice
2

Insert an element to be inserted at the End
3

Choose any one operation that you would like to perform

1.Insert the element at the beginning
2.Insert the element at the end
3. To insert at the specified position
4. To view list
5.To view list size
6.To delete first element
7.To delete last element
8.To find next element
9. To find previous element
10. To find search for an element
11. To quit
Enter your choice
3

Insert an element and the position to insert in the list
5
2

The size of the list is 2:
```

RESULT: Thus the program was successfully executed.

Ex. No.: 02	Implementation of Doubly Linked List	Date:07/03/2024

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

(i) Insertion

(ii) Deletion

(iii) Search

(iv) Display

Algorithm:

1) Start
2) Define a Node structure with data, a pointer to the previous Node, and a pointer to the next Node.
3) Initialize head and tail pointers to NULL. 4) Create functions for the following operations:
a. Insert at the beginning or end:
- Create a new Node with the given data.
- Update the head or tail pointer accordingly.
5) Delete from the beginning or end:
- Update the head or tail pointer to the next or previous Node.
6) Search for a value:
- Traverse the list forward or backward and compare each Node's data with the given value.
- Return the Node if found, otherwise return NULL.
7) Test the operations by inserting, deleting, and searching for elements in the list. 8) Stop

PROGRAM:

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

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

struct node *newnode; struct
node *head = NULL;

void insertfront(int ele)
{
  newnode = (struct node *)malloc(sizeof(struct node));
  if (head == NULL)
  {
     newnode->data = ele;       newnode-
>next = NULL;       newnode->prev =
NULL;
     head = newnode;
  }
  else
  {
     newnode->data = ele;
newnode->next = head;
head->prev = newnode;       head
= newnode;
  }
}

void insertend(int ele)
{
  newnode = (struct node *)malloc(sizeof(struct node));
newnode->data = ele;    newnode->next = NULL;    if
(head != NULL)
  {
     struct node *t;
     t = head;
     while (t->next != NULL)
     {
        t = t->next;
     }
     newnode->prev = t;
     t->next = newnode;
  }
```

```
 else
   {
      newnode->prev = NULL;
      head = newnode;
   }
}

int listsize()
{    int c = 0;
struct node *t;
t = head;
   while (t != NULL)
   {       c++;
t = t->next;
   }
   printf("\n The size of the list is %d:\n", c);
   return c;
}

void insertpos(int ele, int pos)
{    int ls =
0;
   struct node *temp;
ls = listsize();    if
(head == NULL)
   {
      printf("\nInvalid position to insert a node\n");
return;
   }
   if (head != NULL && (pos <= 0 || pos > ls))
   {
      printf("\nInvalid position to insert a node\n");
return;
   }

   newnode = (struct node *)malloc(sizeof(struct node));

   if (newnode != NULL)
   {
      newnode->data = ele;
temp = head;       int
count = 1;
      while (count < pos - 1)
      {
         temp = temp->next;
count++;
      }
      if (pos == 1)
      {
         newnode->next = head;
head->prev = newnode;          head
= newnode;
      }
```

```
 else
      {
        newnode->next = temp->next;
if (temp->next != NULL)              temp-
>next->prev = newnode;          temp-
>next = newnode;          newnode-
>prev = temp;
      }
   }
}

void find(int s)
{    int c =
1;
   struct node *temp;
temp = head;
   if (head == NULL)
   {
      printf("\n List is empty");
   }
   else
   {
      while (temp->data != s && temp->next != NULL)
      {
        temp = temp->next;
        c++;
      }
      if (temp != NULL && temp->data == s)
      {
        printf("\n Searching ele %d is present in the addr of %p in the pos%d", temp->data,
temp, c);
      }
else
      {
        printf("\n Searching elem %d is not present", s);
      }
   }
}

void findnext(int s)
{
   struct node *temp;
temp = head;
   if (temp == NULL || temp->next == NULL)
   {
      printf("No next element ");
   }
   else
   {
      while (temp->data != s)
      {
        temp = temp->next;
      }
      printf("\nNext Element of %d is %d\n", s, temp->next->data);
```

```
 }
}

void findprev(int s)
{
  struct node *temp;
temp = head;
  if (temp == NULL)
  {
    printf("List is empty ");
  }
  else
  {
    while (temp->data != s)
    {
      temp = temp->next;
    }
    printf("\n The previous ele of %d is %d\n", s, temp->prev->data);
  }
}

void deleteAtBeginning()
{
  if (head == NULL)
  {
    printf("List is empty");
  }
  else
  {
    struct node *t = head;
head = head->next;       if
(head != NULL)
head->prev = NULL;
free(t);
  }
}

void deleteAtEnd()
{
  if (head == NULL)
  {
    printf("\n List is empty");
  }
  else
  {
    struct node *temp = head;
    while (temp->next != NULL)
    {
      temp = temp->next;
    }
    if (temp->prev != NULL)          temp->prev-
>next = NULL;
    free(temp);
```

```
}
}

void delete(int ele)
{
  struct node *t = head;
if (t->data == ele)
  {
    head = t->next;       if
(head != NULL)
head->prev = NULL;
free(t);
  }
  else
  {
    while (t->data != ele)
    {
      t = t->next;
    }
    if (t->next != NULL)          t->next-
>prev = t->prev;       t->prev->next =
t->next;
    free(t);
  }
}

void display()
{
  struct node *temp;
temp = head;    while
(temp != NULL)
  {
    printf("%d-->", temp->data);
    temp = temp->next;
  }
}

int main()
{
  insertfront(10);
insertfront(20);
insertfront(30);    display();
  printf("\n Inserted ele 40 at the end\n");
insertend(40);    display();    insertpos(25,
3);    display();    find(25);    findnext(25);
findprev(25);
  printf("\n element deleted in the beginning\n");
deleteAtBeginning();    display();

 deleteAtEnd();
  printf("\n Element deleted at the end\n");
display();
  printf("\n After deleting element 25\n");
delete(25);    display();    return 0;
```

}

OUTPUT:

```
30-->20-->10-->
 Inserted ele 40 at the end
30-->20-->10-->40-->
 The size of the list is 4:
30-->20-->25-->10-->40-->
 Searching ele 25 is present in the addr of 0x55ff3be7d730 in the pos3
Next Element of 25 is 10

 The previous ele of 25 is 20

 element deleted in the beginning
20-->25-->10-->40-->
 Element deleted at the end
20-->25-->10-->
 After deleting element 25
20-->10-->aiml2315011167@cselab:~$
```

RESULT: Thus, the program was successfully executed.

Ex. No.:03	Polynomial Manipulation	Date:21/03/2024

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 a structure for a polynomial term with coefficients and exponents.
3) Create functions for the following operations: a. Polynomial addition:
- Initialize a result polynomial.
- Traverse both input polynomials simultaneously.
- Add coefficients of terms with the same exponent and append to the result.
- Append any remaining terms from both polynomials to the result.
4) Polynomial subtraction:
- Initialize a result polynomial.
- Traverse both input polynomials simultaneously.
- Subtract coefficients of terms with the same exponent and append to the result.
- Append any remaining terms from both polynomials to the result.
5) Polynomial multiplication:
- Initialize a result polynomial.
- Multiply each term of the first polynomial with each term of the second polynomial.
- Add the coefficients of terms with the same exponent and append to the result. 6) Test the operations by performing addition, subtraction, and multiplication on sample polynomials.
7) Stop

PROGRAM:

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

{ int coeff; int pow;
struct node *Next;

};
 struct node *Poly1,*Poly2,*Result; void Create(struct node *List); void
Display(struct node *List); void Addition(struct node *Poly1,struct node
*Poly2,struct node *Result); int main()

{
Poly1=(struct node*)malloc(sizeof(struct node));
Poly2=(struct node*)malloc(sizeof(struct node));
Result=(struct node*)malloc(sizeof(struct node));

Poly1->Next = NULL;
Poly2->Next = NULL;
printf("Enter the values for first polynomial :\n");
Create(Poly1); printf("The polynomial equation is : ");
Display(Poly1); printf("\nEnter the values for second
polynomial :\n"); Create(Poly2); printf("The
polynomial equation is : "); Display(Poly2);
Addition(Poly1, Poly2, Result); printf("\nThe
polynomial equation addition result is : ");
Display(Result); return 0;

}
void Create(struct node *List)
{
```

```
int choice; struct node *Position,
*NewNode; Position = List; do

{
NewNode = malloc(sizeof(struct node));
printf("Enter the coefficient : ");
scanf("%d", &NewNode->coeff);
printf("Enter the power : "); scanf("%d",
&NewNode->pow); NewNode->Next =
NULL;

Position->Next = NewNode; Position
= NewNode;

printf("Enter 1 to continue : "); scanf("%d",
&choice);

} while(choice == 1);
}
void Display(struct node *List)
{
struct node *Position; Position
= List->Next; while(Position !=
NULL)

{
printf("%dx^%d", Position->coeff, Position->pow);
Position = Position->Next; if(Position != NULL &&
Position->coeff > 0)

{ printf("+");

}
```

```
}
}
void Addition(struct node *Poly1, struct node *Poly2, struct node *Result)
{
struct node *Position; struct
node *NewNode; Poly1 =
Poly1->Next;

Poly2 = Poly2->Next;
Result->Next = NULL; Position = Result;
while(Poly1 != NULL && Poly2 != NULL)

{
NewNode = malloc(sizeof(struct node)); if(Poly1->pow
== Poly2->pow)
{
NewNode->coeff = Poly1->coeff + Poly2->coeff;
NewNode->pow = Poly1->pow;
Poly1 = Poly1->Next;
Poly2 = Poly2->Next;
}
else if(Poly1->pow > Poly2->pow)
{
NewNode->coeff = Poly1->coeff;
NewNode->pow = Poly1->pow;
Poly1 = Poly1->Next;
}
else if(Poly1->pow < Poly2->pow)
{
NewNode->coeff = Poly2->coeff;
```

```
NewNode->pow = Poly2->pow;
Poly2 = Poly2->Next;
}
NewNode->Next = NULL;
Position->Next = NewNode;
Position = NewNode;
}
while(Poly1 != NULL || Poly2 != NULL)
{
NewNode = malloc(sizeof(struct node)); if(Poly1
!= NULL)
{
NewNode->coeff = Poly1->coeff;
NewNode->pow = Poly1->pow;
Poly1 = Poly1->Next;
}
if(Poly2 != NULL)
{
NewNode->coeff = Poly2->coeff;
NewNode->pow = Poly2->pow;
Poly2 = Poly2->Next;
}
NewNode->Next = NULL;
Position->Next = NewNode;
Position = NewNode;
}
}
```

Program 2:

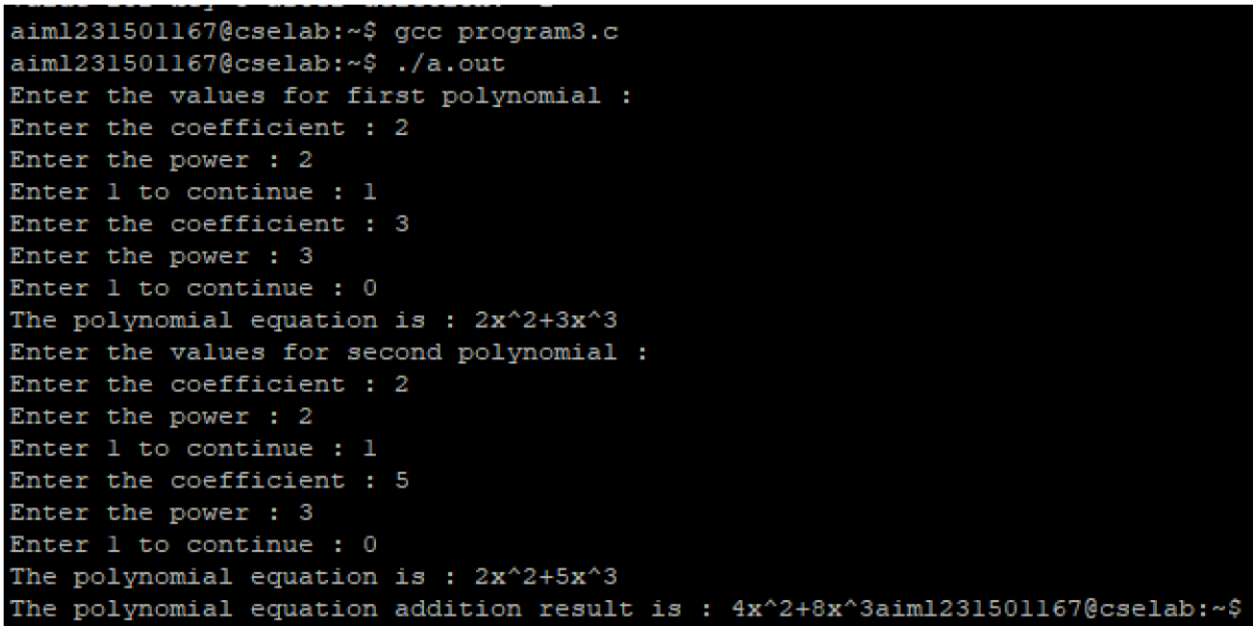
```
#include<stdio.h> #include<stdlib.h>
struct node
```

```
{    int
coeff;    int
expo;
   struct node *next;
};

struct node* insert(struct node *head,int co,int exp)
{
   struct node *temp;
   struct node *newnode=malloc(sizeof(struct node));
newnode->coeff=co;    newnode->expo=exp;
   newnode->next=NULL;



if(head==NULL || exp>head->expo)
   {
      newnode->next=head;
head=newnode;
   }
else
   {
      temp=head;
      while(temp->next!=NULL &&temp->next->expo>=exp)
         temp=temp->next;        newnode->next=temp-
>next;
      temp->next=newnode;
   }
   return head;
}
struct node* create(struct node *head)
{    int n,i;
int coeff;
int expo;
   printf("Enter the no of terms:");
scanf("%d",&n);
   for(i=0;i<n;i++)
   {
      printf("Enter the coeefficient for term %d:",i+1);
scanf("%d",&coeff);

      printf("Enter the exponent for term %d:",i+1);
scanf("%d",&expo);

      head=insert(head,coeff,expo);
   }
   return head;
}
   void print(struct node* head)
   {
      if(head==NULL)
printf("No Polynomial");
      else
      {
```

```
        struct node *temp=head;
        while(temp!=NULL)
        {
            printf("%dx^%d",temp->coeff,temp->expo);
            temp=temp->next;
if(temp!=NULL)
printf("+");


         else
              printf("\n");
        }
     }
  }

  void polyAdd(struct node *head1, struct node *head2)
  {
     struct node *ptr1=head1;
struct node *ptr2=head2;        struct
node *head3=NULL;
     while(ptr1!=NULL && ptr2!=NULL)
     {
        if(ptr1->expo == ptr2->expo)
        {
        head3=insert(head3,ptr1->coeff+ptr2->coeff,ptr1->expo);
ptr1=ptr1->next;                ptr2=ptr2->next;
        }
        else if(ptr1->expo > ptr2->expo)
        {
            head3=insert(head3,ptr1->coeff,ptr1->expo);
            ptr1=ptr1->next;
        }
        else if(ptr1->expo < ptr2->expo)
        {
            head3=insert(head3,ptr2->coeff,ptr2->expo);
            ptr2=ptr2->next;
        }
     }
     while(ptr1!=NULL)
     {
        head3=insert(head3,ptr1->coeff,ptr1->expo);
        ptr1=ptr1->next;
     }
     while(ptr2!=NULL)
     {
        head3=insert(head3,ptr2->coeff,ptr2->expo);
        ptr2=ptr2->next;
     }
    printf("Added Polynomial is: ") ;
print(head3);
  }
  int main()
  {
```

```
        struct node *head1=NULL;
struct node *head2=NULL;
        printf("Enter first polynomial\n");

        head1=create(head1);
        printf("Enter second polynomial\n");
head2=create(head2);
polyAdd(head1,head2);         return 0;

   }
```

OUTPUT:



```
aiml231501167@cselab:~$ gcc program3.c
aiml231501167@cselab:~$ ./a.out
Enter the values for first polynomial :
Enter the coefficient : 2
Enter the power : 2
Enter 1 to continue : 1
Enter the coefficient : 3
Enter the power : 3
Enter 1 to continue : 0
The polynomial equation is : 2x^2+3x^3
Enter the values for second polynomial :
Enter the coefficient : 2
Enter the power : 2
Enter 1 to continue : 1
Enter the coefficient : 5
Enter the power : 3
Enter 1 to continue : 0
The polynomial equation is : 2x^2+5x^3
The polynomial equation addition result is : 4x^2+8x^3aiml231501167@cselab:~$
```

RESULT: Thus the program was successfully executed.

Ex. No.:04	Implementation of Stack using Array and Linked List Implementation	Date:21/03/2024

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

Algorithm:

1) Start
2) Stack using Array:
3) Define a fixed-size array to store the stack elements and initialize a variable top to -1. Implement functions for the following operations:
- Push: Increment top and insert the element at the top index.
- Pop: Remove the element at the top index and decrement top.
- Peek: Return the element at the top index without removing it.
- IsEmpty: Check if top is -1 to determine if the stack is empty.
- IsFull: Check if top is equal to the maximum array size to determine if the stack is full.

4) Stack using Linked List:
a. Define a Node structure with data and a pointer to the next Node.
b. Initialize a head pointer to NULL.
c. Implement functions for the following operations:
- Push: Create a new Node with the given data and insert it at the beginning of the list.
- Pop: Remove the first Node from the list.
- Peek: Return the data of the first Node without removing it.
- IsEmpty: Check if the head pointer is NULL to determine if the stack is empty. 5) Stop

PROGRAM: ARRAY IMPLEMENTATION

```
#include <stdio.h>
#include <string.h>
#include<ctype.h>
int top = -1; int
stack[100]; void push
(int data) {
stack[++top] = data;

} int pop () { int
data; if (top == -
1) return -1; data
= stack[top];
stack[top] = 0;
top--; return
(data);

}
int main()
{
char str[100];
int i, data = -1, operand1, operand2, result;
printf("Enter ur postfix expression:");
fgets(str, 100, stdin); for (i = 0; i <
strlen(str); i++)

 { if
(isdigit(str[i]))

{
data = (data == -1) ? 0 : data; data
= (data * 10) + (str[i] - 48);
continue;
```

```
} if (data != -
1)
{
push(data);
}
if (str[i] == '+' || str[i] == '-'|| str[i] == '*' || str[i] == '/')
{
operand2 = pop(); operand1 = pop();
if (operand1 == -1 || operand2 == -1)
break; switch (str[i])
{
case '+':
result = operand1 + operand2;
push(result); break; case '-':
result = operand1 - operand2;
push(result); break; case '*':
result = operand1 * operand2;
push(result); break; case '/':
result = operand1 / operand2;
push(result); break;
}
}

data = -1;
} if (top ==
0)
printf("The answer is:%d\n", stack[top]); else
printf("u have given wrong postfix expression\n");
return 0;
}
```

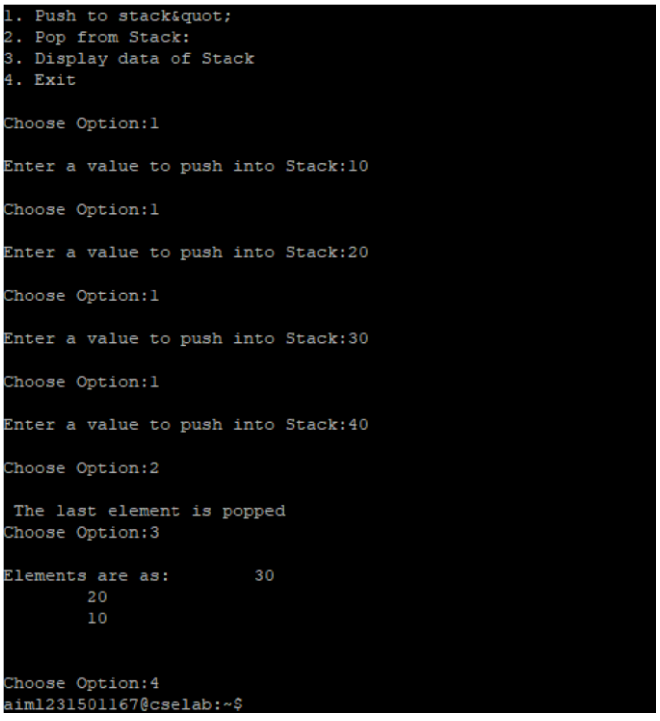
LINKED LIST IMPLEMENTATION:

```
#include <stdio.h>
#include <stdlib.h> struct
Node
{
int Data; struct
Node *next; }*top;
void popStack()
{
struct Node *temp, *var=top; if(var==top)
{
top = top->next; free(var);
}
else printf("\nStack
Empty");
}
void push(int value)
{
struct Node *temp;
temp=(struct Node *)malloc(sizeof(struct Node)); temp->Data=value; if (top == NULL)
{
top=temp; top->next=NULL;
}

else
{
temp->next=top; top=temp;
}
}
void display()
{
struct Node *var=top; if(var!=NULL)
{
```

```
printf("\nElements are as:"); while(var!=NULL)
{
printf("\t%d\n",var->Data); var=var->next;
} printf("\n");
}
else printf("\nStack is
Empty&quot;");
}
int main()
{ int
i=0;
top=NULL;

printf("\n1. Push to stack&quot;");
printf("\n2. Pop from Stack:");
printf("\n3. Display data of Stack");
printf("\n4. Exit\n"); while(1)
{
printf ("\nChoose Option:");
scanf("%d",&i); switch(i)
{
case 1:
{
int value;
printf("\nEnter a value to push into Stack:");
scanf("%d",&value); push(value); break;
}
case 2:
 {
popStack(); printf("\n The last element
is popped"); break;
```

```
}
case 3:
{
display(); break;

}
case 4:
{


struct Node *temp; while(top!=NULL)

{
temp = top->next;
free(top); top=temp;

} exit(0);

}
default:
{
printf("\nwrong choice for operation");
}}}}
```

OUTPUT:



RESULT: Thus the program was successfully executed.

Ex. No.: 05	Infix to Postfix Conversion	Date:28/03/2024

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

Algorithm:

1) Start
2) Initialize an empty stack to hold operators.
3) Initialize an empty string to store the postfix expression.
4) Iterate through each character in the infix expression:
a. If the character is an operand, add it to the postfix string.
b. If the character is an operator:
i. While the stack is not empty and the precedence of the top operator in the stack is greater than or equal to the current operator, pop operators from the stack and add them to the postfix string. ii. Push the current operator onto the stack.
c. If the character is an opening parenthesis '(', push it onto the stack.
d. If the character is a closing parenthesis ')':
i. Pop operators from the stack and add them to the postfix string until an opening parenthesis is encountered. ii. Discard the opening parenthesis.
5) Pop any remaining operators from the stack and add them to the postfix string. 6) Return the postfix expression.
7) Stop

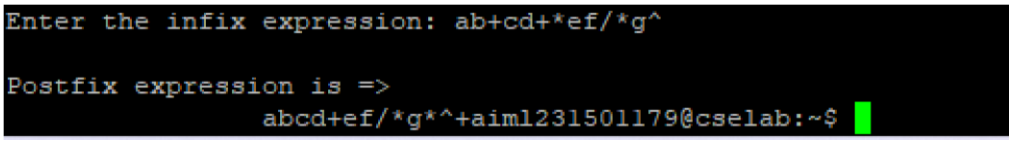
PROGRAM:

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

int top = 0; int stack[20];
char infix[40], postfix[40];


void convertToPostfix();
void push(int); char
pop();


int main() {     printf("Enter the infix
expression: ");     scanf("%s", infix);
convertToPostfix();     return 0;

}


void convertToPostfix() {     int
i, j = 0;     for (i = 0; infix[i] !=
'\0'; i++) {         switch (infix[i]) {
case '+':

               while (stack[top] >= 1)
postfix[j++] = pop();
push(1);                 break;
case '-':                while
```

```
(stack[top] >= 1)
postfix[j++] = pop();

            push(2);
break;          case
'*':

            while (stack[top] >= 3)
postfix[j++] = pop();
push(3);              break;
case '/':

            while (stack[top] >= 4)
postfix[j++] = pop();
push(4);              break;
case '^':             postfix[j++] =
pop();              push(5);
break;          case '(':
push(0);              break;
case ')':

            while (stack[top] != 0)
postfix[j++] = pop();
top--;              break;
default:

            postfix[j++] = infix[i];
       }
   }
   while (top > 0)


         postfix[j++] = pop();
   printf("\nPostfix expression is =>\n\t\t%s", postfix);
}
```

```
void push(int element) {
top++;     stack[top] =
element;

}


char pop() {
   int el;     char
e;     el =
stack[top];     top-
-;     switch (el) {
case 1:            e
= '+';
break;        case
2:            e = '-';
break;        case
3:            e = '*';
break;        case
4:            e = '/';
break;        case
5:            e = '^';
break;

   }

return e;
}
```

OUTPUT:



RESULT: Thus, the program was successfully executed.

Ex. No.:06	Evaluating Arithmetic Expression	Date:04/04/2024

Write a C program to evaluate Arithmetic expressions using stack.

Algorithm:

1) Start
2) Create an empty stack to store operands. 3) Iterate through each character in the expression: a. If the character is a digit, push it onto the stack.

b. If the character is an operator (+, -, *, /), pop two operands from the stack, perform the operation, and push the result back onto the stack.
4) After processing all characters, the final result will be the only element left in the stack.
5) Return this result as the evaluation of the arithmetic expression. 6) Stop

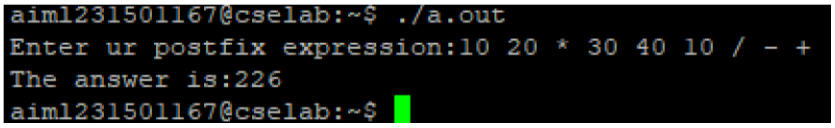
PROGRAM:

```
#include <stdio.h>

#include <string.h>

#include<ctype.h>

int top = -1; int

stack[100]; void push
```

```
(int data) {
stack[++top] = data;

} int pop () { int
data; if (top == -
1) return -1; data
= stack[top];
stack[top] = 0;
top--; return
(data);

}
int main()
{
char str[100];
int i, data = -1, operand1, operand2, result;
printf("Enter ur postfix expression:");
fgets(str, 100, stdin); for (i = 0; i <
strlen(str); i++)

 { if
(isdigit(str[i]))

{
data = (data == -1) ? 0 : data; data
= (data * 10) + (str[i] - 48);
continue;

} if (data != -
1)

{
push(data);
}
if (str[i] == '+' || str[i] == '-'|| str[i] == '*' || str[i] == '/')
{
```

```
operand2 = pop(); operand1 = pop();
if (operand1 == -1 || operand2 == -1)
break; switch (str[i])
{
case '+':
result = operand1 + operand2;
push(result); break; case '-':
result = operand1 - operand2;
push(result); break; case '*':
result = operand1 * operand2;
push(result); break; case '/':
result = operand1 / operand2;
push(result); break;
}
}
data = -1;
} if (top ==
0)
printf("The answer is:%d\n", stack[top]); else
printf("u have given wrong postfix expression\n");
return 0;
}
```

OUTPUT:



RESULT: Thus, the program was successfully executed.

Ex. No.:07	Implementation of Queue using Array and Linked List Implementation	Date:11/04/2024

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

Algorithm:

1) Start
2) 1. For Queue using Array:
3) Initialize an array of fixed size to store the queue elements.
4) Maintain two pointers, front and rear, to track the front and rear of the queue.
5) Implement the following operations:
- Enqueue: Add an element to the rear of the queue by incrementing the rear pointer. - Dequeue: Remove an element from the front of the queue by incrementing the front pointer.
- isEmpty: Check if the queue is empty by comparing the front and rear pointers.

- isFull: Check if the queue is full by comparing the rear pointer with the array size.

6) For Queue using Linked List:
7) Define a Node structure with data and a pointer to the next Node.
8) Maintain pointers to the front and rear Nodes of the queue.
9) Implement the following operations:
- Enqueue: Create a new Node with the given data and update the rear pointer.
- Dequeue: Remove the front Node and update the front pointer.
- isEmpty: Check if the queue is empty by comparing the front pointer with NULL. - Display: Traverse the linked list to display all elements in the queue. 10) Stop

PROGRAM:

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

struct queue
{    int data;    struct
queue *next;

};

struct queue *addq(struct queue *front); struct
queue *delq(struct queue *front);
```

```
int main()
{
   struct queue *front = NULL;
int option;


   do
   {
      printf("\n1. Add to Queue");
printf("\n2. Delete from Queue");
printf("\n3. Exit");        printf("\nSelect
an option: ");        scanf("%d",
&option);


      switch(option)
      {
case 1:

            front = addq(front);
printf("\nElement added to the queue.");
break;          case 2:              front =
delq(front);              break;          case 3:
exit(0);

      }
   } while(1);


   return 0;
}


struct queue *addq(struct queue *front)
{
   struct queue *new_node = (struct queue*)malloc(sizeof(struct queue));
if(new_node == NULL)

   {
```

```
        printf("Insufficient memory.");
return front;
    }

    printf("\nEnter data: ");
scanf("%d", &new_node->data);
new_node->next = NULL;

    if(front == NULL)
    {
        front = new_node;
    }
else
    {


            struct queue *temp = front;         while(temp-
>next != NULL)
        {
            temp = temp->next;
        }
        temp->next = new_node;
    }

    return front;
}

struct queue *delq(struct queue *front)
{
    if(front == NULL)
    {
```

```
        printf("Queue is empty.");
return front;

    }

    struct queue *temp = front;
printf("Deleted data: %d", front->data);
front = front->next;     free(temp);

    return front; }
```

OUTPUT:

```
1. Add to Queue
2. Delete from Queue
3. Exit
Select an option: 1

Enter data: 2

Element added to the queue.
1. Add to Queue
2. Delete from Queue
3. Exit
Select an option: 1

Enter data: 3

Element added to the queue.
1. Add to Queue
2. Delete from Queue
3. Exit
Select an option: 1

Enter data: 4

Element added to the queue.
1. Add to Queue
2. Delete from Queue
3. Exit
Select an option: 2
Deleted data: 2
1. Add to Queue
2. Delete from Queue
3. Exit
Select an option: 2
Deleted data: 3
1. Add to Queue
2. Delete from Queue
3. Exit
Select an option: 4

1. Add to Queue
2. Delete from Queue
3. Exit
Select an option: 3
aiml231501179@cselab:~$
```

RESULT: Thus, the program was successfully executed.

Ex. No.:08	Tree Traversal	Date:18/04/2024

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

Algorithm:

1) Start
2) Define a Node structure with data, left child pointer, and right child pointer. 3) Create functions for the following traversal methods:
4) Inorder traversal:
- Recursively call the function on the left child.
- Print the data of the current Node.
- Recursively call the function on the right child.

5) Preorder traversal:
- Print the data of the current Node.
- Recursively call the function on the left child.
- Recursively call the function on the right child.

6) Postorder traversal:
- Recursively call the function on the left child.
- Recursively call the function on the right child.
- Print the data of the current Node.

7) Initialize the root of the binary tree.
8) Call the traversal functions with the root Node to perform inorder, preorder, and postorder traversal. 9) Stop

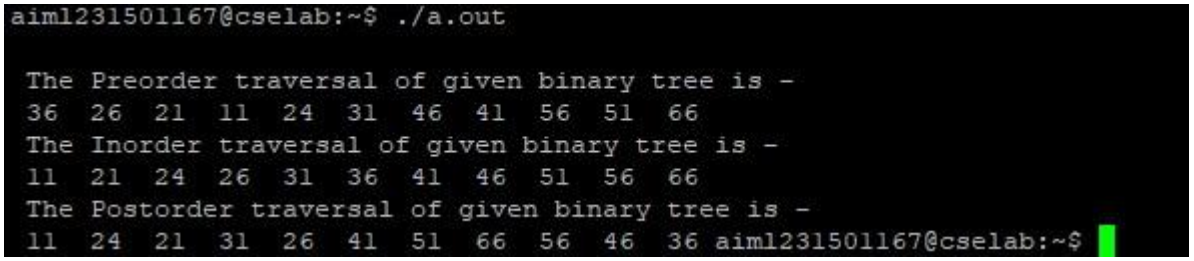
PROGRAM;

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

struct node { int
element; struct
node* left; struct
node* right;
};

struct node* createNode(int val)
{
struct node* Node = (struct node*)malloc(sizeof(struct node));
Node->element = val;
Node->left = NULL;
Node->right = NULL;

return (Node);
}

void traversePreorder(struct node* root)
{
if (root == NULL) return;
printf(" %d ", root->element); traversePreorder(root->left);
traversePreorder(root->right);
}



void traverseInorder(struct node* root)
{
```

```
if (root == NULL) return;
traverseInorder(root->left);
printf(" %d ", root->element);
traverseInorder(root->right);

}
void traversePostorder(struct node* root)
{
if (root == NULL) return;

traversePostorder(root->left);
traversePostorder(root->right); printf("
%d ", root->element);

}

int main()
{
struct node* root = createNode(36); root-
>left = createNode(26); root->right =
createNode(46); root->left->left =
createNode(21); root->left->right =
createNode(31); root->left->left->left =
createNode(11); root->left->left->right =
createNode(24); root->right->left =
createNode(41); root->right->right =
createNode(56); root->right->right->left =
createNode(51); root->right->right->right =
createNode(66);

printf("\n The Preorder traversal of given binary tree is -\n"); traversePreorder(root);

printf("\n The Inorder traversal of given binary tree is -\n"); traverseInorder(root);

printf("\n The Postorder traversal of given binary tree is -\n"); traversePostorder(root);
```

```
return 0;
}
```

OUTPUT:



RESULT: Thus, the program was successfully executed.

Ex. No.:09	Implementation of Binary Search tree	Date:25/04/2024

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

(i) Insert

(ii) Delete

(iii) Search

(iv) Display

Algorithm:

1) Start
2) Define a Node structure with data, left child pointer, and right child pointer.
3) Initialize a root pointer to NULL.
4) Create functions for the following operations: a. Insert:
- If the tree is empty, create a new Node and set it as the root.
- Otherwise, traverse the tree starting from the root:
- If the data is less than the current Node's data, move to the left child.
- If the data is greater than the current Node's data, move to the right child. - Repeat until reaching a NULL child pointer, then insert the new Node. b. Search:
- Start from the root and compare the data with each Node:
- If the data matches, return the Node.
- If the data is less than the current Node's data, move to the left child.
- If the data is greater than the current Node's data, move to the right child.
- Repeat until finding the data or reaching a NULL child pointer.
5) Test the operations by inserting elements into the tree and searching for specific values. 6) Stop

PROGRAM;

```
#include <stdio.h>

#include <stdlib.h> struct
```

```
BinaryTreeNode { int
key;

struct BinaryTreeNode *left, *right;
};


struct BinaryTreeNode* newNodeCreate(int value)
{
struct BinaryTreeNode* temp =
(struct BinaryTreeNode*)malloc(
sizeof(struct BinaryTreeNode));
temp->key = value; temp->left =
temp->right = NULL; return temp;

}
struct BinaryTreeNode*
searchNode(struct BinaryTreeNode* root, int target)
{
if (root == NULL || root->key == target) { return
root;

}
if (root->key < target) { return
searchNode(root->right, target);

}
return searchNode(root->left, target);
}


struct BinaryTreeNode*
insertNode(struct BinaryTreeNode* node, int value)
{
if (node == NULL) { return
newNodeCreate(value);

}
```

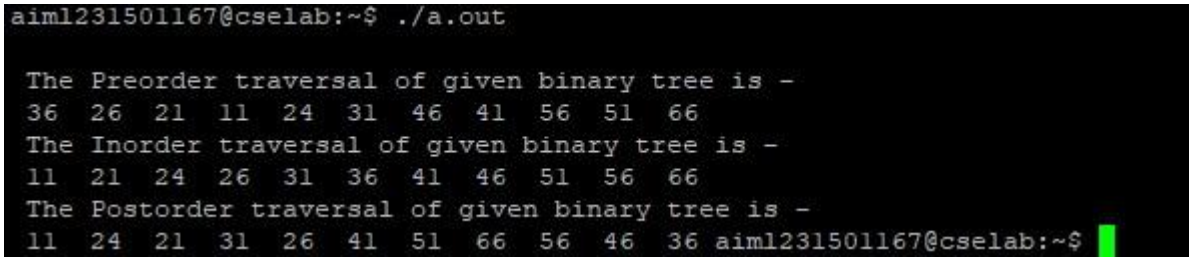
```
if (value < node->key) { node->left =
insertNode(node->left, value);

}
else if (value > node->key) { node->right =
insertNode(node->right, value);

}
return node;
}

void postOrder(struct BinaryTreeNode* root)
{
if (root != NULL) {
postOrder(root->left);
postOrder(root->right); printf("
%d ", root->key);

}
}

void inOrder(struct BinaryTreeNode* root)
{
if (root != NULL) {
inOrder(root->left); printf("
%d ", root->key);
inOrder(root->right);

}
}
void preOrder(struct BinaryTreeNode* root)
{
if (root != NULL) { printf("
%d ", root->key);
preOrder(root->left);
preOrder(root->right);

}
```

```
}

struct BinaryTreeNode* findMin(struct BinaryTreeNode* root)
{
if (root == NULL) { return
NULL;
}
else if (root->left != NULL) { return
findMin(root->left);
}
return root;
}

struct BinaryTreeNode* delete (struct BinaryTreeNode* root, int
x)
{
if (root == NULL) return
NULL;

if (x > root->key) { root->right =
delete (root->right, x);
}
else if (x < root->key) { root->left
= delete (root->left, x);
}
else {
if (root->left == NULL && root->right == NULL) {
free(root); return NULL;
}
else if (root->left == NULL ||
root->right == NULL) { struct
BinaryTreeNode* temp; if
```

```
(root->left == NULL) { temp =
root->right;

} else { temp =
root->left;

} free(root);
return temp;

} else
{

struct BinaryTreeNode* temp = findMin(root->right);
root->key = temp->key; root->right = delete (root-
>right, temp->key);

}
}
return root;
}


int main()
{
struct BinaryTreeNode* root = NULL;


root = insertNode(root, 50);
insertNode(root, 30);
insertNode(root, 20);
insertNode(root, 40);
insertNode(root, 70);
insertNode(root, 60);
insertNode(root, 80); if
(searchNode(root, 60) != NULL) {
printf("60 found");

} else { printf("60 not
found");

}
```

```
printf("\n"); postOrder(root);
printf("\n");


preOrder(root);
printf("\n"); inOrder(root);
printf("\n");


struct BinaryTreeNode* temp = delete (root, 70);
printf("After Delete: \n"); inOrder(root);


return 0;
}
```

OUTPUT:



RESULT: Thus, the program was successfully executed.

Ex. No.: 10	Implementation of AVL Tree	Date:02/05/2024

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.

Algorithm:

1) Start
2) Define the AVL Node Structure.
3) Implement Rotation Operations (left and right rotations).
4) Insert new nodes into the AVL tree, updating heights and balancing as needed.
5) Delete nodes from the AVL tree, updating heights and balancing as needed.
6) Implement traversal functions (in-order, pre-order, post-order) to navigate through the tree.
7) Implement a search function to find specific elements within the AVL tree.
8) Test the AVL tree implementation with various scenarios.
9) Optionally, optimize the implementation for better performance. 10) Stop

PROGRAM:

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

struct node
{  int data;    struct
node* left;    struct
node* right;
 int ht;
};

struct node* root = NULL;
```

```
struct node* create(int);   struct node*
insert(struct node*, int);   struct node*
delete(struct node*, int);   struct node*
search(struct node*, int);   struct node*
rotate_left(struct node*);   struct node*
rotate_right(struct node*);   int
balance_factor(struct node*);   int
height(struct node*);   void
inorder(struct node*);   void
preorder(struct node*);   void
postorder(struct node*);


int main()
{
   int user_choice, data;

   char user_continue = 'y';
struct node* result = NULL;


   while (user_continue == 'y' || user_continue == 'Y')
   {
      printf("\n\n------- AVL TREE --------\n");
printf("\n1. Insert");          printf("\n2.
Delete");          printf("\n3. Search");
printf("\n4. Inorder");          printf("\n5.
Preorder");          printf("\n6. Postorder");
printf("\n7. EXIT");


      printf("\n\nEnter Your Choice: ");
scanf("%d", &user_choice);
```

```
    switch(user_choice)
    {
case 1:
        printf("\nEnter data: ");
scanf("%d", &data);             root
= insert(root, data);
break;

      case 2:
        printf("\nEnter data: ");
scanf("%d", &data);             root
= delete(root, data);
break;

        case 3:
        printf("\nEnter data: ");
scanf("%d", &data);             result
= search(root, data);            if
(result == NULL)
        {
            printf("\nNode not found!");
        }
else
        {
          printf("\n Node found");
        }
break;          case 4:
inorder(root);
break;
```

```
        case 5:
preorder(root);
break;


        case 6:
           postorder(root);
break;


        case 7:
           printf("\n\tProgram Terminated\n");
           return 1;


        default:

           printf("\n\tInvalid Choice\n");
     }


     printf("\n\nDo you want to continue? ");
scanf(" %c", &user_continue);

  }


  return 0;
}
struct node* create(int data)
{
  struct node* new_node = (struct node*) malloc (sizeof(struct node));
if (new_node == NULL)

  {
     printf("\nMemory can&'t be allocated\n");
return NULL;
```

```
  }
  new_node->data = data;
new_node->left = NULL;
new_node->right = NULL;      return
new_node;

}
struct node* rotate_left(struct node* root)
{
  struct node* right_child = root->right;
root->right = right_child->left;
right_child->left = root;      root->ht =
height(root);      right_child->ht =
height(right_child);    return right_child;

}
struct node* rotate_right(struct node* root)
{
  struct node* left_child = root->left;      root-
>left = left_child->right;      left_child->right =
root;


  root->ht = height(root);      left_child-
>ht = height(left_child);              return
left_child;

}

int balance_factor(struct node* root)
{     int lh, rh;      if (root
== NULL)          return 0;
if (root->left == NULL)
```

```
    lh = 0;    else    lh =
1 + root->left->ht;    if
(root->right == NULL)

    rh = 0;    else    rh = 1
+ root->right->ht;    return
lh - rh;

}
int height(struct node* root)
{    int lh,
rh;    if
(root ==
NULL)

  {
    return 0;
  }
  if (root->left == NULL)
    lh = 0;    else    lh =
1 + root->left->ht;    if
(root->right == NULL)

    rh = 0;    else    rh = 1
+ root->right->ht;


  if (lh > rh)
return (lh);
return (rh);

}


struct node* insert(struct node* root, int data)
{
  if (root == NULL)
```

```
  {
     struct node* new_node = create(data);
if (new_node == NULL)
     {
        return NULL;
     }
     root = new_node;
  }
  else if (data > root->data)
  {
          root->right = insert(root->right, data);        if (balance_factor(root) == -2)
     {
        if (data > root->right->data)
        {
           root = rotate_left(root);
        }
else
        {
           root->right = rotate_right(root->right);
root = rotate_left(root);
        }
     }
}
else
  {
     root->left = insert(root->left, data);
if (balance_factor(root) == 2)
     {
        if (data < root->left->data)
        {
```

```
            root = rotate_right(root);
        }
else
        {
            root->left = rotate_left(root->left);
root = rotate_right(root);
        }
      }
   }
   root->ht = height(root);
return root;
}

struct node * delete(struct node *root, int x)
{
   struct node * temp = NULL;

   if (root == NULL)
   {
      return NULL;
   }

   if (x > root->data)
   {
      root->right = delete(root->right, x);
if (balance_factor(root) == 2)
      {
         if (balance_factor(root->left) >= 0)
         {
            root = rotate_right(root);
```

```
            70
        }
else
        {
            root->left = rotate_left(root->left);
root = rotate_right(root);
        }
      }
  }
  else if (x < root->data)
  {
     root->left = delete(root->left, x);


            if (balance_factor(root) == -2)
     {
        if (balance_factor(root->right) <= 0)
        {
            root = rotate_left(root);
        }
else
{
root->right = rotate_right(root->right);   root
= rotate_left(root);
}
}
} else
{
if (root->right != NULL)
{
```

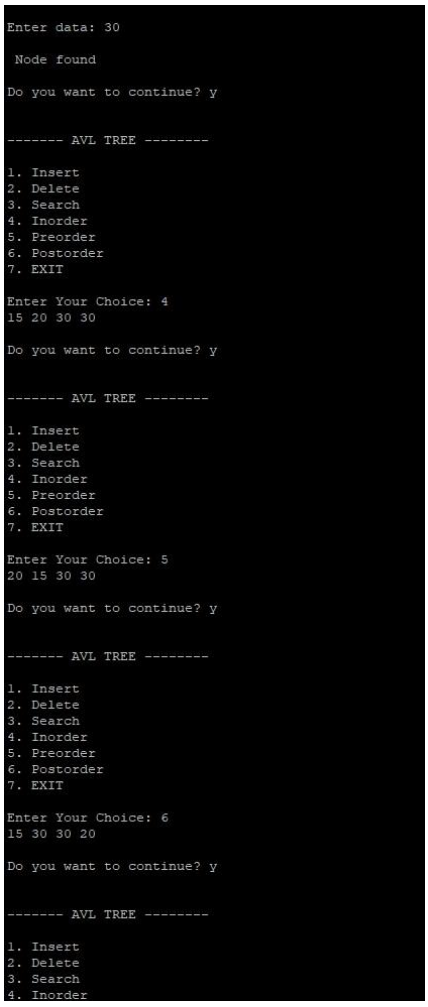
```
temp = root->right;   while
(temp->left != NULL)   temp
= temp->left;


root->data = temp->data;   root->right =
delete(root->right, temp->data);   if
(balance_factor(root) == 2)
{
if (balance_factor(root->left) >= 0)
{
root = rotate_right(root);
}


else
{
root->left = rotate_left(root->left);   root
= rotate_right(root);
}
}
}
else
{
return (root->left);
}
}
root->ht = height(root);   return
(root);
}
struct node* search(struct node* root, int key)
{
```

```
 if(root == NULL)
{
return NULL;
}

if(root->data == key)
{
return root;
}
if(key > root->data)
{
search(root->right, key);
}

else
{       search(root->left, key);

}
}
void inorder(struct node* root)
{
if (root == NULL)
{  return;

}
inorder(root->left);   printf("%d
", root->data);   inorder(root-
>right);

}
void preorder(struct node* root)
{
   if(root == NULL)
{
```

```
return;
}
printf("%d ", root->data); preorder(root-
>left); preorder(root->right);

}
void postorder(struct node* root)
{
if(root == NULL)
{
return;

}

postorder(root->left);
postorder(root->right);   printf("%d
", root->data);

}
```

OUTPUT:

```
------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
5. Preorder
6. Postorder
7. EXIT

Enter Your Choice: 1

Enter data: 15


Do you want to continue? y


------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
5. Preorder
6. Postorder
7. EXIT

Enter Your Choice: 1

Enter data: 20


Do you want to continue? y


------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
5. Preorder
6. Postorder
7. EXIT

Enter Your Choice: 1

Enter data: 25


Do you want to continue? y


------- AVL TREE --------

1. Insert
```



RESULT: Thus, the program was successfully executed.

Ex. No.: 11	Topological Sorting	Date:09/05/2024

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

Algorithm:

1) Start
2) Initialize an empty stack to store the topologically sorted elements.
3) Initialize a set to track visited nodes.
4) Start a depth-first search (DFS) from any unvisited node in the graph.
5) During DFS traversal: a. Mark the current node as visited.
b. Recursively visit all adjacent nodes that are not visited yet.
6) Once a node has no unvisited adjacent nodes, push it onto the stack.
7) Repeat steps 3-5 until all nodes are visited.
Pop elements from the stack to get the topologically sorted order. 8) Stop

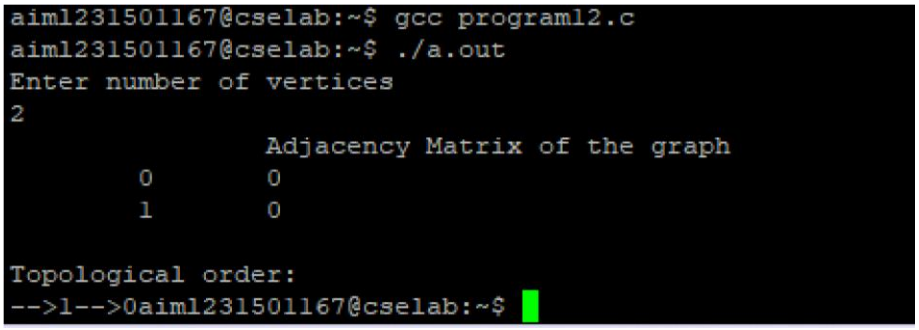
PROGRAM:

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

int s[100], j, res[100; void
AdjacencyMatrix(int a[][100], int n) {
 int i, j; for (i = 0; i < n;
i++) { for (j = 0; j <= n;
j++) { a[i][j] = 0;
} } for (i = 1; i < n;
i++) { for (j = 0; j < i;
j++) { a[i][j] = rand()
% 2; a[j][i] = 0;
}
}
}

void dfs(int u, int n, int a[][100]) {
int v; s[u] = 1; for (v = 0; v < n - 1;
v++) { if (a[u][v] == 1 && s[v] ==
0) { dfs(v, n, a);

}
} j +=
1;
res[j]
= u;
}
```

```
void topological_order(int n, int a[][100]) {
int i, u; for (i = 0; i < n;
i++) { s[i] = 0; } j = 0;
for (u = 0; u < n; u++) {
if (s[u] == 0) { dfs(u, n,
a);
}
}
return;
}
int main() { int
a[100][100], n, i, j;

printf("Enter number of vertices\n");  scanf("%d",
&n);

AdjacencyMatrix(a, n);  printf("\t\tAdjacency Matrix of the graph\n"); /* PRINT
ADJACENCY MATRIX */

for (i = 0; i < n; i++) {
for (j = 0; j < n; j++) {
printf("\t%d", a[i][j]);
}
printf("\n");
}

printf("\nTopological order:\n");

topological_order(n, a);
```

```
for (i = n; i >= 1; i--) { printf("-->%d",
res[i]);

}
return 0;
}
```

OUTPUT:



RESULT: Thus, the program was successfully executed.

Ex. No.:12	Graph Traversal	Date:09/05/2024

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

Algorithm:

DFS

1)Start with an empty stack and a set to track visited nodes.
2) Push the starting node onto the stack and mark it as visited.
3) While the stack is not empty, do the following:
4) If the stack is not empty, pop a node from the stack.
Process the node.
5) For each unvisited neighbor of the node, push the neighbor onto the stack and mark it as visited.
6) Repeat steps 3-6 until the stack is empty.
7) Stop

BFS

1) Start with an empty queue and a set to track visited nodes.
2)Enqueue the starting node into the queue and mark it as visited.
3) While the queue is not empty, do the following:
4) If the queue is not empty, dequeue a node from the queue.
Process the node.
5) For each unvisited neighbor of the node, enqueue the neighbor into the queue and mark it as visited.
6) Repeat steps 3-6 until the queue is empty.
7) Stop

PROGRAM: BFS

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

struct node { int
vertex; struct
node* next;

}; struct adj_list {
struct node* head;

};

struct graph { int
num_vertices; struct
adj_list* adj_lists; int*
visited;

};
struct node* new_node(int vertex) { struct node* new_node =
(struct node*)malloc(sizeof(struct node)); new_node->vertex =
vertex; new_node->next = NULL; return new_node;

}
struct graph* create_graph(int n) { struct graph* graph = (struct
graph*)malloc(sizeof(struct graph)); graph->num_vertices = n;
graph->adj_lists = (struct adj_list*)malloc(n * sizeof(struct adj_list));
graph->visited = (int*)malloc(n * sizeof(int));



int i; for (i = 0; i< n; i++) { graph-
>adj_lists[i].head = NULL; graph-
>visited[i] = 0;
```

```
}

return graph;
}
void add_edge(struct graph* graph, int src, int dest) {
struct node* new_node1 = new_node(dest);
new_node1->next = graph->adj_lists[src].head;
graph->adj_lists[src].head = new_node1; struct node*
new_node2 = new_node(src); new_node2->next =
graph->adj_lists[dest].head; graph-
>adj_lists[dest].head = new_node2;

}
void bfs(struct graph* graph, int v) { int queue[1000]; int
front = -1; int rear = -1; graph->visited[v] = 1;
queue[++rear] = v; while (front != rear) { int current_vertex
= queue[++front]; printf("%d ", current_vertex); struct
node* temp = graph->adj_lists[current_vertex].head; while
(temp != NULL) { int adj_vertex = temp->vertex;




if (graph->visited[adj_vertex] == 0) { graph-
>visited[adj_vertex] = 1; queue[++rear] =
adj_vertex;

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

```
int main() { struct graph* graph =
create_graph(6); add_edge(graph, 0, 1);
add_edge(graph, 0, 2); add_edge(graph, 1, 3);
add_edge(graph, 1, 4); add_edge(graph, 2, 4);
add_edge(graph, 3, 4); add_edge(graph, 3, 5);
add_edge(graph, 4,5); printf("BFS traversal
starting from vertex 0: "); bfs(graph, 0);


return 0;
}
```

DFS:

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

// Globally declared visited array int
vis[100];



struct Graph {
    int V;
int E;
int** Adj;

};

struct Graph* adjMatrix()
{
    struct Graph* G = (struct Graph*)
malloc(sizeof(struct Graph));     if
(!G) {         printf("Memory Error\n");
return NULL;

    }
```

```
    G->V = 7;
    G->E = 7;

    G->Adj = (int**)malloc((G->V) * sizeof(int*));
for (int k = 0; k < G->V; k++) {
        G->Adj[k] = (int*)malloc((G->V) * sizeof(int));
    }

    for (int u = 0; u < G->V; u++) {
for (int v = 0; v < G->V; v++) {
            G->Adj[u][v] = 0;
        }
    }
    G->Adj[0][1] = G->Adj[1][0] = 1;
    G->Adj[0][2] = G->Adj[2][0] = 1;     G-
>Adj[1][3] = G->Adj[3][1] = 1;     G->Adj[1][4] =
G->Adj[4][1] = 1;      G->Adj[1][5] = G-
>Adj[5][1] = 1;
     G->Adj[1][6] = G->Adj[6][1] = 1;     G-
         >Adj[6][2] = G->Adj[2][6] = 1;

    return G;
}
 void DFS(struct Graph* G, int u)

{     vis[u] = 1;     printf("%d ", u);
for (int v = 0; v < G->V; v++) {
if (!vis[v] && G->Adj[u][v]) {
            DFS(G, v);
        }
    }
}
```

```
void DFStraversal(struct Graph* G)
{     for (int i = 0; i < 100; i++)
{         vis[i] = 0;

    }
    for (int i = 0; i < G->V; i++) {
if (!vis[i]) {

            DFS(G, i);
        }
    }
}
 void main()
{


 struct Graph* G;
    G = adjMatrix();
    DFStraversal(G);
}
```

OUTPUT:

```
BFS traversal starting from vertex 0: 0 2 1 4 3 5 6

0 1 3 4 5 6 2
```

RESULT: Thus, the program was successfully executed.

Ex. No.:13	Graph Traversal	Date:16/05/2014

Write a C program to create a graph and find a minimum spanning tree using prim's algorithm.

Algorithm:

1) Start
2) Initialize an empty set to store the minimum spanning tree (MST) and a priority queue to store edges and their weights.
3) Choose a starting node and add it to the MST set.
4) For each edge connected to the starting node, add the edge to the priority queue.
5) While the priority queue is not empty:
a. Extract the edge with the smallest weight from the priority queue.
b. If adding the edge to the MST set does not create a cycle, add the edge to the MST set.
c. For each neighbour of the newly added node in the MST set:
i. If the neighbour is not already in the MST set, add the edge connecting the neighbor to the priority queue.
6) Repeat step 4 until all nodes are included in the MST set.
7) The edges in the MST set form the minimum spanning tree of the graph. 8) Stop

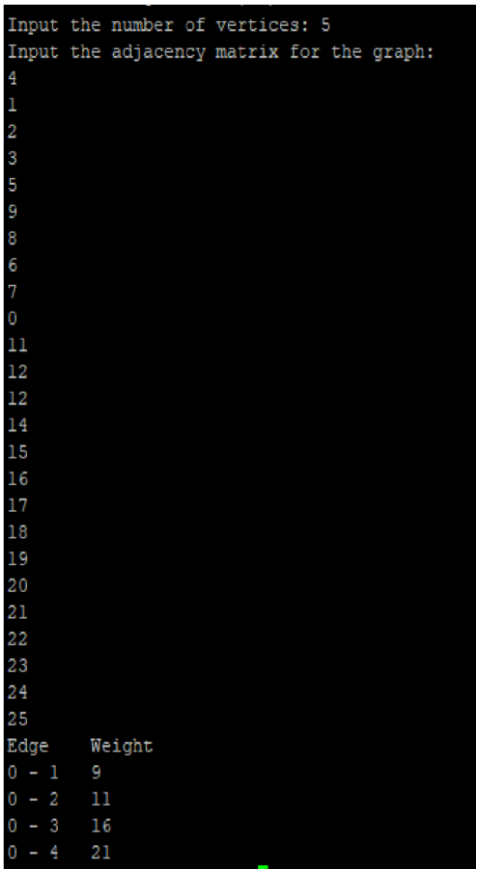
PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
struct node { int
vertex; struct node*
next;

}; struct adj_list {
struct node* head;

};
struct graph { int
num_vertices; struct
adj_list* adj_lists; int*
visited;

};
```

```
struct node* new_node(int vertex) { struct node* new_node =
(struct node*)malloc(sizeof(struct node)); new_node->vertex =
vertex; new_node->next = NULL; return new_node;

}
struct graph* create_graph(int n) { struct graph* graph = (struct
graph*)malloc(sizeof(struct graph)); graph->num_vertices = n;
graph->adj_lists = (struct adj_list*)malloc(n * sizeof(struct adj_list));
graph->visited = (int*)malloc(n * sizeof(int));


int i; for (i = 0; i< n;
i++) { graph-
>adj_lists[i].head =
NULL; graph-
>visited[i] = 0;

}


return graph;
}


void add_edge(struct graph* graph, int src, int dest) {
struct node* new_node1 = new_node(dest);
new_node1->next = graph->adj_lists[src].head;
graph->adj_lists[src].head = new_node1; struct node*
new_node2 = new_node(src); new_node2->next =
graph->adj_lists[dest].head; graph-
>adj_lists[dest].head = new_node2;

}
void bfs(struct graph* graph, int v) { int queue[1000]; int
front = -1; int rear = -1; graph->visited[v] = 1;
queue[++rear] = v; while (front != rear) { int current_vertex
= queue[++front]; printf("%d ", current_vertex); struct
```

```
node* temp = graph->adj_lists[current_vertex].head; while
(temp != NULL) { int adj_vertex = temp->vertex; if (graph-
>visited[adj_vertex] == 0) { graph->visited[adj_vertex] = 1;
queue[++rear] = adj_vertex;

}

temp = temp->next;
}
}
}

int main() { struct graph* graph =
create_graph(6); add_edge(graph, 0, 1);
add_edge(graph, 0, 2); add_edge(graph, 1, 3);
add_edge(graph, 1, 4); add_edge(graph, 2, 4);
add_edge(graph, 3, 4); add_edge(graph, 3, 5);
add_edge(graph, 4,5); printf("BFS traversal
starting from vertex 0: "); bfs(graph, 0);


return 0;
}
```

OUTPUT:



RESULT: Thus, the program was successfully executed.

Ex. No.:14	Graph Traversal	Date: 16/05/2024

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

Algorithm:

1) Start
2) Initialize the distance from the start node to all other nodes as infinity, except for the start node itself which is 0.
3) Create a priority queue to store nodes and their distances from the start node.
4) Add the start node to the priority queue with a distance of 0.
5) While the priority queue is not empty:
a. Extract the node with the smallest distance from the priority queue.
b. For each neighbor of the extracted node:
i. Calculate the distance from the start node to the neighbor through the extracted node.
ii. If this distance is smaller than the current distance stored for the neighbor, update the distance. iii. Add the neighbor to the priority queue with the updated distance.
6) Repeat step 4 until all nodes have been processed.
7) The distances stored for each node after the algorithm completes represent the shortest path from the start node to that node. 8) Stop

PROGRAM;

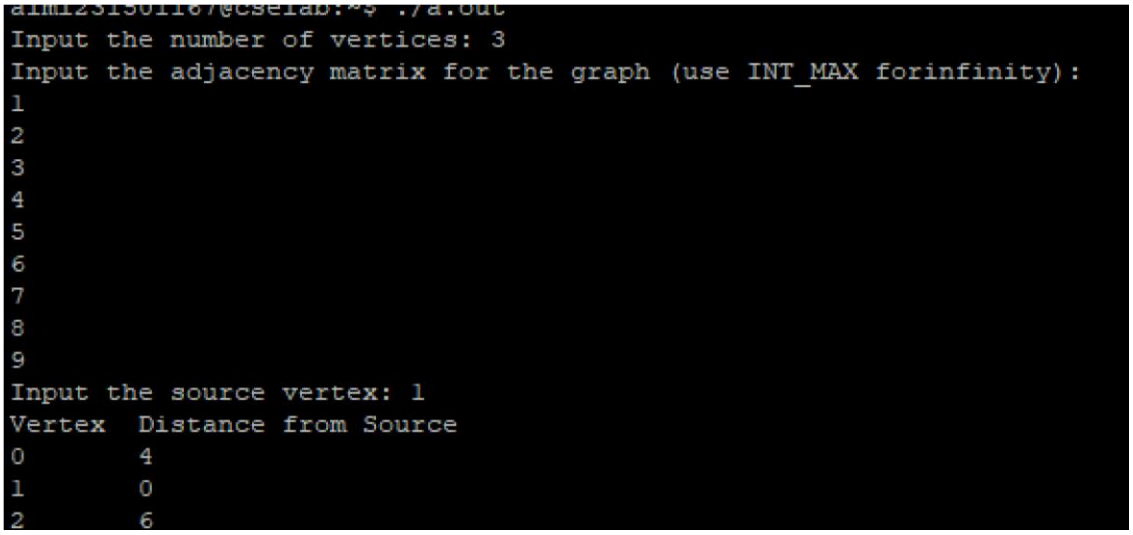
```
#include <stdio.h>

#include <limits.h>

#define MAX_VERTICES 100
```

```
int minDistance(int dist[], int sptSet[], int vertices) {
int min = INT_MAX, minIndex; for (int v = 0; v <
vertices; v++) { if (!sptSet[v] && dist[v] < min) { min
= dist[v]; minIndex = v;

}
}
return minIndex;
}
void printSolution(int dist[], int vertices) {
printf("Vertex \tDistance from Source\n");
for (int i = 0; i < vertices; i++) { printf("%d
\t%d\n", i, dist[i]);

}
}
void dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int src, int
vertices) { int dist[MAX_VERTICES]; int sptSet[MAX_VERTICES];

for (int i = 0; i < vertices; i++) {
dist[i] = INT_MAX; sptSet[i] =
0;

} dist[src] =
0; for (int
count = 0;
count <
vertices - 1;
count++) {
int u =
minDistance
(dist, sptSet,
vertices);
sptSet[u] =
1;
```

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

}
}
}
printSolution(dist, vertices);
}
int main() { int vertices; printf("Input the
number of vertices: "); scanf("%d", &vertices); if
(vertices <= 0 || vertices > MAX_VERTICES) {
printf("Invalid number of vertices. Exiting...\n");
return 1;

}
int graph[MAX_VERTICES][MAX_VERTICES]; printf("Input the adjacency matrix
for the graph (use INT_MAX forinfinity):\n"); for (int i = 0; i < vertices; i++) {


for (int j = 0; j < vertices; j++) { scanf("%d",
&graph[i][j]);

}
}
int source;
printf("Input the source vertex: ");
scanf("%d", &source); if (source < 0 ||
source >= vertices) { printf("Invalid source
vertex. Exiting...\n"); return 1;

}
dijkstra(graph, source, vertices); return
0;
```

```
}
```

OUTPUT:



RESULT: Thus, the program was successfully executed.

Ex. No.:15	Sorting	Date:23/05/2024

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

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

Algorithm:

1) Start
2) If the array has fewer than two elements, return it as it is already sorted.
3) Divide the array into two halves.
4) Recursively sort the two halves using Merge Sort.
5) Merge the sorted halves into a single sorted array.
6) Choose a sorting algorithm (e.g., Bubble Sort, Merge Sort, Quick Sort).
7) Implement the selected sorting algorithm.
8) Pass the unsorted array to the sorting algorithm.
9) Execute the sorting algorithm to sort the array. Obtain the sorted array as output. 10) stop

PROGRAM:

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 arr[] = { 19, 17, 15, 12, 16, 18, 4, 11, 13 };
int n = sizeof(arr) / sizeof(arr[0]);
printf("Original array:'); for (int i = 0; i<n; i++)
{ printf("&%d", arr[i]);

}
quickSort(arr, 0, n 0;
printf("\nSorted array:"); for
(int i = 0; i &lt; n; i++) {
printf("%d", arr[i]);

}


return 0;
}
```

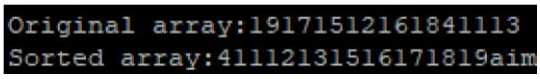
MERGE SORT

```
#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) { int m = l + (r -
l) / 2; 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 arr[] = { 12, 11, 13, 5, 6, 7 }; int
arr_size = sizeof(arr) / sizeof(arr[0]);
printf("Given array is \n");
printArray(arr, arr_size); mergeSort(arr,
0, arr_size - 1); printf("\nSorted array is
\n"); printArray(arr, arr_size); return 0;

}
```

OUTPUT:



RESULT: Thus, the program was successfully executed.

Ex. No.: 16	Hashing	Date:30/05/2024

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

Algorithm:

1. Open Addressing:

1. Compute the hash of the key to be inserted.
2. Check the computed hash index in the hash table.
3. If the slot is empty, insert the key.
4. If the slot is occupied, then it means a collision has occurred. In this case, move to the next slot in the hash table.
5. Repeat the process until an empty slot is found.

2. Closed Addressing (Separate Chaining):

1. Compute the hash of the key to be inserted.
2. Check the computed hash index in the hash table.
3. If the slot is empty, insert the key.
4. If the slot is occupied, then it means a collision has occurred. In this case, add the new key to the linked list at that slot.

3. Rehashing:

1. When the load factor of the hash table reaches a certain threshold (typically > 0.7), create a new hash table of larger size.
2. Compute the hash of each key in the old table.
3. Insert each key into the new table.
4. Delete the old table.

PROGRAM:

A. OPEN ADDRESSING:

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

#define TSIZE 12

 typedef struct Node {          int k;
    struct Node* n;
} Node;

int hFunc(int k) {
    return k % TSIZE;
}

// Open Addressing Hash Table
int oaHashTable[TSIZE];

void initOA() {
    for (int i = 0; i < TSIZE; i++) {
        oaHashTable[i] = -1;
    } }
void insOA(int k) {
int idx = hFunc(k);
int origIdx = idx;
int i = 1;

    while (oaHashTable[idx] != -1) {
idx = (origIdx + i) % TSIZE;
i++;
    }
    oaHashTable[idx] = k;}

             void dispOA() {
printf("OA Hash Table:\n");     for
(int i = 0; i < TSIZE; i++) {         if
(oaHashTable[i] != -1) {
            printf("%d ", oaHashTable[i]);
        } else {
            printf("- ");
        }
    }
    printf("\n");
```

```
}
}
```

B. CLOSED ADDRESSING;

```
#include <stdio.h>

#define max 10

int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 }; int
b[10];

void merging(int low, int mid, int high) {
int l1, l2, i;

for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
if(a[l1] <= a[l2]) b[i] = a[l1++]; else b[i] = a[l2++];
}
while(l1 <= mid) b[i++]
= a[l1++]; while(l2 <=
high) b[i++] = a[l2++];

for(i = low; i <= high; i++)
a[i] = b[i];
}

void sort(int low, int high) { int
mid;

if(low < high) { mid =
(low + high) / 2; sort(low,
mid); sort(mid+1, high);
merging(low, mid, high);
} else { return;
}
}
```

```
int main() { int
i;

printf("List before sorting\n");

for(i = 0; i <= max; i++)
printf("%d ", a[i]);

sort(0, max);

printf("\nList after sorting\n");

for(i = 0; i <= max; i++) printf("%d
", a[i]);
}
```

C. REHASHING:

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

typedef struct Node {
int  key;  int  value;
struct Node* next;
} Node;

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

Node* createNode(int key, int value) { Node*
newNode = (Node*)malloc(sizeof(Node));
newNode->key = key; newNode->value = value;
newNode->next = NULL; return newNode;
}

HashTable* createTable(int size) {
HashTable* newTable = (HashTable*)malloc(sizeof(HashTable));
newTable->size = size; newTable->count = 0;
newTable->table = (Node**)malloc(sizeof(Node*) * size); for
(int i = 0; i < size; i++) {

newTable->table[i] = NULL;
}
return newTable;
}

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

void insert(HashTable* hashTable, int key, int value);
```

```
void rehash(HashTable* hashTable) {
int oldSize = hashTable->size; Node**
oldTable = hashTable->table;

// New size is typically a prime number or double the old size int
newSize = oldSize * 2;
hashTable->table = (Node**)malloc(sizeof(Node*) * newSize);
hashTable->size = newSize; hashTable->count = 0;
for (int i = 0; i < newSize; i++) {
hashTable->table[i] = NULL;
}
for (int i = 0; i < oldSize; i++) {
Node* current = oldTable[i]; while
(current != NULL) {
insert(hashTable, current->key, current->value);
Node* temp = current;

current = current->next; free(temp);
}
}

free(oldTable);
}

void insert(HashTable* hashTable, int key, int value) { if
((float)hashTable->count / hashTable->size >= 0.75) {
rehash(hashTable);
}

int hashIndex = hashFunction(key, hashTable->size);
Node* newNode = createNode(key, value); newNode->next
= hashTable->table[hashIndex]; hashTable-
>table[hashIndex] = newNode; hashTable->count++;
}

int search(HashTable* hashTable, int key) { int
hashIndex = hashFunction(key, hashTable->size);
Node* current = hashTable->table[hashIndex];
while (current != NULL) { if
(current->key == key) {
return current->value;
}
current = current->next;
}

return -1;
}

void delete(HashTable* hashTable, int key) { int
hashIndex = hashFunction(key, hashTable->size);
Node* current = hashTable->table[hashIndex]; Node*
prev = NULL;
while (current != NULL && current->key != key) { prev
= current;
current = current->next;
```

```
}
if (current == NULL) { return;
}
if (prev == NULL) {
hashTable->table[hashIndex] = current->next;
} else {
prev->next = current->next;

}
free(current);
hashTable->count--;
}

void freeTable(HashTable* hashTable) {
for (int i = 0; i < hashTable->size; i++) {
Node* current = hashTable->table[i];
while (current != NULL) { Node* temp =
current; current = current->next;

free(temp);
}
}
free(hashTable->table); free(hashTable);
}

int main() {
HashTable* hashTable = createTable(5);

insert(hashTable, 1, 10); insert(hashTable,
2, 20); insert(hashTable, 3, 30);
insert(hashTable, 4, 40); insert(hashTable,
5, 50);
insert(hashTable, 6, 60); // This should trigger rehashing

printf("Value for key 1: %d\n", search(hashTable, 1)); printf("Value
for key 2: %d\n", search(hashTable, 2)); printf("Value for key 3:
%d\n", search(hashTable, 3)); printf("Value for key 4: %d\n",
search(hashTable, 4)); printf("Value for key 5: %d\n",
search(hashTable, 5)); printf("Value for key 6: %d\n",
search(hashTable, 6));

delete(hashTable, 3);
printf("Value for key 3 after deletion: %d\n", search(hashTable, 3));

freeTable(hashTable);
return 0;

}
```

OUTPUT:

```
aiml2315011167@cselab:~$ gcc program16C.c
aiml2315011167@cselab:~$ ./a.out
Value for key 1: 10
Value for key 2: 20
Value for key 3: 30
Value for key 4: 40
Value for key 5: 50
Value for key 6: 60
Value for key 3 after deletion: -1
aiml2315011167@cselab:~$
```

```
aiml2315011167@cselab:~$ ./a.out
Value for key 1: 10
Value for key 2: 20
Value for key 12: 30
Value for key 3: -1
Value for key 2 after deletion: -1
aiml2315011167@cselab:~$
```

```
aiml2315011167@cselab:~$ gcc program16A.c
aiml2315011167@cselab:~$ ./a.out
List before sorting
10 14 19 26 27 31 33 35 42 44 0
List after sorting
0 10 14 19 26 27 31 33 35 42 44 aiml2315011167@cselab:~$
```

RESULT: Thus, the program was successfully executed.

Rajalakshmi Institutions

Rajalakshmi Engineering College

Rajalakshmi Nagar Thandalam, Chennai - 602 105.

Phone : +91 -44 -67181111, 67181112

Website : www.rajalakshmi.or g