SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)

SRM NAGAR, KATTANKULATHUR-603203

DEPARTMENT OF ELECTRONICS AND INSTRUMENTATION ENGINEERING

LAB MANUAL

CS3666-DATA STRUCTURES LABORATORY

VI SEMESTER

(Academic Year – 2025-2026 EVEN)

Prepared By

Ms.M.Shanthi, A.P. (Sr.G) / EIE

LIST OF EXPERIMENTS

EX. 1 ARRAY IMPLEMENTATION OF LIST ADT

Date:

Aim: To create a list using array and perform operations such as display, insertions and deletions.

Algorithm

1. Start
2. Define list using an array of size n.
3. First item with subscript or position = 0
4. Display menu on list operation
5. Accept user choice
6. If choice = 1 then
7. Locate node after which insertion is to be done
8. Create an empty location in array after the node by moving the existing elements one position ahead.
9. Insert the new element at appropriate position
10. Else if choice = 2
11. Get element to be deleted.
12. Locate the position of the element replace the element with the element in the following position.
13. Repeat the step for successive elements until end of the array.
14. Else
15. Traverse the list from position or subscript 0 to n.
16. Stop

PROGRAM

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
void create();
void insert();
void deletion();
void search();
void display();
int a,b[20],n,p,e,f,i,pos;

void main()
{
//clrscr();
int ch;
char g='y';

        do
        {
                printf("\n main Menu");
                printf("\n 1.Create \n 2.Delete \n 3.Search \n 4.Insert \n 5.Display\n 6.Exit \n");
                printf("\n Enter your Choice");
                scanf("%d", &ch);
        switch(ch)
```

```
        {
        case 1:
        create();
        break;
        case 2:
        deletion();
        break;
        case 3:
        search();
        break;
        case 4:
        insert();
        break;
        case 5:
        display();
        break;
        case 6:
        exit();
        break;
        default:
        printf("\n Enter the correct choice:");
        }
        printf("\n Do u want to continue:::");
        scanf("\n%c", &g);
        }
        while(g=='y'||g=='Y');
//getch();
}
void create()
{
printf("\n Enter the number of nodes");
scanf("%d", &n);
        for(i=0;i<n;i++)
        {
        printf("\n Enter the  Element:",i+1);
        scanf("%d", &b[i]);
        }

}

void deletion()
{
printf("\n Enter the position u want to delete::");
scanf("%d", &pos);
        if(pos>=n)
        {
        printf("\n Invalid Location::");
        }
        else
        {
        for(i=pos+1;i<n;i++)
```

```
        {
        b[i-1]=b[i];
        }
        n--;
        }
        printf("\n The Elements after deletion");
        for(i=0;i<n;i++)
        {
        printf("\t%d", b[i]);
        }
        }
void search()
{
printf("\n Enter the Element to be searched:");
scanf("%d", &e);
        for(i=0;i<n;i++)
        {
        if(b[i]==e)
        {
        printf("\n Value is in the %d Position",i);
        }
        else
        {
        printf("\n Value is not in the %d Position",i);
        continue;
        }
        }
}

void insert()
{
printf("\n Enter the position u need to insert::");
scanf("%d", &pos);

        if(pos>=n)
        {
        printf("\n invalid Location::");
        }
        else
        {
        for(i=MAX-1;i>=pos-1;i--)
        {
        b[i+1]=b[i];
        }
        printf("\n Enter the element to insert::\n");
        scanf("%d",&p);
        b[pos]=p;
        n++;
        }
 printf("\n The list after insertion::\n");
display();
```

```
}


void display()
{
printf("\n The Elements of The list ADT are:");
for(i=0;i<n;i++)
{
printf("\n\n%d", b[i]);
}
}
```

Result: Thus created a list using array and performed operations such as display, insertions and deletions.

Ex. No. 2a ARRAY IMPLEMENTATION OF STACK ADT

Date:

Aim: To implement stack operations using array.

Algorithm

1. Start
2. Define a array *stack* of size *max* = 5
3. Initialize *top* = -1
4. Display a menu listing stack operations
5. Accept choice
6. If choice = 1 then
7. If top < max -1
8. Increment top
9. Store element at current position of top
10. Else
11. Print Stack overflow
12. If choice = 2 then
13. If top < 0 then
14. Print Stack underflow
15. Else
16. Display current top element
17. Decrement top
18. If choice = 3 then
19. Display stack elements starting from top
20. Stop

Program

```
/* 1a - Stack Operation using Arrays */
#include<stdio.h>
#include<conio.h>
#define max 5
static int stack[max];
int top = -1;
void push(int x)
        {
        stack[++top] = x;
        }
int pop()
        {
        return (stack[top--]);
        }
void view()
        {
        int i;
        if (top < 0)
                {
```

```
                printf("\n Stack Empty \n");
                }
        else
        {
        printf("\n Top-->");
        }
        for(i=top; i>=0; i--)
                {
                printf("%4d", stack[i]);
                }
        printf("\n");
        }

void main()
{
        int ch=0, val;
        clrscr();
        while(ch != 4)
                {
        printf("\n STACK OPERATION \n");
        printf("1.PUSH ");
        printf("2.POP ");
        printf("3.VIEW ");
        printf("4.QUIT \n");
        printf("Enter Choice : ");
        scanf("%d", &ch);
switch(ch)
{
case 1:
        if(top<max-1)
        {
        printf("\nEnter Stack element : ");
        scanf("%d", &val);
        push(val);
        }
        else
        {
        printf("\n Stack Overflow \n");
        }
        break;
case 2:
        if(top < 0)
        {
        printf("\n Stack Underflow \n");
```

```
        }
        else
        {
        val = pop();
        printf("\n Popped element is %d\n", val);
        }
        break;
case 3:
        view();
        break;
case 4:
        exit(0);
default:
        printf("\n Invalid Choice \n");
}
}
}
```

Output

```
STACK OPERATION
1.PUSH 2.POP 3.VIEW 4.QUIT
Enter Choice : 1
Enter Stack element : 12
STACK OPERATION
1.PUSH 2.POP 3.VIEW 4.QUIT
Enter Choice : 1
Enter Stack element : 23
STACK OPERATION
1.PUSH 2.POP 3.VIEW 4.QUIT
Enter Choice : 1
Enter Stack element : 34
STACK OPERATION
1.PUSH 2.POP 3.VIEW 4.QUIT
Enter Choice : 1
Enter Stack element : 45
STACK OPERATION
1.PUSH 2.POP 3.VIEW 4.QUIT
Enter Choice : 3
Top--> 45 34 23 12
STACK OPERATION
1.PUSH 2.POP 3.VIEW 4.QUIT
Enter Choice : 2
Popped element is 45
```

```
STACK OPERATION
1.PUSH 2.POP 3.VIEW 4.QUIT
Enter Choice : 3
Top--> 34 23 12
STACK OPERATION
1.PUSH 2.POP 3.VIEW 4.QUIT
Enter Choice : 4
```

Result: Thus push and pop operations of a stack was demonstrated using arrays.

Ex. No. 2b ARRAY IMPLEMENTATION OF QUEUE ADT

Date:

Aim: To implement queue operations using array.

Algorithm

1. Start
2. Define a array *queue* of size *max* = 5
3. Initialize *front = rear* = –1
4. Display a menu listing queue operations
5. Accept choice
6. If choice = 1 then
7. If rear < max -1
8. Increment rear
9. Store element at current position of rear
10. Else
11. Print Queue Full
12. If choice = 2 then
13. If front = –1 then
14. Print Queue empty
15. Else
16. Display current front element
17. Increment front
18. If choice = 3 then
19. Display queue elements starting from front to rear.
20. Stop

Program

```
include<stdio.h>
#include<conio.h>
#define max 5
static int queue[max];
int front = -1;
int rear = -1;
int remove();
void insert(int x)
{
        queue[++rear] = x;
        if (front == -1)
        {
        front = 0;
        }
}
int remove_myval()
{
```

```
        int val;
        val = queue[front];
        if (front==rear && rear==max-1)
        {
        front = rear = -1;
        }
        else
        {
        front ++;
        }
    return(val);
}
void view()
        {
        int i;
        if (front == -1)
        {
        printf("\n Queue Empty \n");
        }
        else
        {
        printf("\n Front-->");
        }
        for(i=front; i<=rear; i++)
        {
        printf("%d", queue[i]);
        printf(" <--Rear\n");
        }
        }
void main()
        {
        int ch= 0,val;
        clrscr();
        while(ch != 4)
        {
        printf("\n QUEUE OPERATION \n");
        printf("1.INSERT ");
        printf("2.DELETE ");
        printf("3.VIEW ");
        printf("4.QUIT\n");
        printf("Enter Choice : ");
        scanf("%d", &ch);
switch(ch)
{
```

```
case 1:
        if(rear < max-1)
        {
        printf("\n Enter element to be inserted : ");
        scanf("%d", &val);
        insert(val);
        }
        else
        {
        printf("\n Queue Full \n");
        }
        break;
case 2:
        if(front == -1)
        {
        printf("\n Queue Empty \n");
        }
        else
        {
        val=remove_myval();
        printf("\n Element deleted : %d \n", val);
        }
        break;
case 3:
        view();
        break;
case 4:
        exit(0);
        default:
        printf("\n Invalid Choice \n");
}
}
}
```

Output

```
QUEUE OPERATION
1.INSERT 2.DELETE 3.VIEW 4.QUIT
Enter Choice : 1
Enter element to be inserted : 12
QUEUE OPERATION
1.INSERT 2.DELETE 3.VIEW 4.QUIT
Enter Choice : 1
Enter element to be inserted : 23
QUEUE OPERATION
1.INSERT 2.DELETE 3.VIEW 4.QUIT
Enter Choice : 1
```

```
Enter element to be inserted : 34
QUEUE OPERATION
1.INSERT 2.DELETE 3.VIEW 4.QUIT
Enter Choice : 1
Enter element to be inserted : 45
QUEUE OPERATION
1.INSERT 2.DELETE 3.VIEW 4.QUIT
Enter Choice : 1
Enter element to be inserted : 56
QUEUE OPERATION
1.INSERT 2.DELETE 3.VIEW 4.QUIT
Enter Choice : 1
Queue Full
QUEUE OPERATION
1.INSERT 2.DELETE 3.VIEW 4.QUIT
Enter Choice : 3
Front--> 12 23 34 45 56 <--Rear
```

Result: Thus insert and delete operations of a queue was demonstrated using arrays.

EX. 3a LINKED LIST IMPLEMENTATION OF LIST [SINGLY LINKED LIST]

Date:

Aim: To define a singly linked list node and perform operations such as insertions and deletions dynamically.

Algorithm

1. Start
2. Define single linked list *node* as self referential structure
3. Create *Head* node with label = -1 and next = NULL using
4. Display menu on list operation
5. Accept user choice
6. If choice = 1 then
7. Locate node after which insertion is to be done
8. Create a new node and get data part
9. Insert the new node at appropriate position by manipulating address
10. Else if choice = 2
11. Get node's data to be deleted.
12. Locate the node and delink the node
13. Rearrange the links
14. Else
15. Traverse the list from Head node to node which points to null
16. Stop

Program

```
/* 3c - Single Linked List */
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <alloc.h>
#include <string.h>
struct node
{
        int label;
        struct node *next;
};
main()
{
  int ch, fou=0;
  int k;
  struct node *h, *temp, *head, *h1;
  /* Head node construction */
  head = (struct node*) malloc(sizeof(struct node));
  head->label = -1;
  head->next = NULL;
  while(-1)
  {
    clrscr();
```

```
printf("\n\n SINGLY LINKED LIST OPERATIONS \n");
printf("1->Add ");
printf("2->Delete ");
printf("3->View ");
printf("4->Exit \n");
printf("Enter your choice : ");
scanf("%d", &ch);
switch(ch)
{
    /* Add a node at any intermediate location */
    case 1:
        printf("\n Enter label after which to add : ");
        scanf("%d", &k);
        h = head;
        fou = 0;
        if (h->label == k)
        fou = 1;
        while(h->next != NULL)
        {
           if (h->label == k)
           {
              fou=1;
              break;
           }
            h = h->next;
        }
       if (h->label == k)
            fou = 1;
        if (fou != 1)
             printf("Node not found\n");
       else
       {
          temp=(struct node *)(malloc(sizeof(struct node)));
           printf("Enter label for new node : ");
           scanf("%d" , &temp->label);
           temp->next = h->next;
           h->next = temp;
        }
        break;

    /* Delete any intermediate node */
    case 2:
       printf("Enter label of node to be deleted\n");
       scanf("%d", &k);
       fou = 0;
        h = h1 = head;
       while (h->next != NULL)
```

```
            {
                h = h->next;
                if (h->label == k)
                {
                        fou = 1;
                        break;
                }
            }
            if (fou == 0)
                printf("Sorry Node not found\n");
            else
            {
                while (h1->next != h)
                h1 = h1->next;
                h1->next = h->next;
                free(h);
                printf("Node deleted successfully \n");
            }
            break;

        case 3:
                printf("\n\n HEAD -> ");
                h=head;
                while (h->next != NULL)
                {
                   h = h->next;
                   printf("%d -> ",h->label);
                }
                printf("NULL");
                break;
        case 4:
                exit(0);
        }
    }
}
```

Output

```
SINGLY LINKED LIST OPERATIONS
1->Add 2->Delete 3->View 4->Exit
Enter your choice : 1
Enter label after which new node is to be added : -1
Enter label for new node : 23
SINGLY LINKED LIST OPERATIONS
1->Add 2->Delete 3->View 4->Exit
Enter your choice : 1
Enter label after which new node is to be added : 23
Enter label for new node : 67
```

SINGLY LINKED LIST OPERATIONS
1->Add 2->Delete 3->View 4->Exit
Enter your choice : 3
HEAD -> 23 -> 67 -> NULL

Result: Thus operation on single linked list is performed.

Ex. No. 3b LINKED LIST IMPLEMENTATION OF STACK

Date:

Aim: To implement stack operations using linked list.

Algorithm

1. Start
2. Define a singly linked list node for stack
3. Create Head node
4. Display a menu listing stack operations
5. Accept choice
6. If choice = 1 then
7. Create a new node with data
8. Make new node point to first node
9. Make head node point to new node
10. If choice = 2 then
11. Make temp node point to first node
12. Make head node point to next of temp node
13. Release memory
14. If choice = 3 then
15. Display stack elements starting from head node till null
16. Stop

Program

```
/* 5c - Stack using Single Linked List */
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <alloc.h>
struct node
{
        int label;
        struct node *next;
};
main()
{
        int ch = 0;
        int k;
        struct node *h, *temp, *head;
        /* Head node construction */
        head = (struct node*) malloc(sizeof(struct node));
        head->next = NULL;
        while(1)
        {
                printf("\n Stack using Linked List \n");
                printf("1->Push ");
```

```
                printf("2->Pop ");
                printf("3->View ");
                printf("4->Exit \n");
                printf("Enter your choice : ");
                scanf("%d", &ch);
                switch(ch)
                {
                case 1:
                        /* Create a new node */
                        temp=(struct node *)(malloc(sizeof(struct node)));
                        printf("Enter label for new node : ");
                        scanf("%d", &temp->label);
                        h = head;
                        temp->next = h->next;
                        h->next = temp;
                        break;
                case 2:
                        /* Delink the first node */
                        h = head->next;
                        head->next = h->next;
                        printf("Node %s deleted\n", h->label);
                        free(h);
                        break;
                case 3:
                        printf("\n HEAD -> ");
                        h = head;
                        /* Loop till last node */
                        while(h->next != NULL)
                        {
                                h = h->next;
                                printf("%d -> ",h->label);
                        }
                        printf("NULL \n");
                        break;
                case 4:
                        exit(0);
            }
        }
}
```

Output

```
Stack using Linked List
1->Push 2->Pop 3->View 4->Exit
Enter your choice : 1
Enter label for new node : 23
New node added
Stack using Linked List
```

```
1->Push 2->Pop 3->View 4->Exit
Enter your choice : 1
Enter label for new node : 34
Stack using Linked List
1->Push 2->Pop 3->View 4->Exit
Enter your choice : 3
HEAD -> 34 -> 23 -> NULL
```

Result: Thus push and pop operations of a stack was demonstrated using linked list.

Ex. No. 3c **LINKED LIST IMPLEMENTATION OF QUEUE**
Date:

Aim: To implement queue operations using linked list.

Algorithm

1. Start
2. Define a singly linked list node for stack
3. Create Head node
4. Display a menu listing stack operations
5. Accept choice
6. If choice = 1 then
7. Create a new node with data
8. Make new node point to first node
9. Make head node point to new node
10. If choice = 2 then
11. Make temp node point to first node
12. Make head node point to next of temp node
13. Release memory
14. If choice = 3 then
15. Display stack elements starting from head node till null
16. Stop

Program

```
/* 3c - Queue using Single Linked List */
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <alloc.h>
struct node
{
        int label;
        struct node *next;
};
main()
{
        int ch=0;
        int k;
        struct node *h, *temp, *head;
        /* Head node construction */
        head = (struct node*) malloc(sizeof(struct node));
        head->next = NULL;
        while(1)
        {
                printf("\n Queue using Linked List \n");
                printf("1->Insert ");
```

```
                printf("2->Delete ");
                printf("3->View ");
                printf("4->Exit \n");
                printf("Enter your choice : ");
                scanf("%d", &ch);
                switch(ch)
                {
                case 1:
                        /* Create a new node */
                        temp=(struct node *)(malloc(sizeof(struct node)));
                        printf("Enter label for new node : ");
                        scanf("%d", &temp->label);
                        /* Reorganize the links */
                        h = head;
                        while (h->next != NULL)
                                h = h->next;
                        h->next = temp;
                        temp->next = NULL;
                        break;

                case 2:
                        /* Delink the first node */
                        h = head->next;
                        head->next = h->next;
                        printf("Node deleted \n");
                        free(h);
                        break;
                case 3:
                        printf("\n\nHEAD -> ");
                        h=head;
                        while (h->next!=NULL)
                        {
                                h = h->next;
                                printf("%d -> ",h->label);
                        }
                        printf("NULL \n");
                        break;
                case 4:
                        exit(0);
                }
        }
}
```

Output

Queue using Linked List
1->Insert 2->Delete 3->View 4->Exit
Enter your choice : 1

```
Enter label for new node : 12
Queue using Linked List
1->Insert 2->Delete 3->View 4->Exit
Enter your choice : 1
Enter label for new node : 23
Queue using Linked List
1->Insert 2->Delete 3->View 4->Exit
Enter your choice : 3
HEAD -> 12 -> 23 -> NULL
```

Result: Thus push and pop operations of a stack was demonstrated using linked list.

Ex. No. 4a **APPLICATIONS OF LIST**
POLYNOMIAL ADDITION AND SUBTRACTION

Date:

Aim: To store a polynomial using linked list. Also, perform addition and subtraction on two polynomials.

Algorithm

Let p and q be the two polynomials represented by linked lists
1. while p and q are not null, repeat step 2.
2. If powers of the two terms are equal then
    If the terms do not cancel then
        Insert the sum of the terms into the sum Polynomial
        Advance p Advance q
    Else if the power of the first polynomial> power of second Then
        Insert the term from first polynomial into sum polynomial
        Advance p
    Else
        insert the term from second polynomial into sum polynomial Advance q
3. Copy the remaining terms from the non empty polynomial into the sum polynomial.
The third step of the algorithm is to be processed till the end of the polynomials has not been reached.

Program

```
/* 4a – Polynomial Addition and Subtraction */

#include <stdio.h>

#include <conio.h>

#include <malloc.h>

struct node

{

    int num;

 int coeff;

struct node *next;

};

struct node *start1 = NULL;

struct node *start2 = NULL;

struct node *start3 = NULL;

struct node *start4 = NULL;
```

```
struct node *last3 = NULL;
struct node *create_poly(struct node *);
struct node *display_poly(struct node *);
struct node *add_poly(struct node *, struct node *, struct node *);
struct node *sub_poly(struct node *, struct node *, struct node *);
struct node *add_node(struct node *, int, int);
int main()
{
   int option;
  clrscr();
  do
  {
printf("\n******* MAIN MENU *******");
printf("\n 1. Enter the first polynomial");
printf("\n 2. Display the first polynomial");
printf("\n 3. Enter the second polynomial");
printf("\n 4. Display the second polynomial");
printf("\n 5. Add the polynomials");
printf("\n 6. Display the result");
printf("\n 7. Subtract the polynomials");
printf("\n 8. Display the result");
printf("\n 9. EXIT");
printf("\n\n Enter your option : ");
scanf("%d", &option);
switch(option)
 {
   case 1:
start1 = create_poly(start1);
break;
```

```
   case 2:
start1 = display_poly(start1);
break;
   case 3:
start2 = create_poly(start2);
break;
   case 4:
start2 = display_poly(start2);
break;
   case 5:
start3 = add_poly(start1, start2, start3);
break;
   case 6:
start3 = display_poly(start3);
break;
   case 7:
start4 = sub_poly(start1, start2, start4);
break;
   case 8:
start4 = display_poly(start4);
break;
            }
  }while(option!=9);
  getch();
  return 0;
}

struct node *create_poly(struct node *start)
{
```

```
struct node *new_node, *ptr;
int n, c;
printf("\n Enter the number : ");
scanf("%d", &n);
printf("\t Enter its coeffient : ");
 scanf("%d", &c);
while(n != -1)
{
 if(start==NULL)
{
new_node = (struct node *)malloc(sizeof(struct node));
 new_node ->num = n;
new_node ->coeff = c;
new_node ->next = NULL;
start = new_node;
}
else
{
 ptr = start;
while(ptr ->next != NULL) ptr = ptr ->next;
new_node = (struct node *)malloc(sizeof(struct node));
new_node ->num = n;
new_node ->coeff = c;
new_node ->next = NULL;
ptr ->next = new_node;
}
printf("\n Enter the number : ");
scanf("%d", &n);
if(n ==-1)
```

```
 break;
 printf("\t Enter its coef?cient : ");
 scanf("%d", &c);
} return start;
}

struct node *display_poly(struct node *start)
{
struct node *ptr;
ptr = start;
while(ptr != NULL)
{
 printf("\n%d x %d\t", ptr ->num, ptr ->coeff);
ptr = ptr ->next;
}
 return start;
}
struct node *add_poly(struct node *start1, struct node *start2, struct node *start3)
{
struct node *ptr1, *ptr2;
int sum_num, c;
ptr1 = start1, ptr2 = start2;
 while(ptr1 != NULL && ptr2 != NULL)
{
 if(ptr1 ->coeff == ptr2 ->coeff)
{
sum_num = ptr1 ->num + ptr2 ->num;
start3 = add_node(start3, sum_num, ptr1 ->coeff);
ptr1 = ptr1 ->next; ptr2 = ptr2 ->next;
```

```
}
else if(ptr1 ->coeff > ptr2 ->coeff)
{
 start3 = add_node(start3, ptr1 ->num, ptr1 ->coeff);
ptr1 = ptr1 ->next;
}
else if(ptr1 ->coeff < ptr2 ->coeff)
{
start3 = add_node(start3, ptr2 ->num, ptr2 ->coeff);
ptr2 = ptr2 ->next;
}
 }
 if(ptr1 == NULL)
{
 while(ptr2 != NULL)
{
 start3 = add_node(start3, ptr2 ->num, ptr2 ->coeff);
ptr2 = ptr2 ->next;
}
 }
 if(ptr2 == NULL)
{
 while(ptr1 != NULL)
{
        start3 = add_node(start3, ptr1 ->num, ptr1 ->coeff);
ptr1 = ptr1 ->next;
}
 }
 return start3;
```

```
}

struct node *sub_poly(struct node *start1, struct node *start2, struct node *start4)
{
 struct node *ptr1, *ptr2;
int sub_num, c;
 ptr1 = start1, ptr2 = start2;
do
{
 if(ptr1 ->coeff == ptr2 ->coeff)
{
 sub_num = ptr1 ->num - ptr2 ->num;
start4 = add_node(start4, sub_num, ptr1 ->coeff);
ptr1 = ptr1 ->next; ptr2 = ptr2 ->next;
 }
 else if(ptr1 ->coeff > ptr2 ->coeff)
{
 start4 = add_node(start4, ptr1 ->num, ptr1 ->coeff);
ptr1 = ptr1 ->next;
}
 else if(ptr1 ->coeff < ptr2 ->coeff)
{
start4 = add_node(start4, ptr2 ->num, ptr2 ->coeff);
ptr2 = ptr2 ->next;
}
 }while(ptr1 != NULL || ptr2 != NULL);
if(ptr1 == NULL)
{
 while(ptr2 != NULL)
```

```
{
 start4 = add_node(start4, ptr2 ->num, ptr2 ->coeff);
ptr2 = ptr2 ->next;
 }
}
if(ptr2 == NULL)
{
 while(ptr1 != NULL)
 {
 start4 = add_node(start4, ptr1 ->num, ptr1 ->coeff);
 ptr1 = ptr1 ->next;
}
 }
 return start4;
}

struct node *add_node(struct node *start, int n, int c)
{
 struct node *ptr, *new_node;
if(start == NULL)
 {
 new_node = (struct node *)malloc(sizeof(struct node));
 new_node ->num = n;
new_node ->coeff = c;
new_node ->next = NULL;
start = new_node;
}
else
{
```

```
 ptr = start;
while(ptr ->next != NULL)
ptr = ptr ->next;
 new_node = (struct node *)malloc(sizeof(struct node));
new_node ->num = n;
 new_node ->coeff = c;
new_node ->next = NULL;
ptr ->next = new_node;
}
 return start;
}
```

Output

******* MAIN MENU *******
1. Enter the 1rst polynomial
2. Display the 1rst polynomial

_____

9. EXIT
Enter your option : 1
Enter the number : 6
Enter its coef□cient : 2
Enter the number : 5
Enter its coef□cient : 1
Enter the number : –1
Enter your option : 2
6 x 2 5 x 1
Enter your option : 9

Ex. No. 4b **INFIX TO POSTFIX**
Date:

Aim: To convert infix expression to its postfix form using stack operations.

Algorithm

1. Start
2. Define a array *stack* of size *max* = 20
3. Initialize *top* = -1
4. Read the infix expression character-by-character
5. If character is an operand print it
6. If character is an operator
7. Compare the operator's priority with the stack[top] operator.
8. If the stack [top] operator has higher or equal priority than the inputoperator, Pop it from the stack and print it.
9. Else
10. Push the input operator onto the stack
11. If character is a left parenthesis, then push it onto the stack.
12. If the character is a right parenthesis, pop all the operators from the stack andprint it until a left parenthesis is encountered. Do not print the parenthesis.

Program

```
/* 4b - Conversion of infix to postfix expression */
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define MAX 20
int top = -1;
char stack[MAX];
char pop();
void push(char item);
int prcd(char symbol)
{
   switch(symbol)
   {
        case '+':
        case '-':
                return 2;
                break;
        case '*':
        case '/':
                return 4;
                break;
        case '^':
        case '$':
                return 6;
                break;
```

```
        case '(':
        case ')':
        case '#':
                return 1;
                break;
    }
}
int isoperator(char symbol)
{
     switch(symbol)
     {
        case '+':
        case '-':
        case '*':
        case '/':
        case '^':
        case '$':
        case '(':
        case ')':
                return 1;
                  break;
        default:
                return 0;
     }
}

void convertip(char infix[],char postfix[])
{
        int i,symbol,j = 0;
        stack[++top] = '#';
        for(i=0;i<strlen(infix);i++)
        {
                symbol = infix[i];
                if(isoperator(symbol) == 0)
                {
                        postfix[j] = symbol;
                        j++;
                }
                else
                {
                        if(symbol == '(')
                                push(symbol);
                        else if(symbol == ')')
                        {
                                while(stack[top] != '(')
                                {
                                        postfix[j] = pop();
```

```
                                        j++;
                                }
                                pop(); //pop out (.
                        }
                        else
                        {
                                if(prcd(symbol) > prcd(stack[top]))
                                        push(symbol);
                                else
                                {
                                        while(prcd(symbol) <= prcd(stack[top]))
                                        {
                                        postfix[j] = pop();
                                        j++;
                                        }
                                        push(symbol);
                                }
                        }
                }
        }

        while(stack[top] != '#')
        {
                postfix[j] = pop();
                j++;
        }
        postfix[j] = '\0';
}
main()
{
   char infix[20],postfix[20];
   clrscr();
   printf("Enter the valid infix string: ");
   gets(infix);
   convertip(infix, postfix);
   printf("The corresponding postfix string is: ");
   puts(postfix);
   getch();
}
void push(char item)
{
        top++;
        stack[top] = item;
}

char pop()
{
```

```
        char a;
        a = stack[top];
        top--;
        return a;
}
```

Output

```
Enter the valid infix string: (a+b*c)/(d$e)
The corresponding postfix string is: abc*+de$/
Enter the valid infix string: a*b+c*d/e
The corresponding postfix string is: ab*cd*e/+
Enter the valid infix string: a+b*c+(d*e+f)*g
The corresponding postfix string is: abc*+de*f+g*+
```

Result: Thus the given infix expression was converted into postfix form using stack.

Ex. No. 4c EXPRESSION EVALUATION

Date:

Aim: To evaluate the given postfix expression using stack operations.

Algorithm

1. Start
2. Define a array *stack* of size *max* = 20
3. Initialize *top* = -1
4. Read the postfix expression character-by-character
5. If character is an operand push it onto the stack
6. If character is an operator
7. Pop topmost two elements from stack.
8. Apply operator on the elements and push the result onto the stack, Eventually only result will be in the stack at end of the expression.
9. Pop the result and print it.

Program

```
/* 5f - Evaluation of Postfix expression using stack */
#include <stdio.h>
#include <conio.h>
struct stack
{
        int top;
        float a[50];
}s;
main()
{
        char pf[50];
        float d1,d2,d3;
        int i;
        clrscr();
        s.top = -1;
        printf("\n\n Enter the postfix expression: ");
        gets(pf);
        for(i=0; pf[i]!='\0'; i++)
        {
           switch(pf[i])
           {
                case '0':
                case '1':
                case '2':
                case '3':
                case '4':
                case '5':
                case '6':
```

```
                case '7':
                case '8':
                case '9':
                        s.a[++s.top] = pf[i]-'0';
                        break;
                case '+':
                        d1 = s.a[s.top--];
                        d2 = s.a[s.top--];
                        s.a[++s.top] = d1 + d2;
                        break;
                case '-':
                        d2 = s.a[s.top--];
                        d1 = s.a[s.top--];
                        s.a[++s.top] = d1 - d2;
                        break;
                case '*':
                        d2 = s.a[s.top--];
                        d1 = s.a[s.top--];
                        s.a[++s.top] = d1*d2;
                        break;
                case '/':
                        d2 = s.a[s.top--];
                        d1 = s.a[s.top--];
                        s.a[++s.top] = d1 / d2;
                        break;
            }
        }
        printf("\n Expression value is %5.2f", s.a[s.top]);
        getch();
}
```

Output

Enter the postfix expression: 6523+8*+3+*
Expression value is 288.00

Result

Thus the given postfix expression was evaluated using stack.

Ex. No. 5 IMPLEMENTATION OF BINARY TREES AND OPERATIONS OF BINARY TREES

Date:

Aim: To implement Binary Tree and perform inorder, preorder and postorder traversals.

Algorithm

Implementation Algorithm

1. A new node is created
2. Count is incremented by 1
3. If the count is even number element in hand is inserted as left child of the current parent else the element is inserted as right child of the current parent.

Traversal Algorithm:

Preorder :

1) Traverse the root.
2) Traverse the left subtree in preorder.
3) Traverse the right subtree in preorder.

Inorder:

1) Traverse the left subtree in inorder.
2) Traverse the root.
3) Traverse the right subtree in inorder.

Postorder:

1) Traverse the left subtree in postorder.
2) Traverse the right subtree in postorder.
3) Traverse the root.

Program

```
/* 5 – Binary Tree implementation and operations */

#include<stdio.h>

int count=1;

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

void insert(node**bt,int n)
{
 if(*bt==NULL)
```

```
 {
  *bt=(node*)malloc(sizeof(node));
  (*bt)->data=n;
  (*bt)->left=NULL;
  (*bt)->right=NULL;
  count++;
 }
 else
 {
 if(count%2==0)
 insert(&((*bt)->left),n);
 else
 insert(&((*bt)->right),n);
 }
}

//traverse the tree in inorder
void inorder(node*bt)
{
 if(bt!=NULL)
 {
  inorder(bt->left);
  printf("%d\t",bt->data);
  inorder(bt->right);
 }
}

//traverse the tree in preorder
void preorder(node*bt)
{
 if(bt!=NULL)
 {
  printf("%d\t",bt->data);
  preorder(bt->left);
  preorder(bt->right);
 }
}

//traverse the tree in postorder
void postorder(node*bt)
{
 if(bt!=NULL)
 {
  postorder(bt->left);
  postorder(bt->right);
  printf("%d\t",bt->data);

 }
}
```

```
void main()
{
 node*bt=NULL;
 insert(&bt,1);
 insert(&bt,2);
 insert(&bt,3);
 insert(&bt,4);
 insert(&bt,5);
 insert(&bt,6);
 inorder(bt);
 printf("\n");
 preorder(bt);
 printf("\n");
 postorder(bt);
}
```

Ex No. 6 IMPLEMENTATION BINARY SEARCH TREE
Date:

Aim: To construct a binary search tree and perform search.

Algorithm

1. Start
2. Call insert to insert an element into binary search tree.
3. Call delete to delete an element from binary search tree.
4. Call findmax to find the element with maximum value in binary search tree
5. Call findmin to find the element with minimum value in binary search tree
6. Call find to check the presence of the element in the binary search tree
7. Call display to display the elements of the binary search tree
8. Call makeempty to delete the entire tree.
9. Stop

Insert function

1. Get the element to be inserted.
2. Find the position in the tree where the element to be inserted by checking the elements in the tree by traversing from the root.
3. If the element to be inserted is less than the element in the current node in the tree then traverse left subtree
4. If the element to be inserted is greater than the element in the current node in the tree then traverse right subtree
5. Insert the element if there is no further move

Delete function

1. Get the element to be deleted.
2. Find the node in the tree that contain the element.
3. Delete the node an rearrange the left and right siblings if any present for the deleted node

Findmax function

1. Traverse the tree from the root.
2. Find the rightmost leaf node of the entire tree and return it
3. If the tree is empty return null.

Findmin function

1. Traverse the tree from the root.
2. Find the leftmost leaf node of the entire tree and return it
3. If the tree is empty return null.

Find function

1. Traverse the tree from the root.
2. Check whether the element to searched matches with element of the current node. If match occurs return it.
3. Otherwise if the element is less than that of the element of the current node then search the leaf subtree
4. Else search right subtree.

Makeempty function

Make the root of the tree to point to null.

Program

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
struct searchtree
{
    int element;
    struct searchtree *left,*right;
}*root;
typedef struct searchtree *node;
typedef int ElementType;
node insert(ElementType, node);
node delete(ElementType, node);
void makeempty();
node findmin(node);
node findmax(node);
node find(ElementType, node);
void display(node, int);
void main()
{
    int ch;
    ElementType a;
    node temp;
    makeempty();
    while(1)
    {
        printf("\n1. Insert\n2. Delete\n3. Find\n4. Find min\n5. Find max\n6.Display\n7.
        Exit\nEnter Your Choice : ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                printf("Enter an element : ");
                scanf("%d", &a);
                root = insert(a, root);
                break;
            case 2:
                printf("\nEnter the element to delete : ");
                scanf("%d",&a);
                root = delete(a, root);
                break;
```

```
            case 3:
                printf("\nEnter the element to search : ");
                scanf("%d",&a);
                temp = find(a, root);
                if (temp != NULL)
                printf("Element found");
                else
                printf("Element not found");
                break;
            case 4:
                temp = findmin(root);
                if(temp==NULL)
                printf("\nEmpty tree");
                else
                printf("\nMinimum element : %d", temp->element);
                break;
            case 5:
                temp = findmax(root);
                if(temp==NULL)
                printf("\nEmpty tree");
                else
                printf("\nMaximum element : %d", temp->element);
                break;
            case 6:
                if(root==NULL)printf("\nEmpty tree");
                else
                display(root, 1);
                break;
            case 7:
            exit(0);
            default:
            printf("Invalid Choice");
        }
    }
}

node insert(ElementType x,node t)
{
    if(t==NULL)
    {
        t = (node)malloc(sizeof(node));
        t->element = x;
        t->left = t->right = NULL;
    }
    else
    {
        if(x < t->element)
```

```
        t->left = insert(x, t->left);
        else
        if(x > t->element)t->right = insert(x, t->right);
    }
    return t;
}

node delete(ElementType x,node t)
{
      node temp;
      if(t == NULL)
      printf("\nElement not found");
      else
      {
            if(x < t->element)
            t->left = delete(x, t->left);
            else if(x > t->element)
            t->right = delete(x, t->right);
            else
            {
            if(t->left && t->right)
            {
            temp = findmin(t->right);
            t->element = temp->element;
            t->right = delete(t->element,t->right);
            }
            else if(t->left == NULL)
            {
            temp = t;
            t=t->right;
            free (temp);
            }
            else
            {
                        temp = t;
                        t=t->left;
                        free (temp);
            }
      }
      }
      return t;
}

void makeempty()
{
    root = NULL;
}
```

```
node findmin(node temp)
{
     if(temp == NULL || temp->left == NULL)
               return temp;
     return findmin(temp->left);
}

node findmax(node temp)
{
     if(temp==NULL || temp->right==NULL)
               return temp;
     return findmax(temp->right);
}

node find(ElementType x, node t)
{
     if(t==NULL)
                return NULL;
     if(x<t->element)
               return find(x,t->left);
     if(x>t->element)
      return find(x,t->right);
return t;
}

void display(node t,int level)
{
     int i;
     if(t)
     {
     display(t->right, level+1);
      printf(" \n");
      for(i=0;i<level;i++)
     printf(" ");
     printf("%d", t->element);
     display(t->left, level+1);
     }
}
```

Sample Output:

1. Insert
2. Delete
3. Find
4. Find Min
5. Find Max

```
6. Display
7. Exit

Enter your Choice : 1
Enter an element : 10

1. Insert
2. Delete
3. Find
4. Find Min
5. Find Max
6. Display
7. Exit

Enter your Choice : 1
Enter an element : 20
1. Insert
2. Delete
3. Find
4. Find Min
5. Find Max
6. Display
7. Exit

Enter your Choice : 1
Enter an element : 5
1. Insert
2. Delete
3. Find
4. Find Min
5. Find Max
6. Display
7. Exit
Enter your Choice : 4
The smallest Number is 5

1. Insert
2. Delete
3. Find
4. Find Min
5. Find Max
6. Display
7. Exit
Enter your Choice : 3
Enter an element : 100
Element not Found
```

1. Insert
2. Delete
3. Find
4. Find Min
5. Find Max
6. Display
7. Exit

Enter your Choice : 2
Enter an element : 20

1. Insert
2. Delete
3. Find
4. Find Min
5. Find Max
6. Display
7. Exit

Enter your Choice : 6
        20
10

1. Insert
2. Delete
3. Find
4. Find Min
5. Find Max
6. Display
7. Exit
Enter your Choice : 7

Result:
The program for binary search tree is executed and verified.

Ex. No. 7 **IMPLEMENTATION OF AVL TREES**
Date:

Aim: To implement AVL Trees.

Algorithm

A binary search tree x is called an AVL tree,
if:
1. b(x.key) ∈ {−1, 0, 1}, and
2. x.leftChild and x.rightChild are both AVL trees. = the height balance of every node must be -1, 0, or 1

insert/delete via standard algorithms
• after insert/delete: load balance b(node) might be changed to +2 or −2 for certain nodes
• re-balance load after each step

Insert operation may cause balance factor to become 2 or –2 for some node
› only nodes on the path from insertion point to root node have possibly changed in height
› So after the Insert, go back up to the root node by node, updating heights
› If a new balance factor (the difference hleft – hright) is 2 or –2, adjust tree by rotation around the node
Let the node that needs rebalancing be α.
There are 4 cases:
Outside Cases (require single rotation) :
1. Insertion into left subtree of left child of α.
2. Insertion into right subtree of right child of α.
Inside Cases (require double rotation) :
3. Insertion into right subtree of left child of α.
4. Insertion into left subtree of right child of α.
The rebalancing is performed through four separate rotation algorithms.
You can either keep the height or just the difference in height, i.e. the balance factor; this has to be modified on the path of insertion even if you don't perform rotations
Once you have performed a rotation (single or double) you won't need to go back up the tree
SINGLE ROTATION

```
RotateFromRight(n : reference node pointer)
{
p : node pointer;
p := n.right;
n.right := p.left;
p.left := n;
n := p
}
```

You also need to modify the heights or balance factors of n and p

DOUBLE ROTATION

```
DoubleRotateFromRight(n : reference node pointer)
```

```
 {
RotateFromLeft(n.right);
RotateFromRight(n);
}
```

INSERTION IN AVL TREES

```
Insert(T : tree pointer, x : element) :
{
if T = null then
T := new tree; T.data := x; height := 0;
case
T.data = x : return ; //Duplicate do nothing
T.data > x : return Insert(T.left, x);
if ((height(T.left)- height(T.right)) = 2){
if (T.left.data > x ) then //outside case
T = RotatefromLeft (T);
else //inside case
T = DoubleRotatefromLeft (T);}
T.data < x : return Insert(T.right, x);
code similar to the left case
Endcase
T.height := max(height(T.left),height(T.right)) +1;
return;
}
```

DELETION

Similar but more complex than insertion

› Rotations and double rotations needed to rebalance

› Imbalance may propagate upward so that many rotations may be needed.

Program

```
// C program to insert a node in AVL tree
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct Node
{
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

// A utility function to get maximum of two integers
```

```
int max(int a, int b);

// A utility function to get the height of the tree
int height(struct Node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and
    NULL left and right pointers. */
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; // new node is initially added at leaf
    return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct Node *rightRotate(struct Node *y)
{
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;

    // Return new root
    return x;
}
```

```
// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct Node *leftRotate(struct Node *x)
{
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(struct Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

// Recursive function to insert a key in the subtree rooted
// with node and returns the new root of the subtree.
struct Node* insert(struct Node* node, int key)
{
    /* 1. Perform the normal BST insertion */
    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 // Equal keys are not allowed in BST
        return node;

    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left),
                           height(node->right));
```

```
    /* 3. Get the balance factor of this ancestor
          node to check whether this node became
          unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then
    // there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

// A utility function to print preorder traversal
// of the tree.
// The function also prints height of every node
void preOrder(struct Node *root)
{
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

```
/* Drier program to test above function*/
int main()
{
struct Node *root = NULL;

/* Constructing tree given in the above figure */
root = insert(root, 10);
root = insert(root, 20);
root = insert(root, 30);
root = insert(root, 40);
root = insert(root, 50);
root = insert(root, 25);

/* The constructed AVL Tree would be
            30
            /\
           20 40
          / \   \
         10 25   50

*/

printf("Preorder traversal of the constructed AVL"
            " tree is \n");
preOrder(root);
return 0;
}
```

Result:

Thus the AVL tree is implemented.

Ex. No. 8 **IMPLEMENTATION OF HEAP USING PRIORITY QUEUES**
Date:

Aim

To Implementation Of Heap Using Priority Queues.

Algorithm

- Priority queue is a type of queue in which every element has a key associated to it and the queue returns the element according to these keys, unlike the traditional queue which works on first come first serve basis.

- Thus, a max-priority queue returns the element with maximum key first whereas, a min-priority queue returns the element with the smallest key first.

  max-priority queue and min-priority queue

- Priority queues are used in many algorithms like Huffman Codes, Prim's algorithm, etc. It is also used in scheduling processes for a computer, etc.

- Heaps are great for implementing a priority queue because of the largest and smallest element at the root of the tree for a max-heap and a min-heap respectively. We use a max-heap for a max-priority queue and a min-heap for a min-priority queue.

There are mainly 4 operations we want from a priority queue:

1. **Insert** → To insert a new element in the queue.

2. **Maximum/Minimum** → To get the maximum and the minimum element from the max-priority queue and min-priority queue respectively.

3. **Extract Maximum/Minimum** → To remove and return the maximum and the minimum element from the max-priority queue and min-priority queue respectively.

4. **Increase/Decrease key** → To increase or decrease key of any element in the queue.

- A priority queue stores its data in a specific order according to the keys of the elements. So, inserting a new data must go in a place according to the specified order. This is what the insert operation does.
- The entire point of the priority queue is to get the data according to the key of the data and the Maximum/Minimum and Extract Maximum/Minimum does this for us.
- We know that the maximum (or minimum) element of a priority queue is at the root of the max-heap (or min-heap). So, we just need to return the element at the root of the heap.

- This is like the pop of a queue, we return the element as well as **delete** it from the heap. So, we have to return and delete the root of a heap. Firstly, we store the value of the root in a variable to return it later from the function and then we just make the root equal to the last element of the heap. Now the root is equal to the last element of the heap, we delete the last element easily by reducing the size of the heap by 1.

- Doing this, we have disturbed the heap property of the root but we have not touched any of its children, so they are still heaps. So, we can call *Heapify* on the root to make the tree a heap again.

```
EXTRACT-MAXIMUM(A)

 max = A[1] // stroing maximum value

 A[1] = A[heap_size] // making root equal to the last element

 heap_size = heap_size-1 // delete last element

 MAX-HEAPIFY(A, 1) // root is not following max-heap property

 return max //returning the maximum value
```

Program

```
#include <stdio.h>

int tree_array_size = 20;
int heap_size = 0;
const int INF = 100000;

void swap( int *a, int *b )
 {
  int t;
  t = *a;
  *a = *b;
  *b = t;
}

//function to get right child of a node of a tree
int get_right_child(int A[], int index)
 {
  if((((2*index)+1) < tree_array_size) && (index >= 1))
    return (2*index)+1;
  return -1;
}
```

```
//function to get left child of a node of a tree
int get_left_child(int A[], int index)
 {
   if(((2*index) < tree_array_size) && (index >= 1))
      return 2*index;
   return -1;
}

//function to get the parent of a node of a tree
int get_parent(int A[], int index)
{
  if ((index > 1) && (index < tree_array_size))
 {
   return index/2;
  }
  return -1;
}

void max_heapify(int A[], int index)
 {
  int left_child_index = get_left_child(A, index);
  int right_child_index = get_right_child(A, index);

  // finding largest among index, left child and right child
  int largest = index;

  if ((left_child_index <= heap_size) && (left_child_index>0))
 {
   if (A[left_child_index] > A[largest]) {
     largest = left_child_index;
   }
  }

  if ((right_child_index <= heap_size && (right_child_index>0)))
{
   if (A[right_child_index] > A[largest])
{
     largest = right_child_index;
   }
  }

  // largest is not the node, node is not a heap
  if (largest != index)
{
   swap(&A[index], &A[largest]);
   max_heapify(A, largest);
  }
```

```
}

void build_max_heap(int A[])
{
  int i;
  for(i=heap_size/2; i>=1; i--)
{
    max_heapify(A, i);
  }
}

int maximum(int A[])
{
  return A[1];
}

int extract_max(int A[])
{
  int maxm = A[1];
  A[1] = A[heap_size];
  heap_size--;
  max_heapify(A, 1);
  return maxm;
}

void increase_key(int A[], int index, int key)
 {
  A[index] = key;
  while((index>1) && (A[get_parent(A, index)] < A[index]))
  {
    swap(&A[index], &A[get_parent(A, index)]);
    index = get_parent(A, index);
  }
}

void decrease_key(int A[], int index, int key)
 {
  A[index] = key;
  max_heapify(A, index);
}

void insert(int A[], int key)
 {
  heap_size++;
  A[heap_size] = -1*INF;
  increase_key(A, heap_size, key);
}
```

```
void print_heap(int A[])
{
  int i;
  for(i=1; i<=heap_size; i++)
 {
    printf("%d\n",A[i]);
  }
  printf("\n");
}

int main()
 {
  int A[tree_array_size];
  insert(A, 20);
  insert(A, 15);
  insert(A, 8);
  insert(A, 10);
  insert(A, 5);
  insert(A, 7);
  insert(A, 6);
  insert(A, 2);
  insert(A, 9);
  insert(A, 1);

  print_heap(A);

  increase_key(A, 5, 22);
  print_heap(A);

  decrease_key(A, 1, 13);
  print_heap(A);

  printf("%d\n\n", maximum(A));
  printf("%d\n\n", extract_max(A));

  print_heap(A);

  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
```

```
  return 0;
}
```

Result:

Thus the Of Heap Using Priority Queues is Implemented.

Ex. No. 9a **REPRESENTATION OF GRAPH**
Date:

Aim To represent graph using adjacency list.

Algorithm

For a graph with |V| vertices, an **adjacency matrix** is a |V|×|V| matrix of 0s and 1s, where the entry in row i and column j is 1 if and only if the edge (i,j) in the graph.

1. Consider the graph to be represented
2. Start from an edge
3. If the edge connects the vertices i,j the mark the i^{th} row and j^{th} column of the adjacency matrix as 1 otherwise store 0
4. Finally print the adjacency matrix

Program

```
*
 * Program  : Adjacency Matrix Representation of Graph
 * Language : C
 */
#include<stdio.h>
#define V 5

//init matrix to 0
void init(int arr[][V])
{
   int i,j;
   for(i = 0; i < V; i++)
      for(j = 0; j < V; j++)
         arr[i][j] = 0;
}

//Add edge. set arr[src][dest] = 1
void addEdge(int arr[][V],int src, int dest)
{
    arr[src][dest] = 1;
}

void printAdjMatrix(int arr[][V])

{
    int i, j;
```

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

//print the adjMatrix
int main()
{
    int adjMatrix[V][V];

    init(adjMatrix);
    addEdge(adjMatrix,0,1);
    addEdge(adjMatrix,0,2);
    addEdge(adjMatrix,0,3);
    addEdge(adjMatrix,1,3);
    addEdge(adjMatrix,1,4);
    addEdge(adjMatrix,2,3);
    addEdge(adjMatrix,3,4);

    printAdjMatrix(adjMatrix);

    return 0;
}
```

OUTPUT

```
0 1 1 1 0
0 0 0 1 1
0 0 0 1 0
0 0 0 0 1
0 0 0 0 0
```

Ex. No. 9b **GRAPH TRAVERSAL-BREADTH FIRST TRAVERSAL**
Date:

Aim To implement breadth first graph traversal.

Algorithm

```
BFS (G, s)                //Where G is the graph and s is the source node
    let Q be queue.
    Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

    mark s as visited.
    while ( Q is not empty)
        //Removing that vertex from queue,whose neighbour will be visited now
        v  =  Q.dequeue( )

      //processing all the neighbours of v
      for all neighbours w of v in Graph G
          if w is not visited
                Q.enqueue( w )          //Stores w in Q to further visit its neighbour
                mark w as visited.
```

Program

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

#define MAX 5

struct Vertex {
   char label;
   bool visited;
};

//queue variables

int queue[MAX];
int rear = -1;
int front = 0;
int queueItemCount = 0;

//graph variables

//array of vertices
struct Vertex* lstVertices[MAX];

//adjacency matrix
```

```
int adjMatrix[MAX][MAX];

//vertex count
int vertexCount = 0;

//queue functions

void insert(int data) {
   queue[++rear] = data;
   queueItemCount++;
}

int removeData() {
   queueItemCount--;
   return queue[front++];
}

bool isQueueEmpty() {
   return queueItemCount == 0;
}

//graph functions

//add vertex to the vertex list
void addVertex(char label) {
   struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
   vertex->label = label;
   vertex->visited = false;
   lstVertices[vertexCount++] = vertex;
}

//add edge to edge array
void addEdge(int start,int end) {
   adjMatrix[start][end] = 1;
   adjMatrix[end][start] = 1;
}

//display the vertex
void displayVertex(int vertexIndex) {
   printf("%c ",lstVertices[vertexIndex]->label);
}

//get the adjacent unvisited vertex
int getAdjUnvisitedVertex(int vertexIndex) {
   int i;

   for(i = 0; i<vertexCount; i++) {
```

```
    if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false)
      return i;
  }

  return -1;
}

void breadthFirstSearch() {
  int i;

  //mark first node as visited
  lstVertices[0]->visited = true;

  //display the vertex
  displayVertex(0);

  //insert vertex index in queue
  insert(0);
  int unvisitedVertex;

  while(!isQueueEmpty()) {
    //get the unvisited vertex of vertex which is at front of the queue
    int tempVertex = removeData();

    //no adjacent vertex found
    while((unvisitedVertex = getAdjUnvisitedVertex(tempVertex)) != -1) {
      lstVertices[unvisitedVertex]->visited = true;
      displayVertex(unvisitedVertex);
      insert(unvisitedVertex);
    }

  }

  //queue is empty, search is complete, reset the visited flag
  for(i = 0;i<vertexCount;i++) {
    lstVertices[i]->visited = false;
  }
}

int main() {
  int i, j;

  for(i = 0; i<MAX; i++) // set adjacency {
    for(j = 0; j<MAX; j++) // matrix to 0
      adjMatrix[i][j] = 0;
  }
```

```
  addVertex('S');   // 0
  addVertex('A');   // 1
  addVertex('B');   // 2
  addVertex('C');   // 3
  addVertex('D');   // 4

  addEdge(0, 1);    // S - A
  addEdge(0, 2);    // S - B
  addEdge(0, 3);    // S - C
  addEdge(1, 4);    // A - D
  addEdge(2, 4);    // B - D
  addEdge(3, 4);    // C - D

  printf("\nBreadth First Search: ");

  breadthFirstSearch();

  return 0;
}
```

If we compile and run the above program, it will produce the following result −

Output

Breadth First Search: S A B C D

Ex. No. 9c **GRAPH TRAVERSAL-DEPTH FIRST TRAVERSAL**
Date:

Aim To implement Depth first graph traversal.

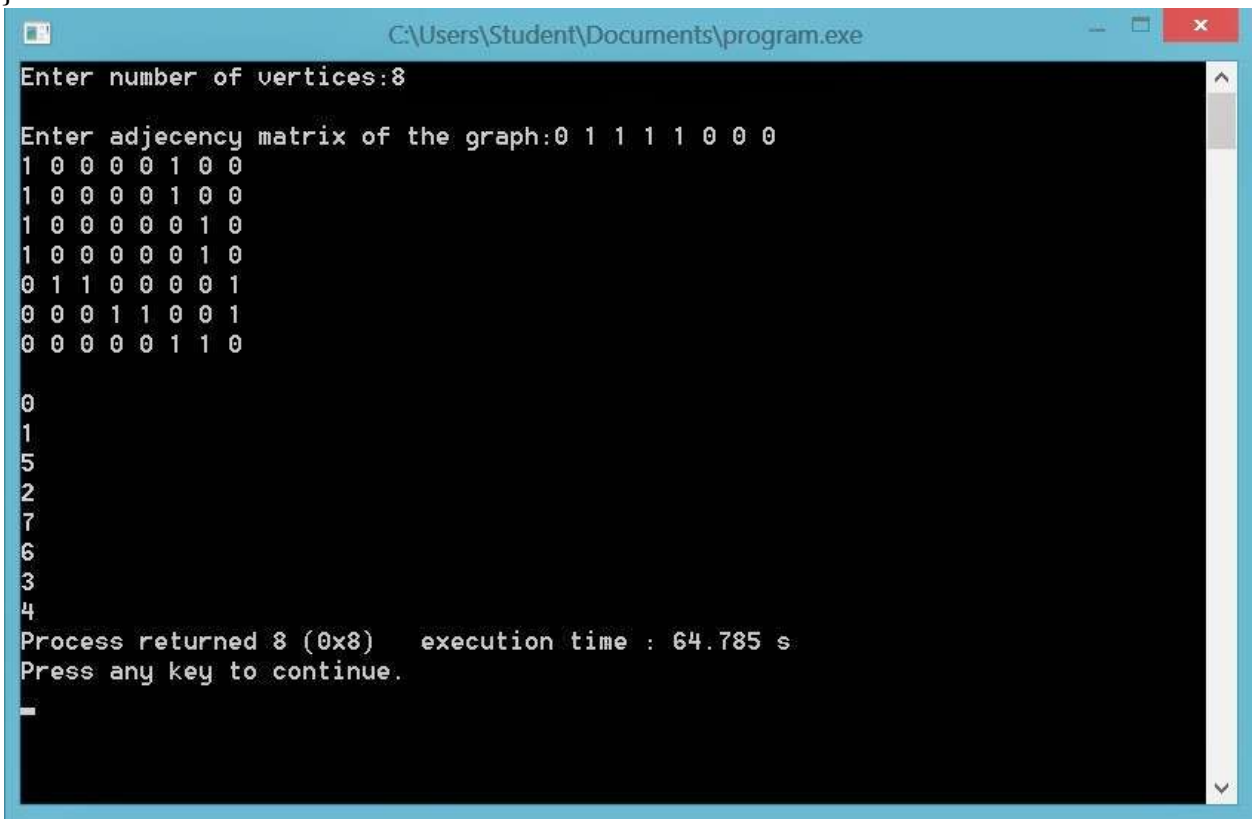
Algorithm

```
DFS-iterative (G, s):                          //Where G is graph and s is source vertex
    let S be stack
    S.push( s )          //Inserting s in stack
    mark s as visited.
    while ( S is not empty):
       //Pop a vertex from stack to visit next
       v  =  S.top( )
       S.pop( )
       //Push all the neighbours of v in stack that are not visited
       for all neighbours w of v in Graph G:
          if w is not visited :
                 S.push( w )
                mark w as visited


  DFS-recursive(G, s):
     mark s as visited
     for all neighbours w of s in Graph G:
        if w is not visited:
           DFS-recursive(G, w)
```

Program

```
#include<stdio.h>

void DFS(int);
int G[10][10],visited[10],n;    //n is no of vertices and graph is sorted in array G[10][10]

void main()
{
   int i,j;
   printf("Enter number of vertices:");

     scanf("%d",&n);

   //read the adjecency matrix
     printf("\nEnter adjecency matrix of the graph:");

     for(i=0;i<n;i++)
      for(j=0;j<n;j++)
                        scanf("%d",&G[i][j]);
```

```
   //visited is initialized to zero
  for(i=0;i<n;i++)
      visited[i]=0;

   DFS(0);
}

void DFS(int i)
{
   int j;
     printf("\n%d",i);
   visited[i]=1;

     for(j=0;j<n;j++)
      if(!visited[j]&&G[i][j]==1)
          DFS(j);
}
```



Ex. No. 10 **APPLICATION OF GRAPH**
Date:
Aim :

To perform the application of graph in data structures
Procedure:

- Create an array of linked lists (or arrays of arrays) of size V, where V is the number of vertices.
- For each edge in the graph, add the destination vertex to the list of the source vertex.
- If the graph is undirected, add the source vertex to the list of the destination vertex as well.
- If the graph is weighted, modify each element in the list to store a pair of values: the destination vertex and the weight of the edge.

Program

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

struct AdjListNode {
   int dest;
   struct AdjListNode* next;
};

struct Graph {
   int V;
   struct AdjListNode** array;
};

// Function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest) {
   struct AdjListNode* newNode = malloc(sizeof(struct AdjListNode));
   newNode->dest = dest;
   newNode->next = NULL;
   return newNode;
}

// Function to create a graph of V vertices
struct Graph* createGraph(int V) {
   struct Graph* graph = malloc(sizeof(struct Graph));
   graph->V = V;
   graph->array = calloc(V, sizeof(struct AdjListNode*));
   return graph;
}

// Function to add an edge to an undirected graph
```

```
void addEdge(struct Graph* graph, int src, int dest) {

    // Add an edge from src to dest
    struct AdjListNode* node = newAdjListNode(dest);
    node->next = graph->array[src];
    graph->array[src] = node;

    // Since the graph is undirected, add an edge from dest to src
    node = newAdjListNode(src);
    node->next = graph->array[dest];
    graph->array[dest] = node;
}

// Function to print the adjacency list
void printGraph(struct Graph* graph) {
    for (int i = 0; i < graph->V; i++) {
        printf("%d:", i);
        for (struct AdjListNode* cur = graph->array[i]; cur; cur = cur->next) {
            printf(" %d", cur->dest);
        }
        printf("\n");
    }
}

int main() {
    int V = 5;
    struct Graph* graph = createGraph(V);

    addEdge(graph, 0, 1);
    addEdge(graph, 0, 4);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);

    printf("Adjacency list representation:\n");
    printGraph(graph);

    return 0;
}
```

Result:

Thus the application of graph is performed .

Ex. No. 11 a **LINEAR SEARCH**
Date:

AimTo perform linear search of an element on the given array.

Algorithm

1. Start
2. Read number of array elements n
3. Read array elements Ai, i = 0,1,2,…n–1
4. Read search value
5. Assign 0 to found
6. Check each array element against search
7. If Ai = search then
   found = 1
   Print "Element found"
   Print position i
   Stop
8. If found = 0 then
   print "Element not found"

Stop

Program

```
/* Linear search on a sorted array */
#include <stdio.h>
#include <conio.h>
main()
{
    int a[50],i, n, val, found;
    clrscr();
    printf("Enter number of elements : ");
    scanf("%d", &n);
    printf("Enter Array Elements : \n");
    for(i=0; i<n; i++)
    scanf("%d", &a[i]);
    printf("Enter element to locate : ");
    scanf("%d", &val);
    found = 0;
    for(i=0; i<n; i++)
    {
        if (a[i] == val)
        {
            printf("Element found at position %d", i);
            found = 1;
            break;
        }
    }
```

```
    if (found == 0)
    printf("\n Element not found");
    getch();
}
```

Output

```
Enter number of elements : 7
Enter Array Elements :
23 6 12 5 0 32 10
Enter element to locate : 5
Element found at position 3
```

Result

Thus an array was linearly searched for an element's existence.

Ex. No. 11 b **BINARY SEARCH**
Date:

Aim
To locate an element in a sorted array using Binary search method

Algorithm
1. Start
2. Read number of array elements, say n
3. Create an array arr consisting n sorted elements
4. Get element, say key to be located
5. Assign 0 to lower and n to upper
6. While (lower < upper)
    a. Determine middle element mid = (upper+lower)/2
    b. If key = arr[mid] then
        i. Print mid
        ii. Stop
    c. Else if key > arr[mid] then
        i. lower = mid + 1
    d. else
        i. upper = mid – 1
7. Print "Element not found"
8. Stop

Program

```
/* Binary Search on a sorted array */
#include <stdio.h>
void main()
{
    int a[50],i, n, upper, lower, mid, val, found, att=0;
    printf("Enter array size : ");
    scanf("%d", &n);
    for(i=0; i<n; i++)
    a[i] = 2 * i;
    printf("\n Elements in Sorted Order \n");
    for(i=0; i<n; i++)
    printf("%4d", a[i]);
    printf("\n Enter element to locate : ");
    scanf("%d", &val);
    upper = n;
    lower = 0;
    found = -1;
    while (lower <= upper)
    {
        mid = (upper + lower)/2;
        att++;
        if (a[mid] == val)
```

```
        {
            printf("Found at index %d in %d attempts", mid, att);
            found = 1;
            break;
        }
        else if(a[mid] > val)
        upper = mid - 1;
        else
        lower = mid + 1;
    }
    if (found == -1)
    printf("Element not found");
}
```

Output

```
Enter array size : 10
Elements in Sorted Order
0 2 4 6 8 10 12 14 16 18
Enter element to locate : 16
Found at index 8 in 2 attempts
```

Result

Thus an element is located quickly using binary search method.

Ex. No. 11 c **INSERTION SORT**
Date:

Aim: To sort an array of N numbers using Insertion sort.

Algorithm

1. Start
2. Read number of array elements n
3. Read array elements Ai
4. Outer index i varies from second element to last element
5. Inner index j is used to compare elements to left of outer index
6. Insert the element into the appropriate position.
7. Display the array elements after each pass
8. Display the sorted array elements.
9. Stop

Program

```
/* 11a - Insertion Sort */
void main()
{
    int i, j, k, n, temp, a[20], p=0;
    printf("Enter total elements: ");
    scanf("%d",&n);
    printf("Enter array elements: ");
    for(i=0; i<n; i++)
    scanf("%d", &a[i]);
    for(i=1; i<n; i++)
    {
        temp = a[i];
        j = i - 1;
        while((temp<a[j]) && (j>=0))
        {
            a[j+1] = a[j];
            j = j - 1;
        }
        a[j+1] = temp;
        p++;
        printf("\n After Pass %d: ", p);
        for(k=0; k<n; k++)
        printf(" %d", a[k]);
    }
    printf("\n Sorted List : ");
    for(i=0; i<n; i++)
    printf(" %d", a[i]);
}
```

Output

```
Enter total elements: 6
Enter array elements: 34 8 64 51 32 21
After Pass 1: 8 34 64 51 32 21
After Pass 2: 8 34 64 51 32 21
After Pass 3: 8 34 51 64 32 21
After Pass 4: 8 32 34 51 64 21
After Pass 5: 8 21 32 34 51 64
Sorted List : 8 21 32 34 51 64
```

Result: Thus array elements was sorted using insertion sort.

Ex. No. 11 d **BUBBLE SORT**
Date:

Aim: To sort an array of N numbers using Bubble sort.

Algorithm

1. Start
2. Read number of array elements n
3. Read array elements Ai
4. Outer Index i varies from last element to first element
    a. Index j varies from first element to i-1
        i. Compare elements Aj and Aj+1
        ii. If out-of-order then swap the elements
        iii. Display array elements after each pass
        iv. Display the final sorted list
5. Stop

Program

```
/* 11b - Bubble Sort */
#include <stdio.h>
main()
{
    int n, t, i, j, k, a[20], p=0;
    printf("Enter total numbers of elements: ");
    scanf("%d", &n);
    printf("Enter %d elements: ", n);
    for(i=0; i<n; i++)
    scanf("%d", &a[i]);
    for(i=n-1; i>=0; i--)
    {
        for(j=0; j<i; j++)
        {
            if(a[j] > a[j+1])
            {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
            }
        }
    p++;
    printf("\n After Pass %d : ", p);
    for(k=0; k<n; k++)
    printf("%d ", a[k]);
    }
    printf("\n Sorted List : ");
    for(i=0; i<n; i++)
    printf("%d ", a[i]);
```

```
}
```

Output

```
Enter total numbers of elements: 8
Enter 8 elements: 8 6 10 3 1 2 5 4
After Pass 1 : 6 8 3 1 2 5 4 10
After Pass 2 : 6 3 1 2 5 4 8 10
After Pass 3 : 3 1 2 5 4 6 8 10
After Pass 4 : 1 2 3 4 5 6 8 10
After Pass 5 : 1 2 3 4 5 6 8 10
After Pass 6 : 1 2 3 4 5 6 8 10
After Pass 7 : 1 2 3 4 5 6 8 10
Sorted List : 1 2 3 4 5 6 8 10
```

Result

Thus an array was sorted using bubble sort.

Ex. No. 11 e **QUICK SORT**
Date:

Aim: To sort an array of N numbers using Quick sort.

Algorithm

1. Start
2. Read number of array elements n
3. Read array elements Ai
4. Select an pivot element x from Ai
5. Divide the array into 3 sequences: elements < x, x, elements > x
6. Recursively quick sort both sets (Ai < x and Ai > x)
7. Display the sorted array elements
8. Stop

Program

```
/* 11c - Quick Sort */
#include<stdio.h>
#include<conio.h>
void qsort(int arr[20], int fst, int last);
main()
{
    int arr[30];
    int i, size;
    printf("Enter total no. of the elements : ");
    scanf("%d", &size);
    printf("Enter total %d elements : \n", size);
    for(i=0; i<size; i++)
    scanf("%d", &arr[i]);
    qsort(arr,0,size-1);
    printf("\n Quick sorted elements \n");
    for(i=0; i<size; i++)
    printf("%d\t", arr[i]);
    getch();
}


void qsort(int arr[20], int fst, int last)
{
    int i, j, pivot, tmp;
    if(fst < last)
    {
        pivot = fst;
        i = fst;
        j = last;
        while(i < j)
```

```
        {
            while(arr[i] <=arr[pivot] && i<last)
            i++;
            while(arr[j] > arr[pivot])
            j--;
            if(i <j )
            {
                tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
            }
        }
        tmp = arr[pivot];
        arr[pivot] = arr[j];
        arr[j] = tmp;
        qsort(arr, fst, j-1);
        qsort(arr, j+1, last);
    }
}
```

Output

Enter total no. of the elements : 8
Enter total 8 elements :
127
-1
04
-2
3
Quick sorted elements
-2 -1 0 1 2 3 4 7

Result

Thus the array was sorted using quick sort's divide and conquers method.

Ex. No. 11 f **MERGE SORT**
Date:

Aim: To sort an array of N numbers using Merge sort.

Algorithm

1. Start
2. Read number of array elements n
3. Read array elements Ai
4. Divide the array into sub-arrays with a set of elements
5. Recursively sort the sub-arrays
6. Display both sorted sub-arrays
7. Merge the sorted sub-arrays onto a single sorted array.
8. Display the merge sorted array elements
9. Stop

Program

```
/* 11d – Merge sort */
#include <stdio.h>
#include <conio.h>
void merge(int [],int ,int ,int );
void part(int [],int ,int );
int size;
main()
{
    int i, arr[30];
    printf("Enter total no. of elements : ");
    scanf("%d", &size);
    printf("Enter array elements : ");
    for(i=0; i<size; i++)
    scanf("%d", &arr[i]);
    part(arr, 0, size-1);
    printf("\n Merge sorted list : ");
    for(i=0; i<size; i++)
    printf("%d ",arr[i]);
    getch();
}

void part(int arr[], int min, int max)
{
    int mid;
    if(min < max)
    {
        mid = (min + max) / 2;
        part(arr, min, mid);
        part(arr, mid+1, max);
        merge(arr, min, mid, max);
```

```
    }
    if (max-min == (size/2)-1)
    {
        printf("\n Half sorted list : ");
        for(i=min; i<=max; i++)
        printf("%d ", arr[i]);
    }
}

void merge(int arr[],int min,int mid,int max)
{
    int tmp[30];
    int i, j, k, m;
    j = min;
    m = mid + 1;
    for(i=min; j<=mid && m<=max; i++)
    {
        if(arr[j] <= arr[m])
        {
            tmp[i] = arr[j];
            j++;
        }
        else
        {
            tmp[i] = arr[m];
            m++;
        }
    }
    if(j > mid)
    {
        for(k=m; k<=max; k++)
        {
            tmp[i] = arr[k];
            i++;
        }
    }
    else
    {
        for(k=j; k<=mid; k++)
        {
            tmp[i] = arr[k];
            i++;
        }
    }
    for(k=min; k<=max; k++)
    arr[k] = tmp[k];
}
```

Output

Enter total no. of elements : 8
Enter array elements : 24 13 26 1 2 27 38 15
Half sorted list : 1 13 24 26
Half sorted list : 2 15 27 38
Merge sorted list : 1 2 13 15 24 26 27 38

Result

Thus array elements was sorted using merge sort's divide and conquer method.

Ex. No. 12 **IMPLEMENTATION OF HASH FUNCTIONS AND COLLISION RESOLUTION TECHNIQUE**

Date:

Aim: To implement hashing technique and collision resolution technique.

Algorithm

1. Start
2. A structure that represent the key, data pair is created
3. Key is generated by hashcode function which returns the value of key%size where size is assumed as 20
4. Insert function is called to insert a new key value pair into the hash table.
5. While inserting an element if the hashcode function returns a key that has been previously assigned to an element in the hash table then a collision occurs.
6. If there is a collision then the hash key is incremented to move to the next place and find whether there is a key-space for inserting the current element. This is called collision resolution. This process is repeated until a space if found for current element or the hash table is full
7. Delete function is called to delete an element from the hash table.
8. Stop

Program

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

#define SIZE 20

struct DataItem
{
int data;
  int key;
};

struct DataItem* hashArray[SIZE];
struct DataItem* dummyItem;
struct DataItem* item;

int hashCode(int key)
{
return key % SIZE;
}

struct DataItem *search(int key)
{
     int hashIndex = hashCode(key);  //get the hash
```

```
     while(hashArray[hashIndex] != NULL) //move in array until an empty
{
        if(hashArray[hashIndex]->key == key)
 return hashArray[hashIndex];
    ++hashIndex;      //go to next cell
     hashIndex %= SIZE;      //wrap around the table
  }
  return NULL;
}

void insert(int key,int data)
{
  struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
  item->data = data;
  item->key = key;

   int hashIndex = hashCode(key);//get the hash

  //move in array until an empty or deleted cell
  while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1)
{
        ++hashIndex;  //go to next cell
     hashIndex %= SIZE;      //wrap around the table
}
  hashArray[hashIndex] = item;
}

struct DataItem* delete(struct DataItem* item)
{
  int key = item->key;
  int hashIndex = hashCode(key);   //get the hash
  while(hashArray[hashIndex] != NULL)   //move in array until an empty
{
     if(hashArray[hashIndex]->key == key)
{
        struct DataItem* temp = hashArray[hashIndex];
        hashArray[hashIndex] = dummyItem; //assign a dummy item at deleted
                                                                  //position
        return temp;
     }
     ++hashIndex;      //go to next cell
     hashIndex %= SIZE;  //wrap around the table
  }
  return NULL;
}
```

```
void display()
{
  int i = 0;

  for(i = 0; i<SIZE; i++)
{
    if(hashArray[i] != NULL)
      printf(" (%d,%d)",hashArray[i]->key,hashArray[i]->data);
    else
      printf(" ~~ ");
  }
  printf("\n");
}

int main()
{
  dummyItem = (struct DataItem*) malloc(sizeof(struct DataItem));
  dummyItem->data = -1;
  dummyItem->key = -1;

  insert(1, 20);
  insert(2, 70);
  insert(42, 80);
  insert(4, 25);
  insert(12, 44);
  insert(14, 32);
  insert(17, 11);
  insert(13, 78);
  insert(37, 97);

  display();
  item = search(37);

  if(item != NULL)
{
    printf("Element found: %d\n", item->data);
} else
{
    printf("Element not found\n");
}

  delete(item);
  item = search(37);

  if(item != NULL)
{
    printf("Element found: %d\n", item->data);
```

```
  } else
{
    printf("Element not found\n");
  }
}
```

Output

~~ (1,20) (2,70) (42,80) (4,25) ~~ ~~ ~~ ~~ ~~ ~~ ~~ (12,44) (13,78) (14,32) ~~ ~~ (17,11) (37,97) ~~
Element found: 97
Element not found

Result

The program for demonstrating hashing and collision resolution is executed and verified.

VIVA QUESTIONS WITH ANSWERS

1. Define global declaration?

The variables that are used in more than one function throughout the program are called global variables and declared in outside of all the function that is before the main() function.

2. Define data types?

Data type is the type of data that are going to access within the program. „C" supports different data types

PrimaryUserdefinedDerivedEmpty

char arrays
int typedef pointers void
float structures
double union
Example: int a,b; here int is data type

3.Define variable with example?

A variable is an identifier and it may take different values at different times of during the execution . A variable name can be any combinations of 1 to 8 alphabets, digits or underscore.
Example: int a,b; here a,b are variables

4. What is decision making statement?

Decision making statement is used to break the normal flow of the program and execute part of the statement based on some condition.

5.What are the various decision making statements available in C ?

- If statement
- if…else statement
- nested if statement
- if…else ladder statement
- switch statement

6. Distinguish between Call by value Call by reference.

Call by value	Call by reference.
In call by value, the value of actual arguments is passed to the formal arguments and the operation is done onformal arguments.	In call by reference, the address of actual argurment values is passed to formal argument values.
Formal arguments values are photocopiesof actual arguments values.	Formal arguments values are pointers to actual argument values.
Changes made in formal argumentsvalued do not affect the actual arguments values.	Since Address is passed, the changes made in the both arguments values are Permanent

7. What is an array?

An array is a collection of data of same data type. the elements of the array are stored in continuous memory location and array elements are processing using its index. Example: int a[10];

Here 'a' is an array name.

8. What is two dimensional array?

Two dimensional is an array of one dimensional array. The elements in the array are referenced with help of its row and column index. Example: Int a[2][2];

9. Define is function?

Function are group of statements that can be perform a task. Function reduce the amount of coding and the function can be called from another program. Example: main()

```
{
-------
fun();
--------
}
fun()
{
-------
}
```

10.What are the various looping statements available in 'C'?

- While statement
- Do….while statement
- For statement

11.What are the uses of Pointers?

- Pointers are used to return more than one value to the function
- Pointers are more efficient in handling the data in arrays
- Pointers reduce the length and complexity of the program
- They increase the execution speed
- The pointers save data storage space in memory

12.What is a Pointer? How a variable is declared to the pointer? (MAY 2009)

Pointer is a variable which holds the address of another variable.

Pointer Declaration:

datatype *variable-name;

Example:

int *x, c=5; x=&a;

13. What is the difference between if and while statement?

If	While
It is a conditional statement	It is a loop control statement
If the condition is true, it executes some statements.	Executes the statements within the while block if the condition is true.
If the condition is false then it stops the execution the statements	If the condition is false the control is transferred to the next statement of the loop.

14. Define pre-processor in C.

Preprocessor are used to link the library files in to source program, that are placed before the main() function and it have three preprocessor directives that are

- Macro inclusion
- Conditional inclusion
- File inclusion

15.Define recursive function?

A function is a set of instructions used to perform a specified task which repeatedly occurs in the main program. If a function calls itself again and again , hten that function is called recursive function.

16. What is the difference between while and do….while statement?

The while is an entry controlled statement. The statement inside the while may not be executed at all when the condition becomes false at the first attempt itself.

The do …while is an exit controlled statement. The statements in the block are executed at least once.

17. Define Operator with example?

An operator is a symbol that specifies an operation to be performed on operands. Some operators require two operands called binary operators, while other acts upon only one operand called unary operator. Example: a+b here a,b are operands and + is operator

18. Define conditional operator or ternary operator?

Conditional operator itself checks the condition and execute the statement depending on the condition. (a>b)?a:b if a is greater than b means the a value will be return otherwise b value will be return. Example: big=a>b?a:b;

19. Compare of switch() case and nested if statement

switch()	**nested if statement**
The switch() can test only constant values.	The if can evaluate relational or logical expressions.
No two case statements have identical constants in the same switch.	Same conditions may be repeated for number of times

Character constants are automatically converted to integers	Character constants are automatically converted to integers
In switch() case statement, nested if can used	In nested if statements, switch() case can be used

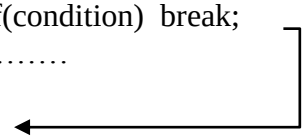
20.What are steps involved in looping statements?

- Initialization of a condition variable.
- Test the control statement.
- Executing the body of the loop depending on the condition.
- Updating the condition variable.

21.Define break statement?

The break statements is used terminate the loop. When the break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. Example:

```
while(condition)
{
……
 If(condition)  break;
……
 }
```



22.Define null pointer?

A pointer is said to be a null pointer when its right value is 0, a null pointer can never point to a valid data. For checking a pointer, if it is assigned to 0, then it is a null pointer and is not valid
Example:
int *a; int *b; a=b=0;

23. Compare arrays and structures.

Arrays	Structures
An array is a collection of data itemsof same data type	A structure is a collection of data items of different data types.
Arrays can only be declared	Structures can be declared anddefined.
There is no keyword for arrays	The keyword for structures is struct
An array name represents the address of the starting element.	A structrure name is known as tag. It is a Shorthand notation of the declaration.
An array cannot have bit fields.	A structure may contain bit fields.

24. Compare structures and unions.

Structure	Union
Every member has its own memory.	All members use the same memory.
The keyword used is struct.	The keyword used is union.
All members occupy separatememory locaton, hence different interpretations of same memory location are not possible	Different interpretations for the same memory location are possible
Consumes more space compared to union	Conservation of memory is possible

25.Define Structure in C.

A structure contains one or more data items of different data type in which the individual elements can differ in type. A simple structure may contain the integer elements, float elements and character elements etc. and the individual elements are called members. Example: struct result { int marks; float avg; char grade; }std;

26.Rules for declaring a structure?

- A structure must end with a semicolon.
- Usually a structure appears at the top of the source program.
- Each structure element must be terminated.
- The structure must be accessed with structure variable with dot (.) operator.

27.Define structure pointers

Pointer is a variable, it contain address of another variable and the structure pointers are declared by placing * in front of a structure variable‟s name.

Example: struct result { int marks;
float avg; char grade; }; struct result *sam;

28. Define union?

A union, is a collection of variables of different types, just like structure. Union is a derived data type and the difference between union and structure is in terms of storage. In structure each member has its own storage location, whereas all the members of union use the same memory location. Example: union result { int marks; float avg; char grade; }std;

29. Define file?

A file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind. The collection of bytes may be interrupted, for example, as characters, words, lines, paragraph and pages from a textual document. Example: FILE *infile; FILE *outfile;

30. Define binary files?

Binary files can be processed sequentially or, depending on the needs of the application, they can process using random access techniques. In C, processing a file using random access techniques involves moving the current file position to an appropriate place in the file before reading or writing data.

31. Define opening a file?

A file requires to be opened first with the file pointer positioned on the first character. No input-output functions on a stream can be performed unless it is opened. When a stream is opened, it is connected to named DOS device or file .C provides a various functions to open a file as a stream. Syntax: FILE *fopen(char * filename, char *mode);

32. Define fseek()?

fseek() will position the file pointer to a particular byte within the file. The file pointer is a pointer is a parameter maintained by the operating system and determines where the next read will comes from , or to where the next write will go.

33.Functions of bit fields?

Bit fields do not have address.
It is not an array.
It cannot be accessed using pointer.
It cannot be store values beyond their limits. If larger values are assigned, the output is undefined

34. What is meant by an abstract data type?

An ADT is an object with a generic description independent of implementation details. This description includes a specification of an components from which the object is made and also behavioral details of objects.

35.Advantages and Disadvantages of arrays?

Advantages:
Data accessing is faster
Arrays are simplein terms of understanding point and in terms of programming.

Disadvantages:
Array size is fixed
Array elements stored continuously
Insertion and deletion of elements in an array is difficult.

36.What is an array?

Array may be defined abstractly as a finite ordered set of homogenous elements. Finite means there is a specific number of elements in the array.

37. What is a linked list?

Linked list is a kind of series of data structures, which are not necessarily adjacent in memory. Each structure contains the element and a pointer to a record containing its successor.

38. What is singly linked list?

A singly linked list is a linked list, there exists only one link field in each and every node and all nodes are linked together in some sequential manner and this type of linked list is called singly linked list.

39. What is a doubly linked list?

In a simple linked list, there will be one pointer named as „NEXT POINTER‟ to point the next element, where as in a doubly linked list, there will be two pointers one to point the next element and the other to point the previous element location.

40. Define double circularly linked list?

In a doubly linked list, if the last node or pointer of the list, point to the first element of the list, then it is a circularly linked list.

41. What is the need for the header?

Header of the linked list is the first element in the list and it stores the number of elements in the list. It points to the first data element of the list.

42.Define Polynomial ADT

A polynomial object is a homogeneous ordered list of pairs <exponent,coefficient>,where each coefficient is unique.

Operations include returning the degree, extracting the coefficient for a given exponent, addition, multiplication, evaluation for a given input.

$10x^4+5x^2+1$

43. How to search an element in list.

Searching can be initiated from first node and it is compared with given element one after the other until the specified key is found or until the end of the list is encountered.

44. Define Dqueue?

Dqueue is also data structure where elements can be inserted from both ends and deleted from both ends. To implement a dqueue operations using singly linked list operations performed insert_front, delete_front, insert_rear, delete_rear and display functions.

45. How to implement stack using singly linked list

Stack is an Last In First Out (LIFO) data structure. Here , elements are inserted from one end called push operation and the same elements are deleted from the same end called pop operation So, using singly linked list stack operations are performed in the front or other way ew can perform rear end also.

46.What are the types of Linear linked list?

- Singly linked lists
- Circular singly linked lists
- Doubly linked lists
- Circular doubly linked lists

47.What are advantages of Linked lists?

- Linked lists are dynamic data structures
- The size is not fixed
- Data can store non-continuous memory blocks
- Insertion and deletion of nodes are easier and efficient

Complex applications

48.Write down the algorithm for solving Towers of Hanoi problem?

- Push parameters and return address on stack.
- If the stopping value has been reached then pop the stack to return to previous level else move all except the final disc from starting to intermediate needle.
- Move final discs from start to destination needle.
- Move remaining discs from intermediate to destination needle.
- Return to previous level by popping stack.

49.What is a Stack ?

A stack is a non-primitive linear data structure and is an ordered collection of homogeneous data elements.The other name of stack is Last-in -First-out list. One of the most useful concepts and frequently used data structure of variable size for problem solving is the stack.

50.What are the two operations of Stack?

- PUSH
- POP

51.What is a Queue ?

A Queue is an ordered collection of items from which items may be deleted at one end called the front of the queue and into which tems may be inserted at the other end called rear of the queue.Queue is called as First-in-First-Out(FIFO).

52. What is a Priority Queue?

Priority queue is a data structure in which the intrinsic ordering of the elements does determine the results of its basic operations. Ascending and Descending priority queue are the two types of Priority queue.

53.What are the different ways to implement list?

- Simple array implementation of list
- Linked list implementation of list
- cursor implementation of list

55.What are the postfix and prefix forms of the expression?
A+B*(C-D)/(P-R)

- Postfix form: ABCD-*PR-/+
- Prefix form: +A/*B-CD-PR

56.Explain the usage of stack in recursive algorithm implementation?
In recursive algorithms, stack data structures is used to store the return address when a recursive call is encountered and also to store the values of all the parameters essential to the current state of the procedure.

57. Write down the operations that can be done with queue data structure?
Queue is a first - in -first out list. The operations that can be done with queue are insert and remove.

58. What is a circular queue?
The queue, which wraps around upon reaching the end of the array is called as circular queue.

59.Give few examples for data structures?

- Stacks
- Queue
- Linked list
- Trees
- Graphs

60.List out Applications of queue
Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.
Computer systems must often provide a "holding area" for messages between two processes, two programs, or even two systems. This holding area is usually called abuffer" and is often implemented as a queue.

61.How do you test for an empty queue?
To test for an empty queue, we have to check whether READ=HEAD where REAR is a pointer pointing to the last node in a queue and HEAD is a pointer that pointer to the dummy header. In the case of array implementation of queue, the condition to be checked for an empty queue is READ<FRONT.

62.What are applications of stack?

- Conversion of expression
- Evaluation of expression
- Parentheses matching
- Recursion

63.Define recursion?

It is a technique and it can be defined as any function that calls itself is called recursion. There are some applications which are suitable for only recursion such as, tower of Hanoi, binary tree traversals etc, can be implementing very easily and efficiently.

64.What are types of Queues?

- Simple queue (ordinary queue)
- Circular queue
- Double ended queue
- Priority queue

65. What is meant by Sorting and searching?

Sorting and searching are fundamentals operations in computer science. Sorting refers to the operation of arranging data in some given order Searching refers to the operation of searching the particular record from the existing information

66.What are the types of sorting available in C?

Insertion sort
Merge Sort
Quick Sort
Radix Sort
Heap Sort
Selection sort
Bubble sort

67.Define Bubble sort.

Bubble sort is the one of the easiest sorting method. In this method each data item is compared with its neighbor and if it is an descending sorting , then the bigger number is moved to the top of all The smaller numbers are slowly moved to the bottom position, hence it is also called as the exchange sort.

68.Mention the various types of searching techniques in C

- Linear search
- Binary search

69.What is linear search?

In Linear Search the list is searched sequentially and the position is returned if the key element to be searched is available in the list, otherwise -1 is returned. The search in Linear Search starts at the beginning of an array and move to the end, testing for a match at each item.

70. What is binary search?

Binary search is simpler and faster than linear search. Binary search the array to be searched is divided into two parts, one of which is ignored as it will not contain the required element One essential condition for the binary search is that the array which is to be searched, should be arranged in order.

71. Define merge sort?

Merge sort is based on divide and conquer method. It takes the list to be stored and divide it in half to create two unsorted lists. The two unsorted lists are then sorted and merge to get a sorted list.

72. Define insertion sort?

Successive element in the array to be sorted and inserted into its proper place with respect to the other already sorted element. We start with second element and put it in its correct place, so that the first and second elements of the array are in order.

73. Define selection sort?

It basically determines the minimum or maximum of the lists and swaps it with the element at the index where its supposed to be. The process is repeated such that the n^{th} minimum or maximum element is swapped with the element at the $n\text{-}1^{th}$ index of the list.

74. What is the basic idea of shell sort?

Shell sort works by comparing elements that are distant rather than adjacent elements in an array or list where adjacent elements are compared. Shell sort uses an increment sequence. The increment size is reduced after each pass until increment size is 1.

75.What is the purpose of quick sort and advantage?

The purpose of the quick sort is to move a data item in the correct direction, just enough for to reach its final place in the array.
Quick sort reduces unnecessary swaps and moves an item to a greater distance, in one move.

76.Define quick sort?

The quicksort algorithm is fastest when the median of the array is chosen as the pivot value. That is because the resulting partitions are of very similar size. Each partition splits itself in two and thus the base case is reached very quickly and it follow the divide and conquer strategy.

77. Advantage of quick sort?

Quick sort reduces unnecessary swaps and moves an item to a greater distance, in one move.

78. Define radix sort?

Radix sort the elements by processing its individual digits. Radix sort processing the digits either by least significant digit(LSD) method or by most significant digit(MSD) method.
Radix sort is a clever and intuitive little sorting algorithm, radix sort puts the elements in order by comparing the digits of the numbers.

79. List out the different types of hashing functions?

The different types of hashing functions are,

- The division method
- The mind square method
- The folding method
- Multiplicative hashing

- Digit analysis

80.Define hashing?
Search from that position for an empty location
Use a second hash function.
Use that array location as the header of a linked list of values that hash to this location

81.Define hash table?
All the large collection of data are stored in a hash table. The size of the hash table is usually fixed and it is bigger than the number of elements we are going to store. The load factor defines the ration of the number of data to be stored to the size of the hash table

82.What are the types of hashing?
Static hashing-In static hashing the process is carried out without the usage of an index structure.
Dynamic hashing- It allows dynamic allocation of buckets, i.e. according to the demand of database the buckets can be allocated making this approach more efficient.

83.Define Rehashing?
Rehashing is technique also called as double hashing used in hash tables to resolve hash collisions, cases when two different values to be searched for produce the same hash key.
It is a popular collision-resolution technique in open-addressed hash tables

84 .What is data structure?
The logical and mathematical model of a particular organization of data is called data structure.
There are twotypes of data structure1.Linear2.Nonlinear

85.What are the goals of Data Structure?
It must rich enough in structure to reflect the actual relationship of data in real world.The structure should be simple enough for efficient processing of data.

86.What does abstract Data Type Mean?
Data type is a collection of values and a set of operations on these values. Abstract data type refer to themathematical concept that define the data type.
It is a useful tool for specifying the logical properties of a datatype.
ADT consists of two parts1.Values definition2.Operation definition

87.What is the difference between a Stack and an Array?
Stack is a ordered collection of items
Stack is a dynamic object whose size is constantly changing as items are pushed and popped .
Stack may contain different data types
Stack is declared as a structure containing an array to hold the element of the stack, and an integer to indicate the current stack top within the array.

ARRAYArray is an ordered collection of items

Array is a static object i.e. no of item is fixed and is assigned by the declaration of the array It contains same data types.
Array can be home of a stack i.e. array can be declared large enough for maximum size of the stack.

88.What do you mean by recursive definition?
The definition which defines an object in terms of simpler cases of itself is called recursive definition.

89.What is sequential search?
In sequential search each item in the array is compared with the item being searched until a match occurs. It is applicable to a table organized either as an array or as a linked list.

90.What actions are performed when a function is called?
When a function is called
i)arguments are passed
ii)local variables are allocated and initialized
iii)transferring control to the function

91.What actions are performed when a function returns?
i)Return address is retrieved
ii)Function's data area is freed
iii)Branch is taken to the return address