Ex.No :1(a) **STACK – ARRAY IMPLEMENTATION**

Date :

AIM:

To write a C program to implement the stack using arrays.

ALGORITHM:

(i) Push Operation:

Step-1: To push an element into the stack, check whether the top of the stack is greaterthan or equal to the maximum size of the stack.

Step-2: If so, then return stack is full and element cannot be pushed into the stack.

Step-3: Else, increment the top by one and push the new element in the new position oftop.

(ii) Pop Operation:

Step-1: To pop an element from the stack, check whether the top of the stack is equalto -1.

Step-2: If so, then return stack is empty and element cannot be popped from the

stack. o Else, decrement the top by one.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5
struct stack {
    int s[SIZE];
    int top;
};
struct stack st;
int stfull() {
    return st.top >= SIZE - 1;
}
void push(int item) {
    if (!stfull()) {
st.s[++st.top] = item;
    }
}
int stempty() {
    return st.top == -1;
}
int pop() {
    if (!stempty()) {
        return st.s[st.top--];
    }
    return -1;
}
void display() {
    if (stempty()) {
printf("Stack is Empty!\n");
    } else {
        for (int i = st.top; i >= 0; i--) {
printf("%d\n", st.s[i]);
        }
    }
}
int main(void) {
    int item, ch;
    st.top = -1;
printf("<----- Stack using Array ----->\n");
    while (1) {
printf("\n1. Push\n2. Pop\n3. Display\n4. Exit\n");
printf("Enter Your Choice: ");
scanf("%d", &ch);
        switch (ch) {
            case 1:
printf("Enter the item to be pushed: ");
scanf("%d", &item);
                if (stfull()) {
printf("Stack is Full!\n");
                } else {
                    push(item);
                }
                break;
            case 2:
                if (stempty()) {
printf("Empty stack!\n");
                } else {
                    item = pop();
printf("The popped element is %d\n", item);
                }
                break;
            case 3:
display();
                break;
            case 4:
exit(0);
```

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

OUTPUT:

```
<-----Stack using Array----->
1.Push
2.Pop
3.Display
4.exit
Enter Your Choice:
1
Enter The item to be pushed: 10
1.Push
2.Pop
3.Display
4.exit
Enter Your Choice:
1
Enter The item to be pushed: 20
1.Push
2.Pop
3.Display
4.exit
Enter Your Choice: 3
20
10
1.Push
```

```
2.Pop
3.Display
4.exit
Enter Your Choice: 2
The popped element is 20
1.Push
2.Pop
3.Display
4.exit
Enter Your Choice: 4
```

`

Ex.No:1(b) **QUEUE – ARRAY IMPLEMENTATION**

DATE:

PROGRAM:

```
#include <stdio.h>
#define MAX 10
int queue[MAX], front = -1, rear = -1;
void insert_element();
void delete_element();
void display_queue();
int main() {
   int option;
printf(">>> C program to implement queue operations <<<");
   do {
printf("\n\n 1. Enqueue an element");
printf("\n 2. Dequeue an element");
printf("\n 3. Display queue");
printf("\n 4. Exit");
printf("\n Enter your choice: ");
scanf("%d", &option);
      switch(option) {
         case 1:
            insert_element();
            break;
         case 2:
            delete_element();
            break;
         case 3:
            display_queue();
```

```
                break;
            case 4:
                return 0;
            default:
printf("\n Invalid option! Please choose again.");
        }
    } while(option != 4);

    return 0;
}
void insert_element() {
    int num;
printf("\n Enter the number to be Enqueued: ");
scanf("%d", &num);
if ((front == 0 && rear == MAX - 1) || (rear + 1) % MAX == front) {
printf("\n Queue Overflow Occurred");
    } else {
        if (front == -1 && rear == -1) {
            front = rear = 0;
        } else {
  rear = (rear + 1) % MAX;
        }
        queue[rear] = num;
}}
void delete_element() {
    if (front == -1) {
printf("\n Underflow");
    } else {
        int element = queue[front];
printf("\n The dequeued element is: %d", element);
   if (front == rear) {
            front = rear = -1; // Queue is now empty
        } else {
front = (front + 1) % MAX;
        }}}
void display_queue() {
    if (front == -1) {
printf("\n No elements to display");
    } else {
printf("\n The queue elements are:\n ");
        int i = front;
        while (1) {
printf("\t %d", queue[i]);
            if (i == rear) break;
            i = (i + 1) % MAX;
 }}}
```

OUTPUT:

```
>>> c program to implement queue operations <<<
1.Enqueue an element
2.Dequeue an element
3.Display queue
4.Exit
Enter your choice: 1
Enter the number to be Enqueued: 10
1.Enqueue an element
2.Dequeue an element
```

```
3.Display queue

4.Exit

Enter your choice: 1

Enter the number to be Enqueued: 20

1.Enqueue an element

2.Dequeue an element

3.Display queue

4.Exit

Enter your choice: 3

The queue elements are: 10 20

1.Enqueue an element

2.Dequeue an element

3.Display queue

4.Exit

Enter your choice: 2

The dequeued element is: 10

1.Enqueue an element

2.Dequeue an element

3.Display queue

4.Exit

Enter your choice: 4
```

Ex.No:1(c) CIRCULAR QUEUE – ARRAY IMPLEMENTATION

DATE:

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 5
typedef struct CircularQueue {
   int items[MAX_SIZE];
   int front;
   int rear;
} CircularQueue;
CircularQueue* createQueue() {
   CircularQueue* queue = (CircularQueue*)malloc(sizeof(CircularQueue));
   queue->front = -1;
   queue->rear = -1;
   return queue;
}
int isFull(CircularQueue* queue) {
   return (queue->rear + 1) % MAX_SIZE == queue->front;
}
int isEmpty(CircularQueue* queue) {
   return queue->front == -1;
}
void enqueue(CircularQueue* queue, int value) {
   if (isFull(queue)) {
      printf("Queue is full! Cannot enqueue %d.\n", value);
      return;
   }
   if (isEmpty(queue)) {
      queue->front = 0;     }
   queue->rear = (queue->rear + 1) % MAX_SIZE;
```

```
  queue->items[queue->rear] = value;
  printf("Enqueued: %d\n", value);
}
int dequeue(CircularQueue* queue) {
  if (isEmpty(queue)) {
    printf("Queue is empty! Cannot dequeue.\n");
    return -1;
  }
  int removedValue = queue->items[queue->front];
  if (queue->front == queue->rear) {        queue->front = -1;
    queue->rear = -1;
  } else {
    queue->front = (queue->front + 1) % MAX_SIZE;
  }
  printf("Dequeued: %d\n", removedValue);
  return removedValue;
}

int peek(CircularQueue* queue) {
  if (isEmpty(queue)) {
    printf("Queue is empty! Nothing to peek.\n");
    return -1;     }
  return queue->items[queue->front];
}
void display(CircularQueue* queue) {
  if (isEmpty(queue)) {
    printf("Queue is empty!\n");
    return;
  }
  int i = queue->front;
  printf("Queue elements: ");
  while (1) {
    printf("%d ", queue->items[i]);
    if (i == queue->rear) break;
    i = (i + 1) % MAX_SIZE;
  }
  printf("\n");
}
int main() {
  CircularQueue* queue = createQueue();
  enqueue(queue, 10);
  enqueue(queue, 20);
  enqueue(queue, 30);
  enqueue(queue, 40);
  enqueue(queue, 50);
  display(queue);
  dequeue(queue);
  dequeue(queue);
  display(queue);
 enqueue(queue, 60);
  enqueue(queue, 70);
  display(queue);
  free(queue);
  return 0;
}
```

OUTPUT:

```
Enqueued: 10
Enqueued: 20
Enqueued: 30
Enqueued: 40
Enqueued: 50
Queue elements: 10 20 30 40 50
Dequeued: 10
Dequeued: 20
```

```
Queue elements: 30 40 50

Enqueued: 60

Enqueued: 70

Queue elements: 30 40 50 60 70
```

RESULT:

Thus a C program to implement the stack, queue ,circular using arrays is written and executed successfully.

Ex.No:2

DATE :

LIST – ARRAY IMPLEMENTATION

AIM:

To write a C program to implement the List using arrays.

ALGORITHM:

Step-1: Start the program.

Step-2: Read the number of elements in the list and create the list.

Step-3: Read the position and element to be inserted.

Step-4: Adjust the position and insert the element.

Step-5: Read the position and element to be deleted.

Step-6: Remove the element from the list and adjust the position.

Step-7: Read the element to be searched.

Step-8: Compare the elements in the list with searching element.

Step-9: If element is found, display it else display element is not found.

Step-10: Display all the elements in the list.

Step-11: Stop the program.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
void create();
void insert();
void deletion();
void search();
void display();
int b[MAX], n = 0;
int main() {
int ch;
char g = 'y';
do {
printf("\nMain Menu");
printf("\n1. Create");
printf("\n2. Delete");
printf("\n3. Search");
printf("\n4. Insert");
printf("\n5. Display");
printf("\n6. Exit");
printf("\nEnter 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(0);
default:
printf("\nEnter the correct choice:");
}
printf("\nDo you want to continue (y/n)? ");
scanf(" %c", &g);
} while (g == 'y' || g == 'Y');
return 0;
```

```
        }
        void create() {
        printf("\nEnter the number of nodes (max %d): ", MAX);
        scanf("%d", &n);
        if (n > MAX) {
        printf("Error: Cannot exceed %d elements.\n", MAX);
        n = MAX;
        }
        for (int i = 0; i < n; i++) {
        printf("\nEnter the Element %d: ", i + 1);
        scanf("%d", &b[i]);
        }
        }
        void deletion() {
        int pos;
        printf("\nEnter the position you want to delete: ");
        scanf("%d", &pos);
        if (pos < 0 || pos >= n) {
        printf("\nInvalid Location:\n");
        } else {
        for (int i = pos + 1; i < n; i++) {
        b[i - 1] = b[i];
        }
        n--;
        printf("\nThe Elements after deletion: ");
        display();
        }
        }
        void search() {
        int e;
        printf("\nEnter the Element to be searched: ");
        scanf("%d", &e);
        int found = 0;
        for (int i = 0; i < n; i++) {
        if (b[i] == e) {
        printf("Value %d is in position %d\n", e, i);
        found = 1;
        break; // Exit loop on first match
        }
        }
        if (!found) {
        printf("Value %d is not in the list.\n", e);
        }
        }
        void insert() {
        int pos, p;
printf("\nEnter the position you need to insert (0 to %d): ", n);
        scanf("%d", &pos);
```

```
                    if (pos < 0 || pos > n || n >= MAX) {
                    printf("\nInvalid Location or List is Full:\n");
                    } else {
                    for (int i = n; i > pos; i--) {
                    b[i] = b[i - 1];
                    }
                    printf("\nEnter the element to insert: ");
                    scanf("%d", &p);
                    b[pos] = p;
                    n++;
                    printf("\nThe list after insertion: ");
                    display();
                    }
                    }
                    void display() {
                    if (n == 0) {
                    printf("\nThe list is empty.\n");
                    return;
                    }
                    printf("\nThe Elements of The list ADT are: ");
                    for (int i = 0; i < n; i++) {
                    printf("%d ", b[i]);
                    }
                    printf("\n");
}
```

OUTPUT:

```
Main Menu

1.Create

2.Delete

3.Search

4.Insert

5.Display

6.Exit

Enter your Choice:1

Enter the number of nodes:4

Enter the Element: 11

Enter the Element: 22

Enter the Element: 33

Enter the Element: 44

Do u want to continue:y
```

```
Main Menu
1.Create
2.Delete
3.Search
4.Insert
5.Display
6.Exit
Enter your Choice:2
Enter the position u want to delete:2
Elements after deletion: 11 33 44
Do u want to continue:y
Main Menu
1.Create
2.Delete
3.Search
4.Insert
5.Display
6.Exit
Enter your Choice:3
Enter the Element to be searched:44
Value is in the 3 Position
Do u want to continue:y
Main Menu
1.Create
2.Delete
3.Search
4.Insert
5.Display
6.Exit
Enter your Choice:4
Enter the position u need to insert:2
Enter the element to insert:25
The list after insertion: 11 25 33 44
```

```
Do u want to continue:y
Main Menu
1.Create
2.Delete
3.Search
4.Insert
5.Display
6.Exit
Enter your Choice:5
The Elements of The list ADT are: 11 25 33 44
Do u want to continue:y
Main Menu
1.Create
2.Delete
3.Search
4.Insert
5.Display
6.Exit
Enter your Choice:6
```

RESULT:

Thus a C program to implement the List using array is written and executed successfully.

Ex.No:3(a) # LIST – LINKED LIST IMPLEMENTATION

DATE:

AIM:

To write a C program to implement the List ADT using linked list.

ALGORITHM:

Step-1: Start the program.

Step-2: Create a node using structure

Step-3: Dynamically allocate memory to node

Step-4: Create and add nodes to linked list.

Step-5: Read the element to be inserted.

Step-6: Insert the elements into the list.

Step-7: Read the element to be deleted.

Step-8: Remove that element and node from the list.

Step-9: Adjust the pointers.

Step-10: Display the elements in the list.

Step-11: Stop the program.

PROGRAM:

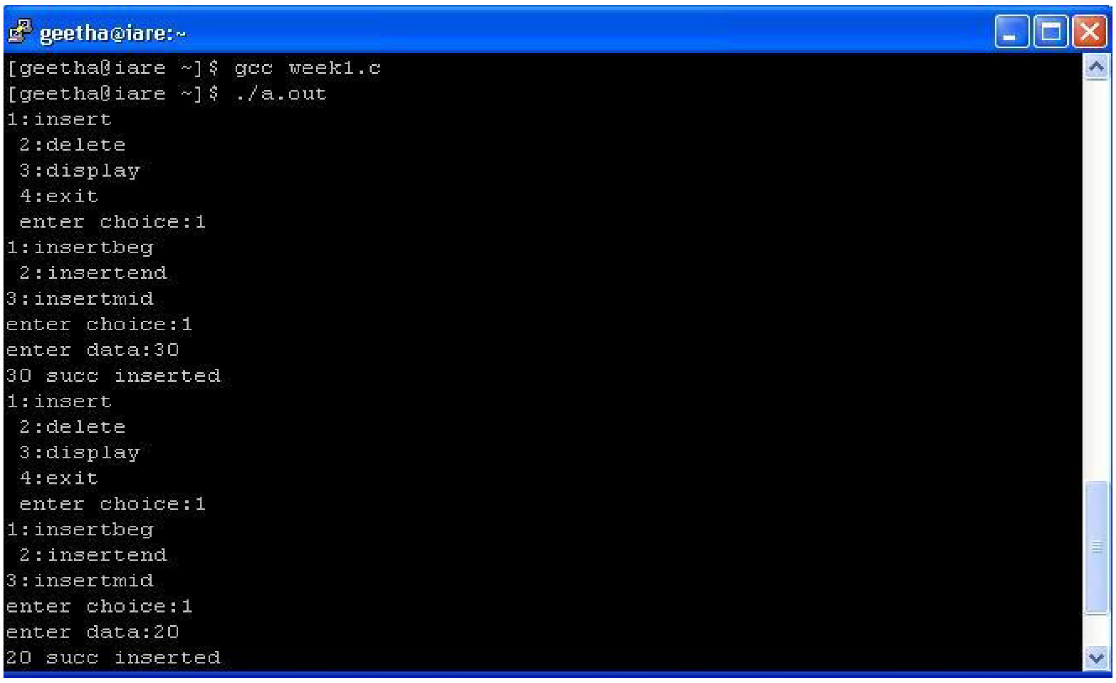
```
#include <stdio.h>
#include <stdlib.h>
struct node {
    int data;
    struct node *next;
};
struct node *start = NULL; void insertbeg(void) {
    struct node *nn = (struct node *)malloc(sizeof(struct node));
    if (!nn) {
printf("Memory allocation failed.\n");
        return;
    }
printf("Enter data: ");
scanf("%d", &nn->data);
    nn->next = start;
    start = nn;
printf("%d successfully inserted at the beginning\n", nn->data);
}
void insertend(void) {
    struct node *nn = (struct node *)malloc(sizeof(struct node));
    if (!nn) {
printf("Memory allocation failed.\n");
        return;
    }
printf("Enter data: ");
scanf("%d", &nn->data);
    nn->next = NULL;     if (start == NULL) {
        start = nn;
    } else {
        struct node *lp = start;
        while (lp->next != NULL) {
            lp = lp->next;
```

```
  }
        lp->next = nn;
    }
printf("%d successfully inserted at the end\n", nn->data);
}
void insertmid(void) {
    struct node *nn = (struct node *)malloc(sizeof(struct node));
    if (!nn) {
printf("Memory allocation failed.\n");
        return;
    }
printf("Enter data before which number to be inserted: ");
    int x;
scanf("%d", &x);
    if (start == NULL) {
printf("The list is empty.\n");
        return;
    }
    if (x == start->data) {
insertbeg();
        return;
    }
     struct node *ptemp = start;
    struct node *temp = start->next;
    while (temp != NULL && temp->data != x) {
        ptemp = temp;
        temp = temp->next;
    }
        if (temp == NULL) {
printf("%d does not exist\n", x);
        free(nn);
        return
  }
```

```
printf("Enter data: ");
scanf("%d", &nn->data);
    ptemp->next = nn;
    nn->next = temp;
printf("%d successfully inserted before %d\n", nn->data, x);
}
void deletion(void) {
    if (start == NULL) {
printf("The list is empty.\n");
        return;
        int x;
printf("Enter data to be deleted: ");
scanf("%d", &x);
    struct node *pt = NULL;
    struct node *t = start;
    if (t->data == x) {
        start = start->next;
        free(t);
printf("%d successfully deleted\n", x);
        return;
    }
        while (t != NULL && t->data != x)
{
        pt = t;
        t = t->next;
    }
        if (t == NULL) {
printf("%d does not exist\n", x);
        return;
    }
    pt->next = t->next;
    free(t);
printf("%d successfully deleted\n", x);
```

```
}
void display(void) {
    struct node *temp = start;
    if (temp == NULL) {
printf("The list is empty.\n");
        return;
    }
printf("Elements are:\n");
    while (temp != NULL) {
printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
int main() {
    int c, a;
    do {
printf("1: Insert\n2: Delete\n3: Display\n4: Exit\nEnter choice: ");
scanf("%d", &c);
        switch (c) {
            case 1:
printf("1: Insert at Beginning\n2: Insert at End\n3: Insert in Middle\nEnter choice: ");
scanf("%d", &a);
                switch (a) {
                    case 1: insertbeg(); break;
                    case 2: insertend(); break;
                    case 3: insertmid(); break;
                    default: printf("Invalid choice.\n"); break;
                }
                break;
            case 2: deletion(); break;
            case 3: display(); break;
            case 4: printf("Program ends\n"); break;
            default: printf("Wrong choice.\n"); break;
        }
    } while (c != 4);
      return 0;
}
```

OUTPUT:



```
[geetha@iare ~]$ gcc week1.c
[geetha@iare ~]$ ./a.out
1:insert
 2:delete
 3:display
 4:exit
 enter choice:1
1:insertbeg
 2:insertend
3:insertmid
enter choice:1
enter data:30
30 succ inserted
1:insert
 2:delete
 3:display
 4:exit
 enter choice:1
1:insertbeg
 2:insertend
3:insertmid
enter choice:1
enter data:20
20 succ inserted
```

RESULT:

Thus a C program to implement the List ADT using linked list is written and executed successfully.

Ex.No:3(b) **STACK – LINKED LIST IMPLEMENTATION**

DATE :

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node {
   int data;
   struct node *link;
} n;
n *top = NULL;
void push();
void pop();
void display();
int main() {
   int choice;
printf("STACK USING LINKED LIST\n1. PUSH\n2. POP\n3. DISPLAY\n4.
EXIT\n");
      do {
printf("\nEnter your choice: ");
scanf("%d", &choice);
        switch(choice) {
          case 1:
push();
             break;
          case 2:
pop();
             break;
          case 3:
display();
             break;
          case 4:
printf("Exiting...\n");
             break;
```

```
            default:
printf("Invalid choice\n");
                break;
        }
    } while(choice != 4);
    return 0;
}
void push() {
    int item;
    n *temp = (n*)malloc(sizeof(n));
    if (temp == NULL) {
printf("Memory allocation failed.\n");
        return;
    }
printf("Enter the item to push: ");
scanf("%d", &item);
    temp->data = item;
    temp->link = top;
    top = temp;
printf("%d pushed onto the stack\n", item);
}
void pop() {
    n *temp;
    if (top == NULL) {
printf("Stack is empty\n");
        return;
    }
    temp = top;
printf("The element popped = %d\n", temp->data);
    top = top->link;
    free(temp);
}
```

```
void display() {
    n *save = top;
    if (top == NULL) {
printf("Stack is empty\n");
        return;
    }
printf("The elements of the stack are: ");
    while (save != NULL) {
printf("%d\t", save->data);
        save = save->link;
    }
printf("\nTopmost element = %d\n", top->data);
}
```

OUTPUT:

```
STACK USING LINKED LIST
1.PUSH
2.POP
3.DISPLAY
4.EXIT
Enter your choice 1
Enter the item 10
Enter your choice 1
Enter the item 20
Enter your choice 3
The elements of the stack are :20 10
Topmost element = 20
Enter your choice 2
The element deleted = 20
Enter your choice 4
```

Ex.No :3(c) **QUEUE – LINKED LIST IMPLEMENTATION**

DATE :

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>G
struct node {
    int data;
    struct node* next;
};
struct node* front = NULL;
struct node* rear = NULL;
void enqueue(int value) {
    struct node* temp = (struct node*)malloc(sizeof(struct node));
    if (temp == NULL) {
printf("Memory allocation failed.\n");
        return;
    }
    temp->data = value;
    temp->next = NULL;
    if (rear == NULL) {
        front = rear = temp;
    } else {
        rear->next = temp;
        rear = temp;
    }
printf("%d enqueued to the queue\n", value);
}
void dequeue() {
    if (front == NULL) {
printf("Queue is empty\n");
        return;
    }
    struct node* temp = front;
    front = front->next;
    if (front == NULL) {
        rear = NULL;
    }
printf("Dequeued element: %d\n", temp->data);
    free(temp);
}
void display() {
    if (front == NULL) {
printf("Queue is empty\n");
        return;
    }
    struct node* temp = front;
printf("Elements in Queue: ");
    while (temp != NULL) {
printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
int main() {
    int choice;
printf("<----- Queue using Linked List ----->\n");
printf("1. Enqueue an element\n");
printf("2. Dequeue an element\n");
printf("3. Display Queue\n");
printf("4. Exit\n");
    while (1) {
printf("Enter your choice: ");
scanf("%d", &choice);
        switch (choice) {
            case 1: {
                int value;
printf("Enter a value to enqueue: ");
scanf("%d", &value);
                enqueue(value);
                break;
            }
            case 2:
dequeue();
                break;
            case 3:
display();
                break;
            case 4:
printf("Exiting...\n");
exit(0);
            default:
printf("Invalid choice, please try again.\n");
        }
    }
    return 0;
}
```

OUTPUT:

```
<-----Queue using Linked List----->
1. Enqueue an element
2. Dequeue an element
3. Display Queue
4. Exit
Enter your choice: 1
Enter a value to Enqueue: 10
Elements in Queue: 10
Enter your choice: 1
Enter a value to Enqueue: 20
Elements in Queue: 10 20
```

```
Enter your choice: 1
Enter a value to Enqueue: 30
Elements in Queue: 10 20 30
Enter your choice: 2
Elements in Queue: 20 30
Enter your choice: 4
```

RESULT:

Thus a C program to implement the queue, stack, queue using linked list is written and executed successfully.

Ex.No:4

POLYNOMIALMANIPULATION

DATE :

AIM:

To write a C program to perform polynomial manipulation using linked list.

ALGORITHM:

Step-1: Start the program.

Step-2: Declare the node as link.

Step-3: Allocate memory space for pol1, pol2 and poly.

Step-4: Read pol1 and pol2.

Step-5: Call the function polyadd.

Step-6: Add the co-efficients of poly1 and poly2 having the equal power and store it in poly.

Step-7: If there is no co-efficient having equal power in poly1 and poly2, then add it to poly.

Step-8: Display the poly1, poly2 and poly.

Step-9: Stop the program.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
struct link {
    int coeff;
    int pow;
    struct link *next;
};
void create(struct link **node);
void display(struct link *node);
struct link* polyadd(struct link *poly1, struct link *poly2);
int main() {
    struct link *poly1 = NULL, *poly2 = NULL, *result = NULL;
printf("\nEnter the First Polynomial:");
    create(&poly1);
printf("\nEnter the Second Polynomial:");
    create(&poly2);
   result = polyadd(poly1, poly2)
printf("\nFirst Polynomial: ");
    display(poly1);
printf("\nSecond Polynomial: ");
    display(poly2);
printf("\nAddition of Two Polynomials: ");
    display(result);
    return 0;
}
void create(struct link **node) {
    char ch;
    struct link *temp;
    struct link *last = NULL;
    do {
        temp = (struct link *)malloc(sizeof(struct link));
        if (temp == NULL) {
```

```
printf("Memory allocation failed.\n");
            return;
        }
printf("\nEnter Coefficient: ");
scanf("%d", &temp->coeff);
printf("Enter Power: ");
scanf("%d", &temp->pow);
        temp->next = NULL;
        if (*node == NULL) {
            *node = temp;
        } else {
            last->next = temp;
        }
        last = temp;
printf("Continue (y/n): ");
getchar();
        ch = getchar();
    } while (ch == 'y' || ch == 'Y');
}
void display(struct link *node) {
    if (node == NULL) {
printf("No terms in polynomial.");
        return;
    }
while (node != NULL) {
printf("%dx^%d", node->coeff, node->pow);
        node = node->next;
        if (node != NULL) {
printf(" + ");
        }
    }
    printf("\n");
}
```

```
        struct link* polyadd(struct link *poly1, struct link *poly2) {
        struct link *result = NULL, *last = NULL;
        while (poly1 != NULL && poly2 != NULL) {
        struct link *temp = (struct link *)malloc(sizeof(struct link));
         if (temp == NULL) {
printf("Memory allocation failed.\n");
      return NULL;
        }
        if (poly1->pow > poly2->pow) {
            temp->coeff = poly1->coeff;
            temp->pow = poly1->pow;
            poly1 = poly1->next;
        } else if (poly1->pow < poly2->pow) {
            temp->coeff = poly2->coeff;
            temp->pow = poly2->pow;
            poly2 = poly2->next;
        } else {
           temp->coeff = poly1->coeff + poly2->coeff;
            temp->pow = poly1->pow;
            poly1 = poly1->next;
            poly2 = poly2->next;
        }
        temp->next = NULL;        if (result == NULL) {
            result = temp;
        } else {
            last->next = temp;
        }
        last = temp;
    }
        while (poly1 != NULL) {
        struct link *temp = (struct link *)malloc(sizeof(struct link));
        if (temp == NULL) {
printf("Memory allocation failed.\n");
```

```
            return NULL;
        }
        temp->coeff = poly1->coeff;
        temp->pow = poly1->pow;
        temp->next = NULL;
        if (result == NULL) {
            result = temp;
        } else {
            last->next = temp;
        }
        last = temp;
        poly1 = poly1->next;
    }
    while (poly2 != NULL) {
        struct link *temp = (struct link *)malloc(sizeof(struct link));
        if (temp == NULL) {
printf("Memory allocation failed.\n");
            return NULL;
        }
        temp->coeff = poly2->coeff;
        temp->pow = poly2->pow;
        temp->next = NULL;
        if (result == NULL) {
            result = temp;
        } else {
            last->next = temp;
        }
        last = temp;
        poly2 = poly2->next;
    }
    return result;
}
```

OUTPUT:

```
Enter the First Polynomial:
Enter Coeff:3
Enter Power:3
Continue(y/n):y
Enter Coeff:6
Enter Power:2
Continue(y/n):y
Enter Coeff:9
Enter Power:1
Continue(y/n):y
Enter Coeff:8
Enter Power:0
Continue(y/n):n
Enter the Second Polynomial:
Enter Coeff:76
Enter Power:3
Continue(y/n):y
Enter Coeff:43
Enter Power:2
Continue(y/n):y
Enter Coeff:23
Enter Power:1
Continue(y/n):y
Enter Coeff:24
Enter Power:0
Continue(y/n):n
First Polynomial: 3x^3+6x^2+9x^1+8x^0
Second Polynomial: 76^3+43x^2+23x^1+24x^0
Addition of two Polynomials: 79^3+49x^2+32x^1+32x^0
```

RESULT:

Thus a C program to perform polynomial manipulation using linked list is written and executed successfully.

Ex.No:5(a)

EVALUATING POSTFIX EXPRESSION

DATE :

AIM:

To write a C program to evaluating postfix expression

ALGORITHM:

Step-1: Start the program.

Step-2: Create an empty stack to hold operands.

Step-3: Process the expression from left to right.

Step-4: If the token is an operand (number), push it onto the stack.

If the token is an operator (like +, -, *, /):

Pop the top two operands from the stack.

Apply the operator to these operands. The first operand popped is the right operand, and the second popped is the left operand.

Push the result back onto the stack.

Step-5: Stop the program.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define SIZE 50
int stack[SIZE];
int top = -1;
void push(int elem) {
    if (top < SIZE - 1) {
        stack[++top] = elem;
    } else {
printf("Stack overflow\n");
    }
}
int pop() {
    if (top >= 0) {
        return stack[top--];
    } else {
printf("Stack underflow\n");
        return 0;
    }
}
int main() {
    char pofx[SIZE], ch;
    int i = 0, op1, op2;
printf("<-----Stack Application: Evaluating Postfix Expression----->\n");
printf("Read the Postfix Expression: ");
scanf("%s", pofx);
    while ((ch = pofx[i++]) != '\0') {
        if (isdigit(ch)) {
          push(ch - '0'); // Convert char to int
        } else {
            op2 = pop();
            op1 = pop();
            switch (ch) {
                case '+':
push(op1 + op2);
                    break;
                case '-':
push(op1 - op2);
                    break;
                case '*':
push(op1 * op2);
                    break;
                case '/':
                    if (op2 != 0) {
push(op1 / op2);
                    } else {
printf("Error: Division by zero\n");
                        return -1;
                    }
                    break;
                default:
printf("Error: Invalid operator %c\n", ch);
                    return -1;
            }
        }
    }
printf("\nGiven Postfix Expression: %s\n", pofx);
printf("Result after Evaluation: %d\n", pop());item left in the stack
    return 0;
}
```

OUTPUT:

```
<-----Stack Application: Evaluating Postfix Expression----->
Read the Postfix Expression ? 456*+7-
Given Postfix Expn: 456*+7-
Result after Evaluation: 27
```

RESULT:

Thus a C program to evaluate the postfix expression is written and executed successfull

Ex.No:5(b) **INFIX TO POSTFIX CONVERSION**

DATE:
PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define SIZE 50
char stack[SIZE];
int top = -1;
void push(char elem) {
    if (top < SIZE - 1) {
        stack[++top] = elem;
    } else {
printf("Stack overflow\n");
    }
}
char pop() {
    if (top >= 0) {
        return stack[top--];
    } else {
printf("Stack underflow\n");
        return '\0';    }
}
int precedence(char elem) {
    switch (elem) {
        case '#': return 0;
        case '(': return 1;
        case '+':
        case '-': return 2;
        case '*':
        case '/': return 3;
        default: return -1;
    }
```

```
}
int main() {
    char infix[SIZE], postfix[SIZE], ch, elem;
    int i = 0, k = 0;
printf("<-----Stack Application: Infix to Postfix Conversion----->\n");
printf("Enter the Infix Expression: ");
scanf("%s", infix);
push('#'); // Sentinel value to indicate the bottom of the stack
    while ((ch = infix[i++]) != '\0') {
        if (ch == '(') {
            push(ch); // Push '(' onto the stack
        } else if (isalnum(ch)) {
            postfix[k++] = ch;
        } else if (ch == ')') {
            while (stack[top] != '(') {
                postfix[k++] = pop();
            }
pop();
(' from stack')
        } else {
            while (precedence(stack[top]) >= precedence(ch)) {
                postfix[k++] = pop();
            }
            push(ch); // Push current operator onto stack
        }
    }
    while (stack[top] != '#') {
        postfix[k++] = pop();
    }
    postfix[k] = '\0'; // Null-terminate the postfix string
printf("\nGiven Infix Expression: %s\nPostfix Expression: %s\n", infix, postfix);
     return 0;
}
```

OUTPUT:

```
<-----Stack Application: Infix to Postfix Conversion----->
Read the Infix Expression ? a+b*c-d
Given Infix Expn: a+b*c-d Postfix Expn: abc*+d-
```

RESULT:

Thus a C program to convert the expression in infix to postfix, infix to postfix conversion is written and executed successfully

Ex.No:6

BINARY SEARCH TREE

DATE :

AIM:

To write a C program to implement binary search tree.

ALGORITHM:

Step-1: Start the program.

Step-2: Declare the node.

Step-3: Read the elements to be inserted.

Step-4: Create the binary search tree.

Step-5: Read the element to be searched.

Step-6: Visit the nodes by in order.

Step-7: Find the searching node and display if it is present with parent node.

Step-8: Read the element to be removed from BST.

Step-9: Delete that node from BST.

Step-10: Display the binary search tree by in order.

Step-11: Stop the program.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
Type def  struct  bst {
    int data;
    struct bst *left, *right;
} node;
void insert(node **root, node *new node);
void in order(node *temp);
node *search(node *root, int key, node **parent);
void del(node **root, int key);
node *getnode();
int main() {
    int choice;
    char ans = 'N';
    int key;
    node *newnode, *root = NULL, *temp, *parent;
    do {
printf("\n\t Program for Binary Search Tree");
printf("\n\t 1. Create");
printf("\n\t 2. Search");
printf("\n\t 3. Delete");
printf("\n\t 4. Display");
printf("\n\t 5. Exit");
printf("\n\t Enter your Choice: ");
scanf("%d", &choice);
        switch (choice) {
            case 1:
                do {
                    newnode = getnode();
printf("\n\t Enter the element: ");
scanf("%d", &newnode->data);
insert(&root, newnode);
printf("\n\t Do you want to enter more elements? (y/n): ");
getchar();
                    ans = getchar();
                } while (ans == 'y' || ans == 'Y');
                break;

            case 2:
printf("\n\t Enter the element to be searched: ");
scanf("%d", &key);
                temp = search(root, key, &parent);
                if (temp != NULL) {
printf("\n\t The element %d is present. Parent is %d\n", temp->data, parent ?
parent->data : -1);
                } else {
printf("\n\t Element not found.\n");
                }
                break;
            case 3:
printf("\n\t Enter the element to be deleted: ");
scanf("%d", &key);
del(&root, key);
                break;

            case 4:
                if (root == NULL) {
printf("\n\t Tree is not created.\n");
                } else {
printf("\n The Tree is: ");
                    inorder(root);
                }
                break;
   case 5:
printf("\n\t Exiting...\n");
                break;
            default:
printf("\n\t Invalid choice. Please try again.\n");
        }
    } while (choice != 5);
    return 0;
}
node *getnode() {
    node *temp = (node *)malloc(sizeof(node));
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}
void insert(node **root, node *newnode) {
    if (*root == NULL) {
        *root = newnode;
    } else if (newnode->data < (*root)->data) {
        insert(&(*root)->left, newnode);
    } else {
        insert(&(*root)->right, newnode);
    }
}
void inorder(node *temp) {
    if (temp != NULL) {
        inorder(temp->left);
printf(" %d", temp->data);
        inorder(temp->right);
    }
}
node *search(node *root, int key, node **parent) {
    node *temp = root;
    *parent = NULL; // Initialize parent
    while (temp != NULL) {
        if (temp->data == key) {
            return temp;
        }
        *parent = temp;
        temp = (temp->data > key) ? temp->left : temp->right;
    }
    return NULL;
}
void del(node **root, int key) {
    node *temp, *parent, *successor;
    temp = search(*root, key, &parent);
    if (temp == NULL) {
printf("Element not found.\n");
        return;
    }
      if (temp->left && temp->right) {
        successor = temp->right;
        while (successor->left != NULL) {
            parent = successor;
            successor = successor->left;
        }
        temp->data = successor->data;
        del(&(temp->right), successor->data);
    }
        else {
        node *child = (temp->left) ? temp->left : temp->right;
        if (parent == NULL) {
            *root = child;
        } else if (parent->left == temp) {
            parent->left = child;
        } else {
            parent->right = child;
        }
        free(temp);
```

```
printf("Node deleted.\n");
    }
}
```

OUTPUT:

```
Program for Binary Search Tree
1.Create
2.Search
3.Delete
4.Display
5.Exit
Enter your Choice:1
Enter the element:10
Do u want to enter more elements?(y/n):y
Enter the element:8
Do u want to enter more elements?(y/n):y
Enter the element:9
Do u want to enter more elements?(y/n):y
Enter the element:7
Do u want to enter more elements?(y/n):y
Enter the element:15
Do u want to enter more elements?(y/n):y
Enter the element:13
Do u want to enter more elements?(y/n):y
Enter the element:14
Do u want to enter more elements?(y/n):y
Enter the element:12
Do u want to enter more elements?(y/n):y
Enter the element:16
Do u want to enter more elements?(y/n):n
1.Create
2.Search
3.Delete
```

```
4.Display
5.Exit
Enter your Choice:4
The Tree is: 7 8 9 10 12 13 14 15 16
1.Create
2.Search
3.Delete
4.Display
5.Exit
Enter your Choice:2
Enter the element to be searched:16
The 16 element is present
Parent of node 16 is 15
1.Create
2.Search
3.Delete
4.Display
5.Exit
Enter your Choice:5
```

RESULT:

Thus a C program to implement the binary search tree is written and executed successfully.

Ex.No :7

AVL TREE

DATE :

AIM:

To wr]ite a C program to implement an AVL tree.

ALGORITHM:

Step-1: Start the program.

Step-2: Declare the node.

Step-3: Read the elements and create a tree.

Insert:

Step-1: Insert a new node as new leaf node just as in ordinary binary search tree.

Step-2: Now trace the path from inserted node towards root. For each node, „n" encountered,

check if heights of left(n) and right(n) differ by atmost 1.

a) if yes, move towards parent(n).

b) Otherwise restructure the by doing either a single rotation or a double rotation.

Delete:

Step-1: Search the node to be deleted.

Step-2: If the node to be deleted is a leaf node, then simply make it NULL to remove.

Step-3: If the node to be deleted is not a leaf node, then the node must be swapped with

its inorder successor. Once the node is swapped, then remove the node.

Step-4: Traverse back up the path towards root, check the balance factor of every node

along the path.

Step-5: If there is unbalanced in some subtree then balance the subtree using

appropriate single or double rotation.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node {
    int data;
    struct node *left, *right;
    int ht;
} node;
node *insert(node *, int);
node *Delete(node *, int);
void preorder(node *);
void inorder(node *);
int height(node *);
node *rotateright(node *);
node *rotateleft(node *);
node *RR(node *);
node *LL(node *);
node *LR(node *);
node *RL(node *);
int BF(node *);
int main() {
    node *root = NULL;
    int x, n, i, op;
    do {
        printf("\n1) Create:");
        printf("\n2) Insert:");
        printf("\n3) Delete:");
        printf("\n4) Print:");
        printf("\n5) Quit:");
printf("\n\nEnter Your Choice: ");
scanf("%d", &op);
        switch (op) {
            case 1:
```

```
printf("\nEnter number of elements: ");
scanf("%d", &n);
printf("\nEnter tree data: ");
                root = NULL;
                for (i = 0; i < n; i++) {
scanf("%d", &x);
                    root = insert(root, x);
                }
                break;
            case 2:
printf("\nEnter a data: ");
scanf("%d", &x);
                root = insert(root, x);
                break;
            case 3:
printf("\nEnter a data: ");
scanf("%d", &x);
                root = Delete(root, x);
                break;
            case 4:
printf("\nPreorder sequence:\n");
                preorder(root);
printf("\nInorder sequence:\n");
                inorder(root);
                printf("\n");
                break;
            case 5:
                printf("\nExiting...\n");
                break;

            default:
printf("\nInvalid choice, please try again.\n");
        }
```

```
    } while (op != 5);
   return 0;
}
node *insert(node *T, int x) {
    if (T == NULL) {
        T = (node *)malloc(sizeof(node));
        T->data = x;
        T->left = NULL;
        T->right = NULL;
        T->ht = 1;  // Height of a new node is set to 1
    } else if (x > T->data) { // insert in right subtree
        T->right = insert(T->right, x);
        if (BF(T) == -2) {
            if (x > T->right->data)
                T = RR(T);
            else
                T = RL(T);
        }
    } else if (x < T->data) {
        T->left = insert(T->left, x);
        if (BF(T) == 2) {
            if (x < T->left->data)
                T = LL(T);
            else
                T = LR(T);
        }
    }
    T->ht = height(T);
    return T;
}
node *Delete(node *T, int x) {
    node *p;
    if (T == NULL) {
        return NULL;
    } else if (x > T->data) { // search in right subtree
        T->right = Delete(T->right, x);
        if (BF(T) == 2) {
            if (BF(T->left) >= 0)
                T = LL(T);
            else
                T = LR(T);
        }
    } else if (x < T->data) {
        T->left = Delete(T->left, x);
        if (BF(T) == -2) { // Rebalance during windup
            if (BF(T->right) <= 0)
                T = RR(T);
            else
                T = RL(T);
        }
    } else {
        if (T->right != NULL) {
            p = T->right;
            while (p->left != NULL)
                p = p->left;
            T->data = p->data;
            T->right = Delete(T->right, p->data);
            if (BF(T) == 2) {
                if (BF(T->left) >= 0)
                    T = LL(T);
                else
                    T = LR(T);
            }
        } else {
            node *temp = T->left;
            free(T);
            return temp;
        }
    }
    T->ht = height(T);
    return T;
}
int height(node *T) {
    if (T == NULL)
        return 0;
    int leftHeight = height(T->left);
    int rightHeight = height(T->right);
    return 1 + (leftHeight >rightHeight ? leftHeight : rightHeight);
}
node *rotateright(node *x) {
    node *y = x->left;
    x->left = y->right;
    y->right = x;
    x->ht = height(x);
    y->ht = height(y);
    return y;
}
node *rotateleft(node *x) {
    node *y = x->right;
    x->right = y->left;
    y->left = x;
    x->ht = height(x);
    y->ht = height(y);
    return y;
}
node *RR(node *T) {
    return rotateleft(T);
}
node *LL(node *T) {
    return rotateright(T);
}
node *LR(node *T) {
    T->left = rotateleft(T->left);
    return rotateright(T);
}
node *RL(node *T) {
    T->right = rotateright(T->right);
    return rotateleft(T);
}
int BF(node *T) {
    if (T == NULL)
        return 0;
    return (T->left ? T->left->ht : 0) - (T->right ? T->right->ht : 0);
}
void preorder(node *T) {
    if (T != NULL) {
        printf("%d(Bf=%d) ", T->data, BF(T));
        preorder(T->left);
        preorder(T->right);
    }
}
void inorder(node *T) {
    if (T != NULL) {
        inorder(T->left);
        printf("%d(Bf=%d) ", T->data, BF(T));
        inorder(T->right);
    }
}
```

OUTPUT:

```
1)Create:
2)Insert:
3)Delete:
4)Print:
5)Quit:
Enter Your Choice:1
Enter no. of elements:4
Enter tree data:7 12 4 9
1)Create:
2)Insert:
3)Delete:
4)Print:
5)Quit:
Enter Your Choice:4
Preorder sequence:
7(Bf=-1)4(Bf=0)12(Bf=1)9(Bf=0)
Inorder sequence:
4(Bf=0)7(Bf=-1)9(Bf=0)12(Bf=1)
1)Create:
2)Insert:
3)Delete:
4)Print:
5)Quit:
Enter Your Choice:3
Enter a data:7
1)Create:
2)Insert:
3)Delete:
4)Print:
5)Quit:
Enter Your Choice:4
Preorder sequence:
```

```
9(Bf=0)4(Bf=0)12(Bf=0)
Inorder sequence:
4(Bf=0)9(Bf=0)12(Bf=0)
1)Create:
2)Insert:
3)Delete:
4)Print:
5)Quit:
Enter Your Choice:5
```

RESULT:

Thus a C program to implement an AVL tree is written and executed successfully.

Ex.No:8

HEAP USING PRIORITY QUEUE

DATE :

AIM:

To write a C program to implement heap sort using priority queue.

ALGORITHM:

Step-1: Start the program.

Step-2: Read the elements to be inserted in heap.

Step-3: Insert the element one by one.

Step-4: Construct the heap structure to satisfy heap property of maxheap.

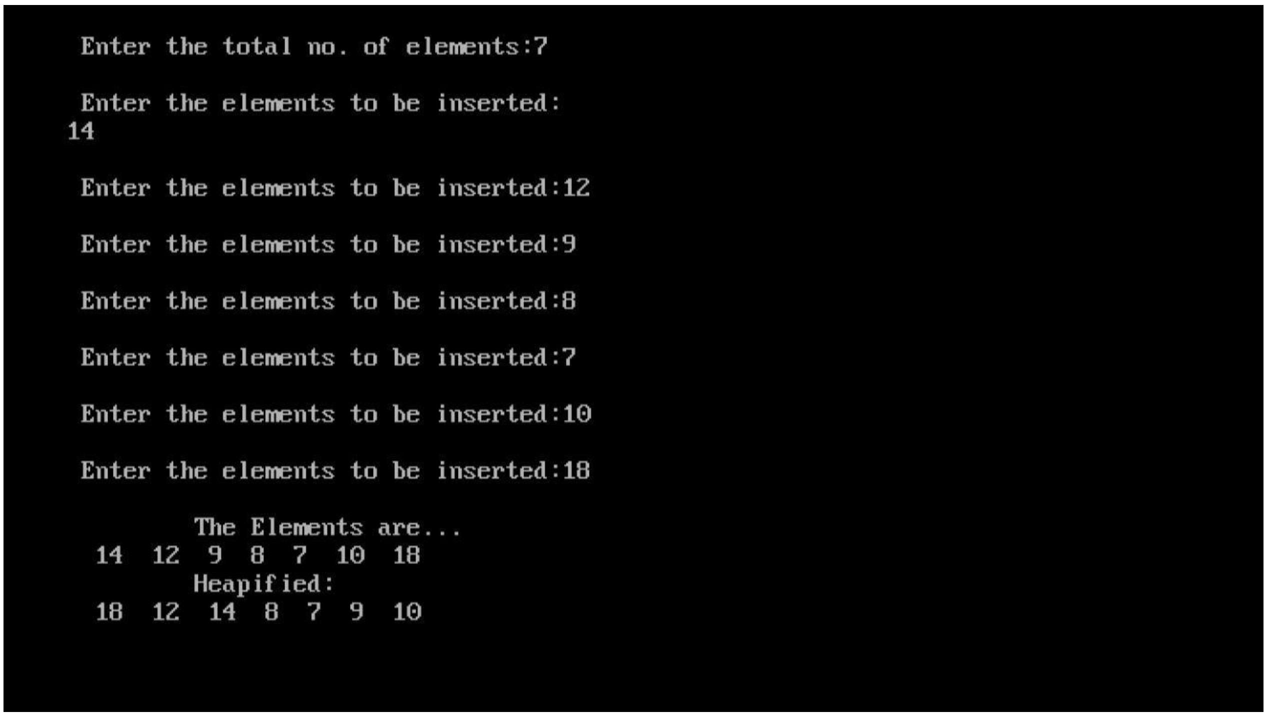
Step-5: Display the heapified structure.

Step-6: Stop the program.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
int arr[MAX];
int n = 0;
void insert(int num) {
    if (n < MAX) {
        arr[n] = num;
        n++;
    } else {
printf("\nArray is full\n");
    }
}
void heapify(int index) {
    int largest = index;
    int left = 2 * index + 1;  // Left child index
    int right = 2 * index + 2; // Right child index
    if (left < n && arr[left] > arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest])
        largest = right;
        if (largest != index) {
        int temp = arr[index];
        arr[index] = arr[largest];
        arr[largest] = temp;
        heapify(largest);
    }
}
void makeheap() {
    for (int i = (n / 2) - 1; i >= 0; i--) {
        heapify(i);
    }
}
void display() {
    printf("\n");
    for (int i = 0; i < n; i++) {
printf(" %d", arr[i]);
    }
    printf("\n");
}
int main() {
printf("\nEnter the total number of elements (max %d): ", MAX);
scanf("%d", &n);
    if (n <= 0 || n > MAX) {
printf("Error: Number of elements must be between 1 and %d.\n", MAX);
        return 1;
    }
    for (int i = 0; i < n; i++) {
        int item;
printf("\nEnter the element to be inserted: ");
scanf("%d", &item);
        insert(item);
    }
printf("\n\tThe Elements are...");
display();
makeheap();
    printf("\n\tHeapified:");
display();
    return 0;
}
```

OUTPUT:



```
 Enter the total no. of elements:7

 Enter the elements to be inserted:
14

 Enter the elements to be inserted:12

 Enter the elements to be inserted:9

 Enter the elements to be inserted:8

 Enter the elements to be inserted:7

 Enter the elements to be inserted:10

 Enter the elements to be inserted:18

         The Elements are...
  14  12  9  8  7  10  18
         Heapified:
  18  12  14  8  7  9  10
```

RESULT:

Thus a C program to implement heap sort using priority queue is written and executed successfully.

Ex.No :9 **DIJKTRA ALGORITHM**

DATE :

AIM:

The aim of Dijkstra's algorithm is to find the shortest path from a starting node to all other nodes in a weighted graph, minimizing the total path cost.

ALGORITHM:

Step 1 : Set the distance to the starting node as 0 and all other nodes as infinity.

Step 2: Place all nodes in a priority queue, ordered by their current shortest distance.

Step3 : Remove the node with the smallest distance from the queue (let's call this node `current`).

Step 4: Continue processing nodes until all nodes are visited, and the shortest paths from the start node to all reachable nodes are determined.

Step 5: The algorithm ends when all nodes have been processed, and the shortest path from the start node to every other node is stored.

PROGRAM:

```
#include <stdio.h>
#include <limits.h>
#define V 5
int minDistance(int dist[], int sptSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++) {
        if (sptSet[v] == 0 && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    }
    return min_index;
}
void printSolution(int dist[], int n) {
    printf("Vertex\tDistance from Source\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\n", i, dist[i]);
    }
}
void dijkstra(int graph[V][V], int src) {
    int dist[V];
    int sptSet[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = 0;
    }
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = 1;
        for (int v = 0; v < V; v++) {
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v]
< dist[v]) {
```

```
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }
    printSolution(dist, V);
}
int main() {
from node i to j
    int graph[V][V] = {
        {0, 10, 0, 0, 0},
        {10, 0, 5, 0, 0},
        {0, 5, 0, 20, 1},
        {0, 0, 20, 0, 2},
        {0, 0, 1, 2, 0}
    };
int start_node = 0;
    dijkstra(graph, start_node);
    return 0;
}
```

OUTPUT:

```
Vertex Distance from Source
0          0
1          10
2          15
3          18
4          16
```

RESULT:

Thus a C program to implement graph traversals by Breadth First Search and Depth First Search is written and executed successfully.

Ex.No :10

PRIMS ALGORITHM

DATE :

AIM:

The aim of Prim's algorithm is to find a minimum spanning tree that connects all nodes in a weighted, undirected graph with the minimum total edge weight and no cycles.

ALGORITHM:

Step 1: Start with an arbitrary node, setting it as part of the growing minimum spanning tree (MST)

Step 2 : Choose the edge with the smallest weight that connects a visited node to an unvisited no

Step 3: For the newly added node, update the edge weights for its unvisited neighbors.

Step 4: Continue selecting the smallest edge connecting a visited node to an unvisited node, adding nodes to the MST until all nodes are included.

Step 5: The algorithm ends when all nodes have been visited and are connected by the minimum set of edges with the least possible total weight, forming a minimum spanning tree.

PROGRAM:

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
#define V 5
int minKey(int key[], bool mstSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++) {
        if (!mstSet[v] && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }
    return min_index;
}
void primMST(int graph[V][V]) {
    int parent[V];
    int key[V];
    bool mstSet[V];
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = false;
    }
    key[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < V - 1; count++) {
                int u = minKey(key, mstSet);
        mstSet[u] = true;
        for (int v = 0; v < V; v++) {
            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++) {
        printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);
    }
}
int main() {
        int graph[V][V] = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };
 primMST(graph);
return 0;
}
```

OUTPUT

```
Edge  Weight
0 - 1  2
1 - 2  3
0 - 3  6
1 - 4  5
```

RESULT:

Thus a C program to perform topological sorting is written and executed successfully.

Ex.No:11 # LINEAR & BINARY SEARCH

DATE :

AIM:

To write a C program to perform linear search and binary search.

ALGORITHM:

Linear Search:

Step-1: Read n numbers and search value.

Step-2: If search value is equal to first element then print value is found.

Step-3: Else search with the second element and so on.

Binary Search:

Step-1: Read n numbers and search value.

Step-2: If search value is equal to middle element then print value is found.

Step-3: If search value is less than middle element then search left half of list with the same method.

Step-4: Else search right half of list with the same method.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
void main() {
   int a[100], i, n, item, s, ch, beg, end, mid;
printf("Enter number of elements: ");
scanf("%d", &n);
printf("\nEnter elements in sorted order:\n");
   for (i = 0; i < n; i++) {
scanf("%d", &a[i]);
   }
   while (1) {
printf("\n1. Linear Search\n2. Binary Search\n3. Exit\n");
printf("Enter your choice: ");
scanf("%d", &ch);
      switch (ch) {
         case 1:
printf("<----- LINEAR SEARCH ----->\n");
printf("\nEnter element you want to search: ");
scanf("%d", &item);
            s = 0;
            for (i = 0; i < n; i++) {
               if (a[i] == item) {
printf("\nData is found at location: %d\n", i + 1);
                  s = 1;
                  break;
               }
            }
            if (s == 0) {
printf("Data is not found.\n");
            } break;
         case 2:
printf("<----- BINARY SEARCH ----->\n");
printf("\nEnter item you want to search: ");
scanf("%d", &item);
                beg = 0;
                end = n - 1;
                mid = (beg + end) / 2;
                while (beg <= end) {
                    mid = (beg + end) / 2;
                    if (a[mid] == item) {
printf("\nData is found at location: %d\n", mid + 1);
                        break;
                    } else if (a[mid] < item) {
                        beg = mid + 1;
                    } else {
                        end = mid - 1;
                    }
                }
                if (beg > end) {
printf("Data is not found.\n");
                }
                break;

            case 3:
exit(0);
            default:
printf("Invalid choice. Please try again.\n");
        }
    }
}
```

OUTPUT:

```
Enter No. of Elements:
5
Enter Elements:
2 4 3 5 1
1.Linear Search
2.Binary Search
3.Exit
Enter your choice: 1
<-----LINEAR SEARCH----->
Enter Element you want to Search: 1
Data is Found at Location : 5
1.Linear Search
2.Binary Search
3.Exit
Enter your choice: 2
<-----BINARY SEARCH----->
Enter Item you want to Search: 3
Data is Found at Location : 3
1.Linear Search
2.Binary Search
3.Exit
Enter your choice: 3
```

RESULT:

Thus a C program to implement the linear search and binary search is written and executed successfully.

Ex.No:12 (a) **INSERTION SORT & SELECTION SORT**

DATE :

AIM:

To write a c program to create hash table and collision handling by linear probing.

ALGORITHM:

Step 1: Start with the second element (assuming the first element is already sorted).

Step 2: Compare the current element with the previous elements.

Step 3: Shift all elements greater than the current element one position to the right.

Step 4:Insert the current element into its correct position.

Step 5: Stop the program

PROGRAM:

```
#include <stdio.h>
void insertion Sort(int arr[], int n) {
int i, key, j;
for (i = 1; i < n; i++) {
key = arr[i];
j = i - 1;
while (j >= 0 && arr[j] > key) {
arr[j + 1] = arr[j];
j = j - 1;
}
arr[j + 1] = key;
}
}
void printArray(int arr[], int n) {
for (int i = 0; i < n; i++) {
printf("%d ", arr[i]);
}
printf("\n");
}
int main() {
int arr[] = {12, 11, 13, 5, 6};
int n = sizeof(arr) / sizeof(arr[0]);
printf("Original Array: ");
printArray(arr, n);
insertionSort(arr, n);
printf("Sorted Array: ");
printArray(arr, n);
return 0;
}
```

Output:

```
Original Array: 12 11 13 5 6
Sorted Array: 5 6 11 12 13
```

Ex.No:12(b) **SELECTION SORT**

DATE:

PROGRAM:

```
#include <stdio.h>
void selectionSort(int arr[], int n) {
int i, j, minIndex, temp;
for (i = 0; i < n - 1; i++) {
minIndex = i;
for (j = i + 1; j < n; j++) {
if (arr[j] < arr[minIndex]) {
minIndex = j;
}
}
temp = arr[i];
arr[i] = arr[minIndex];
arr[minIndex] = temp;
}
}
void printArray(int arr[], int n) {
for (int i = 0; i < n; i++) {
printf("%d ", arr[i]);
}
printf("\n");
}
int main() {
int arr[] = {64, 25, 12, 22, 11};
int n = sizeof(arr) / sizeof(arr[0]);
printf("Original Array: ");
printArray(arr, n);
selectionSort(arr, n);
printf("Sorted Array: ");
printArray(arr, n);
return 0;
}
```

Output:

```
Original Array: 64 25 12 22 11
Sorted Array: 11 12 22 25 64
```

RESULT:

Thus a C program to implement the insertion and selection sort is written and executed successfully.

Ex.No:13

MERGE SORT

DATE :

AIM:

To write a c program to create hash table and collision handling by merge sort

ALGORITHM:

Step 1: Divide: Split the unsorted list into two halves.

Step 2: Conquer: Recursively sort both halves.

Step 3: Combine: Merge the two sorted halves to produce a sorted array

PROGRAM

```
#include <stdio.h>
void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;
   int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
    i = 0;
    j = 0;
    k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
void merge Sort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }   }
void print Array(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int arrSize = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: \n");
    printArray(arr, arrSize);
    mergeSort(arr, 0, arrSize - 1);
    printf("Sorted array: \n");
    printArray(arr, arrSize);
    return 0;
}
```

OUTPUT:

```
Original array:
38 27 43 3 9 82 10
Sorted array:
3 9 10 27 38 43 82
```

RESULT:

Thus a C program to create hash table and merge sort by is written and executed successfully.

Ex.No:14(a) **IMPLEMENTATION OF OPEN ADDRESSING**

DATE :

AIM:

Open Addressing in hashing is to resolve collisions that occur when two or more keys hash

to the same index in the hash table. Instead of using separate chains to store colliding

elements, open addressing resolves the collision by finding another available spot within the

table through a process called probing

ALGORITHM:

Insertion Algorithm:

Step 1: Hash the key to find the index in the hash table using the hash function.

Step 2: If the calculated index is empty, insert the key at that index.

Step 3: If the index is occupied (collision occurs), move to the next index in a linear manner (index+ 1).

Step 4: Repeat Step 3 until an empty index is found (wrap around to the beginning of the table if necessary).

Step 5: Insert the key at the first available empty index.

Search Algorithm:

Step 1: Hash the key to determine the initial index in the table.

Step 2: Check if the key at the computed index matches the search key.

Step 3: If the key at the current index matches, return the index.

Step 4: If there is a mismatch, move to the next index (index + 1), and continue searching until either the key is found or an empty slot is encountered.

Step 5: If an empty slot is encountered and the key has not been found, the key is not present in the

table.

Deletion Algorithm:

Step 1: Search for the key using the search algorithm described above.

Step 2: If the key is found, mark the slot as deleted.

Step 3: After deletion, rehash the subsequent keys in the probe sequence (if any), since they may have been displaced due to previous collisions.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 10
typedef struct {
int key;
int isOccupied;
} HashEntry;
void initializeTable(HashEntry hashTable[]) {
for (int i = 0; i < TABLE_SIZE; i++) {
hashTable[i].isOccupied = 0;
}
}
int hash(int key) {
return key % TABLE_SIZE;
}
void insert(HashEntry hashTable[], int key) {
int index = hash(key);
while (hashTable[index].isOccupied == 1) {
index = (index + 1) % TABLE_SIZE;
}
hashTable[index].key = key;
hashTable[index].isOccupied = 1;
printf("Inserted %d at index %d\n", key, index);
}
int search(HashEntry hashTable[], int key) {
int index = hash(key);
int originalIndex = index;
while (hashTable[index].isOccupied == 1) {
if (hashTable[index].key == key) {
return index;
}
index = (index + 1) % TABLE_SIZE;
if (index == originalIndex) {
break;
}
}
return -1;
}
void delete(HashEntry hashTable[], int key) {
int index = search(hashTable, key);
if (index == -1) {
printf("Key %d not found for deletion\n", key);
} else {
hashTable[index].isOccupied = 0;
printf("Deleted %d from index %d\n", key, index);
}
}
void display(HashEntry hashTable[]) {
for (int i = 0; i < TABLE_SIZE; i++) {
if (hashTable[i].isOccupied) {
printf("Index %d: %d\n", i, hashTable[i].key);
} else {
printf("Index %d: Empty\n", i);
}
}
}
int main() {
HashEntry hashTable[TABLE_SIZE];
initializeTable(hashTable);
insert(hashTable, 15);
insert(hashTable, 25);
insert(hashTable, 35);
insert(hashTable, 45);
printf("\nHash Table after insertions:\n");
display(hashTable);
int searchKey = 25;
int searchResult = search(hashTable, searchKey);
if (searchResult != -1) {
printf("\nFound key %d at index %d\n", searchKey, searchResult);
} else {
printf("\nKey %d not found\n", searchKey);
}
delete(hashTable, 25);
printf("\nHash Table after deletion:\n");
display(hashTable);
searchResult = search(hashTable, searchKey);
if (searchResult != -1) {
printf("\nFound key %d at index %d\n", searchKey, searchResult);
} else {
printf("\nKey %d not found\n", searchKey);
}
return 0;
}
```

Output:

```
Inserted 15 at index 5
Inserted 25 at index 5
Inserted 35 at index 6
Inserted 45 at index 7
Hash Table after insertions:
Index 0: Empty
Index 1: Empty
Index 2: Empty
Index 3: Empty
Index 4: Empty
Index 5: 15
Index 6: 35
Index 7: 45
Index 8: Empty
Index 9: Empty
Found key 25 at index 5
Key 25 not found for deletion
Hash Table after deletion:
Index 0: Empty
Index 1: Empty
Index 2: Empty
Index 3: Empty
Index 4: Empty
Index 5: Empty
Index 6: 35
Index 7: 45
Index 8: Empty
Index 9: Empty
Key 25 not found
```

Ex.No:14 (b) **LINEAR POBING**
DATE:

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
int create(int num);
void linearprob(int a[], int key, int num);
void display(int a[]);
int main() {
    int a[MAX], num, key, i;
    char ans;
    for (i = 0; i < MAX; i++)
        a[i] = -1;
printf("\nCOLLISION HANDLING BY LINEAR PROBING");
    do {
printf("\nEnter the number: ");
scanf("%d", &num);
        key = create(num);
linearprob(a, key, num);
printf("\nDo you wish to continue? (y/n): ");
getchar();
        ans = getchar();
    } while (ans == 'y' || ans == 'Y');
    display(a);
    return 0;
}
int create(int num) {
    return num % MAX; }
void linearprob(int a[], int key, int num) {
    int flag = 0;
    int i;
   if (a[key] == -1) {
        a[key] = num;
    } else {
                for (i = (key + 1) % MAX; i != key; i = (i + 1) % MAX) {
            if (a[i] == -1) {
                a[i] = num;
                flag = 1;
                break;
            }
        }
        if (!flag) {
printf("\nHash Table is full");
            display(a);
exit(1);
        }
    }
}
void display(int a[]) {
printf("\nHash Table is:\n");
    for (int i = 0; i < MAX; i++) {
printf("Index %d: ", i);
        if (a[i] != -1)
printf("%d", a[i]);
        else
            printf("EMPTY");
        printf("\n");
    }
}
```

OUTPUT:

```
COLLISION HANDLING BY LINEAR PROBING
Enter the number:131
Do U wish to continue? (y/n)y
Enter the number:21
```

```
Do U wish to continue? (y/n)y
Enter the number:3
Do U wish to continue? (y/n)y
Enter the number:4
Do U wish to continue? (y/n)y
Enter the number:8
Do U wish to continue? (y/n)y
Enter the number:9
Do U wish to continue? (y/n)y
Enter the number:18
Do U wish to continue? (y/n)n
Hash Table is..
0 18
1 131
2 21
3 3
4 4
5 5
6 -1
7 -1
8 8
9 9
```

Ex.No:14 (c) **QUADRATIC PROBING**
DATE:

PROGRAM :

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 10
#define EMPTY -1
#define DELETED -2
int hashTable[TABLE_SIZE];
void initializeTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashTable[i] = EMPTY;
    }
}
int hashFunction(int key) {
    return key % TABLE_SIZE;
}
void insert(int key) {
    int hashIndex = hashFunction(key);
    int i = 0;
    while (hashTable[(hashIndex + i * i) % TABLE_SIZE] != EMPTY &&
           hashTable[(hashIndex + i * i) % TABLE_SIZE] != DELETED &&
           i < TABLE_SIZE) {
        i++;
    }
 int newIndex = (hashIndex + i * i) % TABLE_SIZE;
    if (i < TABLE_SIZE) {
        hashTable[newIndex] = key;
        printf("Inserted %d at index %d\n", key, newIndex);
    } else {
        printf("Table is full, could not insert %d\n", key);
    }
}
int search(int key) {
    int hashIndex = hashFunction(key);
    int i = 0;
    while (hashTable[(hashIndex + i * i) % TABLE_SIZE] != EMPTY &&
           i < TABLE_SIZE) {
        int newIndex = (hashIndex + i * i) % TABLE_SIZE;
        if (hashTable[newIndex] == key) {
            printf("Found %d at index %d\n", key, newIndex);
            return newIndex;
        }
        i++;
    }
    printf("%d not found in the hash table\n", key);
    return -
void delete(int key) {
    int index = search(key);
    if (index != -1) {
        hashTable[index] = DELETED;
        printf("Deleted %d from index %d\n", key, index);
    } else {
        printf("Cannot delete %d as it is not in the hash table\n", key);
    }
}
void display() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashTable[i] == EMPTY) {
            printf("hashTable[%d]: EMPTY\n", i);
        } else if (hashTable[i] == DELETED) {
            printf("hashTable[%d]: DELETED\n", i);
        } else {
            printf("hashTable[%d]: %d\n", i, hashTable[i]);
        }
    }
}

int main() {
    initializeTable();
    insert(23);
    insert(43);
    insert(13);
    insert(27);
    printf("Hash Table after insertions:\n");
    display();
    search(43);
    search(35);
    delete(43);
    delete(35);
    printf("\nHash Table after deletions:\n");
    display();

    return 0;
}
```

OUTPUT :

```
Inserted 23 at index 3
Inserted 43 at index 4
Inserted 13 at index 7
Inserted 27 at index 8
Hash Table after insertions:
hashTable[0]: EMPTY
hashTable[1]: EMPTY
```

```
hashTable[2]: EMPTY
hashTable[3]: 23
hashTable[4]: 43
hashTable[5]: EMPTY
hashTable[6]: EMPTY
hashTable[7]: 13
hashTable[8]: 27
hashTable[9]: EMPTY
Found 43 at index 4
35 not found in the hash table
Found 43 at index 4
Deleted 43 from index 4
35 not found in the hash table
Cannot delete 35 as it is not in the hash table

Hash Table after deletions:
hashTable[0]: EMPTY
hashTable[1]: EMPTY
hashTable[2]: EMPTY
hashTable[3]: 23
hashTable[4]: DELETED
hashTable[5]: EMPTY
hashTable[6]: EMPTY
hashTable[7]: 13
hashTable[8]: 27
hashTable[9]: EMPTY
```

RESULT:

Thus a C program to create hash table and collision handling by linear probing is written and executed successfully.