INDEX

1) Single linked list operations

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

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

struct Node *head,*tail;

//To display the linked list data
void linkedListTraversal()
{
    struct Node *ptr=head;
    while(ptr!=NULL)
    {
        printf("Element: %d\n", ptr->data);
        ptr = ptr->next;
    }
}

struct Node * InsertAtfirst(struct Node *head,int data){
    struct Node *ptr=(struct Node*)malloc(sizeof(struct Node));
    ptr->data=data;
    if(head == NULL) { // check if the list is empty
        head = ptr;
        ptr->next = NULL;
    }
```

```
    ptr->next=head;
    head=ptr;
    return head;
}

struct Node * InsertAtIndex(struct Node *head,int data,int index){
    struct Node *ptr=(struct Node *)malloc(sizeof(struct Node));
    struct Node *p=head;
    ptr->data=data;
    if(head == NULL) { // check if the list is empty
        head = ptr;
        ptr->next = NULL;
    }
    else if(index==0){
        head=InsertAtfirst(head,data);
    }
    int i=0;
    while(i!=(index-1)){
        p=p->next;
        i++;
    }
    ptr->next= p->next;
    p->next=ptr;
    return head;
}

struct Node * InsertAtEnd(struct Node *head,int data){
    struct Node *ptr= (struct Node *)malloc(sizeof(struct Node));
    struct Node *p=head;
    ptr->data=data;
```

```
    if(head == NULL) { // check if the list is empty
        head = ptr;
        ptr->next = NULL;
    }

    while(p->next!=NULL){
        p=p->next;
    }
    p->next=ptr;
    ptr->next=NULL;
    return head;
}

struct Node * deleteFirst(struct Node * head){
    struct Node * ptr = head;
    if(head == NULL) { // check if the list is empty
    printf("The list is empty\n");
    }
    head = head->next;
    free(ptr);
    return head;
}

struct Node * deleteAtIndex(struct Node * head, int index){
    struct Node *p = head;
    struct Node *q = head->next;
    if(head == NULL) { // check if the list is empty
    printf("The list is empty\n");
    }
    if(index==0){
```

```
        head=deleteFirst(head);
    }
    for (int i = 1; i < index-1; i++)
    {
        p = p->next;
        q = q->next;
    }

    p->next = q->next;
    free(q);
    return head;
}

struct Node *deleteAtLast(struct Node * head){
    struct Node *p = head;
    struct Node *q = head->next;
    if(head == NULL) { // check if the list is empty
    printf("The list is empty\n");
    }
    while(q->next !=NULL)
    {
        p = p->next;
        q = q->next;
    }
    p->next = NULL;
    free(q);
    return head;
}

void Create(int i){
```

```
    struct Node *newNode=(struct Node*)malloc(sizeof(struct Node));
    printf("Enter the data in the node %d: ",i);
    scanf("%d",&(newNode->data));
    newNode->next=NULL;
    if(head==NULL){
        head=newNode;
        tail=newNode;
    }
    else{
        tail->next=newNode;
        tail=newNode;
    }
}

int main()
{
    int choice,num;
     int index;//index only for inserting at an index operation
     int n;
    printf("Enter the no.of nodes need to be added: ");
    scanf("%d",&n);
    for(int i=0;i<n;i++){
        Create(i);
    }
    linkedListTraversal();
while(1){
    printf("Operations on linked list are: \n1.Insert at begining.\n");
    printf("2.Insert at index.\n");
    printf("3.Insert at end.\n");
    printf("4.Delete at first\n");
```

```
printf("5.Delete at index\n");
printf("6.Delete at last\n");
printf("7.End\n");
printf("Enter the choice of insertion you need in linked list: ");
scanf("%d",&choice);

switch(choice){

case 1 :

    printf("Linked list before insertion:\n");
    linkedListTraversal();
    printf("Enter the data to be inserted: ");
    scanf("%d",&num);
    head = InsertAtfirst(head,num);
    printf("After insertion at begining:\n");
    linkedListTraversal(head);
    break;

case 2 :

    printf("Linked list before insertion:\n");
    linkedListTraversal();
    printf("Enter the data to be inserted: ");
    scanf("%d",&num);
    fflush(stdin);
    printf("Enter the index at which the node to be inserted: ");
    scanf("%d",&index);
    head = InsertAtIndex(head,num,index);
    printf(" After insertion at index %d :\n",index);
```

```
        linkedListTraversal();
        break;
    case 3:
        printf("Linked list before insertion:\n");
        linkedListTraversal();
        printf("Enter the data to be inserted: ");
        scanf("%d",&num);
        head = InsertAtEnd(head,num);
        printf(" After insertion at end: \n");
        linkedListTraversal();
        break;
    case 4:
        printf("Deleting the first Node\n");
        head=deleteFirst(head);
        printf("Linked list after Deletion\n");
        linkedListTraversal();
        break;
    case 5:
        printf("Enter the index at which the node is to be deleted:");
        scanf("%d",&index);
        head=deleteAtIndex(head,index);
        printf("Linked list after Deletion\n");
        linkedListTraversal();
        break;
    case 6:
        printf("Deleting the last Node\n");
        head=deleteAtLast(head);
        printf("Linked list after Deletion\n");
        linkedListTraversal();
        break;
```

```
    case 7:
        printf("The program has ended\n");
        exit(0);
    default:
        printf("****** Invalid choice *******\n");
}
}
return 0;
}
```

Output:

```
Enter the no.of nodes need to be added: 4
Enter the data in the node 0: 1
Enter the data in the node 1: 2
Enter the data in the node 2: 3
Enter the data in the node 3: 4
Element: 1
Element: 2
Element: 3
Element: 4
Operations on linked list are:
1.Insert at begining.
2.Insert at index.
3.Insert at end.
4.Delete at first
5.Delete at index
6.Delete at last
7.End
Enter the choice of insertion you need in linked list: 1
```

Linked list before insertion:

Element: 1

Element: 2

Element: 3

Element: 4

Enter the data to be inserted: 20

After insertion at begining:

Element: 20

Element: 1

Element: 2

Element: 3

Element: 4

Operations on linked list are:

1.Insert at begining.

2.Insert at index.

3.Insert at end.

4.Delete at first

5.Delete at index

6.Delete at last

7.End

Enter the choice of insertion you need in linked list: 5

Deleting the first Node

Linked list after Deletion

Element: 1

Element: 2

Element: 3

Element: 4

Operations on linked list are:

1.Insert at begining.

2.Insert at index.

3.Insert at end.

4.Delete at first

5.Delete at index

6.Delete at last

7.End

Enter the choice of insertion you need in linked list: 2

Linked list before insertion:

Element: 1

Element: 2

Element: 3

Element: 4

Enter the data to be inserted: 23

Enter the index at which the node to be inserted: 1

After insertion at index 1 :

Element: 1

Element: 23

Element: 2

Element: 3

Element: 4

Operations on linked list are:

1.Insert at begining.

2.Insert at index.

3.Insert at end.

4.Delete at first

5.Delete at index

6.Delete at last

7.End

Enter the choice of insertion you need in linked list: 7

The program has ended

2) Double Linked List Operations.

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

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

    struct Node *head;
    struct Node *tail;

void CreateDLL(int i){
    struct Node *newNode=(struct Node *)malloc(sizeof(struct Node));
    printf("Enter the data to be inserted in node %d: ",i);
    scanf("%d",&(newNode->data));
    newNode->next=NULL;
    newNode->prev=NULL;
    if(head==NULL){
        head=newNode;
        tail=newNode;
    }
    else{
        tail->next=newNode;
        newNode->prev=tail;
        tail=newNode;
    }
}
```

```c
struct Node * DllTraversal(struct Node *head){
    struct Node *ptr=head;
    while(ptr!=NULL){
        printf("Element:%d \n",ptr->data);
        ptr=ptr->next;
    }

}

struct Node * InsertAtBeg(struct Node *head){
    struct Node *newNode=(struct Node*)malloc(sizeof(struct Node));
     printf("Enter the data in the Node: ");
     scanf("%d",&newNode->data);
     newNode->prev=NULL;
     head->prev=newNode;
     newNode->next=head;
     head=newNode;
     return head;
}

struct Node * InsertAtEnd(struct Node *tail){
    struct Node *newNode=(struct Node*)malloc(sizeof(struct Node));
    newNode->prev=NULL;
    newNode->next=NULL;
    printf("Enter the data in the Node: ");
    scanf("%d",&(newNode->data));



    tail->next=newNode;
```

```c
    newNode->prev=tail;
    tail=newNode;
    return tail;

}

struct Node * InsertAtIndex(struct Node *head){
    int index;

    printf("Enter the index at which the node is to be inserted: ");
    scanf("%d",&index);
    if(index==0){
        InsertAtBeg(head);
    }
    else{
    struct Node *newNode=(struct Node *)malloc(sizeof(struct Node));
    struct Node *p=head;
    newNode->prev=NULL;
    newNode->next=NULL;
    printf("Enter the data in the Node: ");
    scanf("%d",&(newNode->data));
    for(int i=0;i<(index-1);i++){
        p=p->next;
    }

    newNode->prev=p;
    newNode->next=p->next;
    p->next=newNode;
    newNode->next->prev=newNode;
    }
```

```
    return head;
}

struct Node * DeleteAtBeg(struct Node *head){
    struct Node *ptr;
    ptr=head;
    if(head==NULL){
        printf("Linked list is empty\n");
    }
    else{
        head=head->next;
        head->prev=NULL;
    }
    free(ptr);
    return head;
}

struct Node * DeleteAtEnd(struct Node *tail){
    struct Node *ptr;
    ptr=tail;
    tail=tail->prev;
    tail->next=NULL;
    free(ptr);
    return tail;
}

struct Node * DeleteAtIndex(struct Node *head){
    struct Node *ptr=head;
    int i=1,index;
    printf("Enter the index at which the node is to be inserted:");
```

```
    scanf("%d",&index);
    while(i<index){
        ptr=ptr->next;
        i++;
    }
    ptr->prev->next=ptr->next;
    ptr->next->prev=ptr->prev;
    free(ptr);
    return head;
}

struct Node * DLLReversal(){
    struct Node *Current,*nextNode=NULL;
    Current=head;
    while(Current!=NULL){
        nextNode=Current->next;
        Current->next=Current->prev;
        Current->prev=nextNode;
        Current=nextNode;
    }

    Current=head;
    head=tail;
    tail=Current;

}

int main(){
    int n,choice;
    printf("Enter the no.of nodes of linked list: ");
```

```
scanf("%d",&n);
for(int i=0;i<n;i++){
    CreateDLL(i);
}
printf("The linked list:\n");
DllTraversal(head);
while(1){

printf("Operations on Doubly linked list are: \n1.Insert at begining.\n");
printf("2.Insert at index.\n");
printf("3.Insert at end.\n");
printf("4.Delete at first\n");
printf("5.Delete at index\n");
printf("6.Delete at last\n");
printf("7.End\n");
printf("Enter the choice of operation you need in Doubly linked list: ");
scanf("%d",&choice);

switch(choice){

case 1 :
    printf("Linked list before insertion:\n");
    DllTraversal(head);
    fflush(stdin);
    head=InsertAtBeg(head);
    printf("After insertion at begining:\n");
    DllTraversal(head);
    break;

case 2 :
```

```
    printf("Linked list before insertion:\n");
    DllTraversal(head);
    fflush(stdin);
    head=InsertAtIndex(head);
    printf("After insertion\n");
    DllTraversal(head);
    break;
case 3:
    printf("Linked list before insertion:\n");
    DllTraversal(head);
    tail=InsertAtEnd(tail);
    printf(" After insertion at end: \n");
    DllTraversal(head);
    break;
case 4:
    printf("Deleting the Node at begining\n");
    head=DeleteAtBeg(head);
    printf("Linked list after Deletion\n");
    DllTraversal(head);
    break;
case 5:
    head=DeleteAtIndex(head);
    printf("Linked list after Deletion\n");
    DllTraversal(head);
    break;
case 6:
    printf("Deleting the last Node\n");
    tail=DeleteAtEnd(tail);
    printf("Linked list after Deletion\n");
```

```
        DllTraversal(head);
        break;
    case 7:
        printf("The program has ended\n");
        exit(0);
    default:
        printf("****** Invalid choice *******\n");
        break;
    }
}
}
```

Output:

```
Enter the no.of nodes of linked list: 4
Enter the data to be inserted in node 0: 1
Enter the data to be inserted in node 1: 2
Enter the data to be inserted in node 2: 3
Enter the data to be inserted in node 3: 4
The linked list:
Element:1
Element:2
Element:3
Element:4
Operations on Doubly linked list are:
1.Insert at begining.
2.Insert at index.
3.Insert at end.
4.Delete at first
5.Delete at index
6.Delete at last
7.End
Enter the choice of operation you need in Doubly linked list: 1
```

Linked list before insertion:

Element:1

Element:2

Element:3

Element:4

Enter the data in the Node: 11

After insertion at begining:

Element:11

Element:1

Element:2

Element:3

Element:4

Operations on Doubly linked list are:

1.Insert at begining.

2.Insert at index.

3.Insert at end.

4.Delete at first

5.Delete at index

6.Delete at last

7.End

Enter the choice of operation you need in Doubly linked list: 3

Linked list before insertion:

Element:11

Element:1

Element:2

Element:3

Element:4

Enter the data in the Node: 25

After insertion at end:

Element:11

Element:1

Element:2

Element:3

Element:4

Element:25

Operations on Doubly linked list are:

1.Insert at begining.

2.Insert at index.

3.Insert at end.

4.Delete at first

5.Delete at index

6.Delete at last

7.End

Enter the choice of operation you need in Doubly linked list: 7

The program has ended

3 A) Circular single linked list

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

struct Node{
    int data;
    struct Node *next;
};
struct Node *head,*tail;

void Create(int i);

//To display the linked list data
void linkedListTraversal(struct Node * head){
    struct Node *ptr=head;
    while(ptr->next!=head){
        printf("%d\n",ptr->data);
        ptr=ptr->next;
    }
    printf("%d\n",ptr->data);
}

struct Node * InsertAtBeg(struct Node *head){
    struct Node *ptr=(struct Node*)malloc(sizeof(struct Node));
    printf("enter the data in the new Node\n");
    scanf("%d",&ptr->data);
    tail->next=ptr;
    ptr->next=head;
    head=ptr;
    return head;
}

struct Node * InsertAtIndex(struct Node *head){
    struct Node *ptr=(struct Node *)malloc(sizeof(struct Node));
    struct Node *p=head->next;
    int index;
    printf("enter the data in the new Node\n");
    scanf("%d",&ptr->data);
    printf("Enter the index: ");
    scanf("%d",&index);
    if(index==0){
        InsertAtBeg(head);
    }
    int i=0;
    while(i!=(index-1)){
        p=p->next;
        i++;
    }
    ptr->next= p->next;
```

```
    p->next=ptr;
    return head;
}

struct Node * InsertAtEnd(struct Node *head){
    struct Node *ptr= (struct Node *)malloc(sizeof(struct Node));
    struct Node *p=head->next;
    printf("enter the data in the new Node\n");
    scanf("%d",&ptr->data);
    while(p->next!=head){
        p=p->next;
    }
    ptr->next=p->next;
    p->next=ptr;
    tail=ptr;
    return head;
}

struct Node *DeleteAtBeg(struct Node *head){
    struct Node *ptr=head,*p=head;
    while(ptr->next!=head){
        ptr=ptr->next;
    }
    ptr->next=head->next;
    head=ptr->next;
    free(p);
    return head;
}

struct Node *DeleteAtEnd(struct Node *head){
    if(head==NULL){
        printf("UnderflowI\n");
    }
    else if(head==tail){
        tail=NULL;
        free(head);
    }
    else{
    struct Node *p=head;
    struct Node *q=head->next;
    while(q->next!=head){
        p=p->next;
        q=q->next;
    }
    p->next=q->next;
    tail=p;
    free(q);
}
return head;
}
```

```
struct Node *DeleteAtindex(struct Node *head){
    int n,i=0;
    struct Node *p=head;
    struct Node *q=head->next;

    printf("Enter the index at which the node is to be deleted: ");
    scanf("%d",&n);

    if(n==0){
        DeleteAtBeg(head);
    }

    while(i!=n-1){
        p=p->next;
        q=q->next;
        i++;
    }
    p->next=q->next;
    free(q);
    return head;
}


int main()
{
    int choice,num;
     int index;//index only for inserting at an index operation
     int n;
    printf("Enter the no.of nodes need to be added: ");
    scanf("%d",&n);
    for(int i=0;i<n;i++){
        Create(i);
    }
while(1){
    printf("Operations on Circular linked list are: \n1.Insert at begining.\n");
    printf("2.Insert at index.\n");
    printf("3.Insert at end.\n");
    printf("4.Delete at first\n");
    printf("5.Delete at index\n");
    printf("6.Delete at last\n");
    printf("7.End\n");
    printf("Enter the choice of insertion you need in linked list: ");
    scanf("%d",&choice);
    switch (choice){
    case 1 :
        printf("Linked list before insertion:\n");
        linkedListTraversal(head);
        fflush(stdin);
        head=InsertAtBeg(head);
```

```
        printf("After insertion at begining:\n");
        linkedListTraversal(head);
        break;

    case 2 :

        printf("Linked list before insertion:\n");
        linkedListTraversal(head);
        fflush(stdin);
        head=InsertAtIndex(head);
        printf("After insertion\n");
        linkedListTraversal(head);
        break;
    case 3:
        printf("Linked list before insertion:\n");
        linkedListTraversal(head);
        head=InsertAtEnd(head);
        printf(" After insertion at end: \n");
        linkedListTraversal(head);
        break;

   case 4:
        printf("Deleting the Node at begining\n");
        head=DeleteAtBeg(head);
        printf("Linked list after Deletion\n");
        linkedListTraversal(head);
        break;
    case 5:
        head=DeleteAtindex(head);
        printf("Linked list after Deletion\n");
        linkedListTraversal(head);
        break;
    case 6:
        printf("Deleting the last Node\n");
        head=DeleteAtEnd(head);
        printf("Linked list after Deletion\n");
        linkedListTraversal(head);
        break;
    case 7:
        printf("The program has ended\n");
        exit(0);
    default:
        printf("****** Invalid choice *******\n");
        break;
    }
}
return 0;
}

void Create(int i){
```

```
    struct Node *newNode=(struct Node*)malloc(sizeof(struct Node));
    printf("Enter the data in the node %d: ",i);
    scanf("%d",&(newNode->data));
    newNode->next=NULL;
    if(head==NULL){
        head=newNode;
        tail=newNode;
    }
    else{
        tail->next=newNode;
        tail=newNode;
    }
    tail->next=head;
}
```

Output:
Enter the no.of nodes need to be added: 3
Enter the data in the node 0: 1
Enter the data in the node 1: 2
Enter the data in the node 2: 3
Operations on Circular linked list are:
1.Insert at begining.
2.Insert at index.
3.Insert at end.
4.Delete at first
5.Delete at index
6.Delete at last
7.End
Enter the choice of insertion you need in linked list: 1
Linked list before insertion:
1
2
3
enter the data in the new Node
45
After insertion at begining:
45
1
2
3
Operations on Circular linked list are:
1.Insert at begining.
2.Insert at index.
3.Insert at end.
4.Delete at first
5.Delete at index
6.Delete at last
7.End
Enter the choice of insertion you need in linked list: 4
Deleting the Node at begining
Linked list after Deletion

```
1
2
3
Operations on Circular linked list are:
1.Insert at begining.
2.Insert at index.
3.Insert at end.
4.Delete at first
5.Delete at index
6.Delete at last
7.End
Enter the choice of insertion you need in linked list: 6
Deleting the last Node
Linked list after Deletion
1
2
Operations on Circular linked list are:
1.Insert at begining.
2.Insert at index.
3.Insert at end.
4.Delete at first
5.Delete at index
6.Delete at last
7.End
Enter the choice of insertion you need in linked list: 7
The program has ended
```

3 B) Circular double linked list

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


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

struct Node *head=NULL;
struct Node *tail=NULL;

void Create()
{
    struct Node *newNode=(struct Node *)malloc(sizeof(struct Node));
    printf("Enter the data in the newnode: ");
    scanf("%d",&newNode->data);
    newNode->next=NULL;
    if(head==NULL){
        head=newNode;
        tail=newNode;
        head->next=head;
        head->prev=head;
    }
    else{
        tail->next=newNode;
        newNode->prev=tail;
        newNode->next=head;
        head->prev=newNode;
        tail=newNode;
    }

}

void linkedListTraversal(){
    struct Node *ptr=(struct Node *)malloc(sizeof(struct Node));
    ptr=head;
    while(ptr!=tail){
        printf("%d\n",ptr->data);
        ptr=ptr->next;
    }
    printf("%d\n",ptr->data);
}

struct Node *InsertAtfirst(){
    struct Node *newNode=(struct Node *)malloc(sizeof(struct Node));
    struct Node *p=head;
    printf("Enter the data in the new node: ");
```

```
    scanf("%d",&newNode->data);
    newNode->next=NULL;
    if(head==NULL){
        head=newNode;
        tail=newNode;
        newNode->prev=tail;
        newNode->next=head;
    }
    else{
        newNode->next=head;
        head->prev=newNode;
        newNode->prev=tail;
        tail->next=newNode;
    head=newNode;
    }
    return head;
}

struct Node *InsertAtEnd(){
    struct Node *newNode=(struct Node *)malloc(sizeof(struct Node));
    printf("Enter the data in the new node: ");
    scanf("%d",&newNode->data);
    newNode->next=NULL;
    if(head==NULL){
        head=newNode;
        tail=newNode;
        newNode->prev=tail;
        newNode->next=head;
    }
    else{
        newNode->prev=tail;
        tail->next=newNode;
        newNode->next=head;
        head->prev=newNode;
        tail=newNode;
    }
    return head;
}

struct Node *InsertAtIndex(){
    int i=1,n;
    struct Node *newNode=(struct Node *)malloc(sizeof(struct Node));
    struct Node *p=head;
    printf("Enter the data in the new node: ");
    scanf("%d",&newNode->data);
    newNode->next=NULL;
    printf("Enter the index: ");
    scanf("%d",&n);
    while(i!=(n-1)){
        p=p->next;
```

```
        i++;
    }
    newNode->prev=p;
    newNode->next=p->next;
    p->next->prev=newNode;
    p->next=newNode;
    return head;
}

struct Node *deleteFirst(){
    struct Node *ptr=head;
    head=head->next;
    head->prev=tail;
    tail->next=head;
    free(ptr);
    return head;
}
struct Node *deleteAtLast(){
    struct Node *ptr=tail;
    tail=tail->prev;
    tail->next=head;
    head->prev=tail;
    free(ptr);
    return head;
}

struct Node *deleteAtIndex(){
    struct Node *ptr=head;
    int i=1,index;
    printf("Enter the index at which the node is to be inserted:");
    scanf("%d",&index);
    while(i<index){
        ptr=ptr->next;
        i++;
    }
    ptr->prev->next=ptr->next;
    ptr->next->prev=ptr->prev;
    if(ptr->next==head){
        tail=ptr->prev;
        free(ptr);
    }
    else{
    free(ptr);
    }
    return head;
}
int main()
{
    int choice,num;
     int index;//index only for inserting at an index operation
```

```
    int n;
    printf("Enter the no.of nodes need to be added: ");
    scanf("%d",&n);
    for(int i=0;i<n;i++){
        Create(i);
    }
    linkedListTraversal();
while(1){
    printf("Operations on Doubly Circular linked list are: \n1.Insert at begining.\n");
    printf("2.Insert at index.\n");
    printf("3.Insert at end.\n");
    printf("4.Delete at first\n");
    printf("5.Delete at index\n");
    printf("6.Delete at last\n");
    printf("7.End\n");
    printf("Enter the choice of operation you need in linked list: ");
    scanf("%d",&choice);
    switch(choice){

    case 1 :

        printf("Linked list before insertion:\n");
        linkedListTraversal();
        head = InsertAtfirst();
        printf("After insertion at begining:\n");
        linkedListTraversal(head);
        break;

    case 2 :

        printf("Linked list before insertion:\n");
        linkedListTraversal();
        printf("Enter the index at which the node to be inserted: ");
        scanf("%d",&index);
        head = InsertAtIndex();
        printf(" After insertion at index %d :\n",index);
        linkedListTraversal();
        break;
    case 3:
        printf("Linked list before insertion:\n");
        linkedListTraversal();
        head = InsertAtEnd();
        printf(" After insertion at end: \n");
        linkedListTraversal();
        break;
    case 4:
        printf("Deleting the first Node\n");
        head=deleteFirst();
        printf("Linked list after Deletion\n");
        linkedListTraversal();
```

```
        break;
    case 5:
        head=deleteAtIndex();
        printf("Linked list after Deletion\n");
        linkedListTraversal();
        break;
    case 6:
        printf("Deleting the last Node\n");
        head=deleteAtLast();
        printf("Linked list after Deletion\n");
        linkedListTraversal();
        break;
    case 7:
        printf("The program has ended\n");
        exit(0);
    default:
        printf("****** Invalid choice *******\n");
 }
 }
 return 0;
 }
```

Output:

Enter the no.of nodes need to be added: 3

Enter the data in the newnode: 1

Enter the data in the newnode: 2

Enter the data in the newnode: 3

1

2

3

Operations on Doubly Circular linked list are:

1.Insert at begining.

2.Insert at index.

3.Insert at end.

4.Delete at first

5.Delete at index

6.Delete at last

7.End

Enter the choice of operation you need in linked list: 4

Deleting the first Node

Linked list after Deletion

2

3

Operations on Doubly Circular linked list are:

1.Insert at begining.

2.Insert at index.

3.Insert at end.

4.Delete at first

5.Delete at index

6.Delete at last

7.End

Enter the choice of operation you need in linked list: 1

Linked list before insertion:

2

3

Enter the data in the new node: 1

After insertion at begining:

1

2

3

Operations on Doubly Circular linked list are:

1.Insert at begining.

2.Insert at index.

3.Insert at end.

4.Delete at first

5.Delete at index

6.Delete at last

7.End

Enter the choice of operation you need in linked list: 7

The program has ended

4 A)Stack using linked list

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

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

struct Node *top=NULL;

void push(int x);
void pop();
void peek();
void display();


int main(){
    int choice;
    int x;//This variable is created for Enqueue operation
    while(1){
    printf("The operations can be performed on stack are:\n");
    printf("1.push\n2.pop\n3.peek\n4.display\n5.End\n");
    printf("Enter the choice: ");
    scanf("%d",&choice);
    switch(choice){
        case 1:
            printf("Enter the element to be pushed into the stack:");
            scanf("%d",&x);
```

```
            push(x);
            break;
        case 2:
            pop();
            break;
        case 3:
            peek();
            break;
        case 4:
            display();
            break;
        case 5:
        printf("Program has been ended\n");
            exit(0);
        default: printf("Enter a valid choice!\n");
    }
    }
}

void push(int x){
    struct Node *ptr=(struct Node*)malloc(sizeof(struct Node));
    ptr->data=x;
    ptr->next=top;
    top=ptr;
}

void display(){
    struct Node *ptr=top;
    while(ptr!=NULL){
        printf("%d\n",ptr->data);
```

```
        ptr=ptr->next;
    }
}

void peek(){
    if(top==NULL){
        printf("stack is empty\n");
    }
    else{
        printf("top element: %d",top->data);
    }
}

void pop(){
    struct Node *temp;
    temp=top;
    if(top==NULL){
        printf("stack is empty\n");
    }
    else{
        printf("popped element is: %d\n",top->data);
        top=top->next;
        free(temp);
    }
}
```

Output:

The operations can be performed on stack are:

1.push

2.pop

3.peek

4.display

5.End

Enter the choice: 1

Enter the element to be pushed into the stack:1

The operations can be performed on stack are:

1.push

2.pop

3.peek

4.display

5.End

Enter the choice: 1

Enter the element to be pushed into the stack:2

The operations can be performed on stack are:

1.push

2.pop

3.peek

4.display

5.End

Enter the choice: 4

2

1

The operations can be performed on stack are:

1.push

2.pop

3.peek

4.display

5.End

Enter the choice: 2

popped element is: 2

The operations can be performed on stack are:

1.push

2.pop

3.peek

4.display

5.End

Enter the choice: 4

1

The operations can be performed on stack are:

1.push

2.pop

3.peek

4.display

5.End

Enter the choice: 5

Program has been ended

4 B) Stack using array

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

#define N 5
int stack[N];// a stack of max size 5 is declared
int top=-1;

void push();
void pop();
void Display();

int main(){
   int choice;
   while(1){
   printf("The operations can be performed on stack are:\n");
   printf("1.push\n2.pop\n3.show\n4.End\n");
   printf("Enter the choice: ");
   scanf("%d",&choice);
   switch(choice){
      case 1:
         push();
         break;
      case 2:
         pop();
         break;
      case 3:
         Display();
         break;
      case 4:
```

```
printf("The program has ended\n"):
            exit(0);
        default: printf("Enter a valid choice!\n");
    }
    }
}
void push ()
{
    int val;
    if (top == N )
    printf("\n Overflow");
    else
    {
        printf("Enter the value?");
        scanf("%d",&val);
        top = top +1;
        stack[top] = val;
    }
}

void pop ()
{
    if(top == -1)
    printf("Underflow");
    else
    top = top -1;
}
void Display()
{
    for (int i=top;i>=0;i--)
```

```
    {
        printf("%d\n",stack[i]);
    }
    if(top == -1)
    {
        printf("Stack is empty");
    }
}
```

Output:

```
The operations can be performed on stack are:
1.push
2.pop
3.show
4.End
Enter the choice: 1
Enter the value?1
The operations can be performed on stack are:
1.push
2.pop
3.show
4.End
Enter the choice: 1
Enter the value?2
The operations can be performed on stack are:
1.push
2.pop
3.show
4.End
Enter the choice: 1
Enter the value?3
```

The operations can be performed on stack are:

1.push

2.pop

3.show

4.End

Enter the choice: 3

3

2

1

The operations can be performed on stack are:

1.push

2.pop

3.show

4.End

Enter the choice: 4

The program has ended

4 C) Queue using linked list

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

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

struct Node *front=NULL;
struct Node *rear=NULL;

void Enqueue(int x);
void Dequeue();
void peek();
void display();



int main(){
    int choice;
    int x;//This variable is created for Enqueue operation
    while(1){
    printf("The operations can be performed on Queue are:\n");
    printf("1.Enqueue\n2.Dequeue\n3.peek\n4.display\n5.End\n");
    printf("Enter the choice: ");
    scanf("%d",&choice);
    switch(choice){
        case 1:
            printf("Enter the element to be pushed into the Queue:");
            scanf("%d",&x);
```

```
            Enqueue(x);
            break;
        case 2:
            Dequeue();
            break;
        case 3:
            peek();
            break;
        case 4:
            display();
            break;
        case 5:
        printf("Program has been ended\n");
            exit(0);
        default: printf("Enter a valid choice!\n");
    }
    }
}

void Enqueue(int x){
    struct Node *ptr=(struct Node*)malloc(sizeof(struct Node));
    ptr->data=x;
    ptr->next=NULL;
    if(front==NULL&&rear==NULL){
        front=ptr;
        rear=ptr;
    }
        else{
            rear->next=ptr;
            rear=ptr;
```

```
        }
    }

void display(){
    if(front==NULL&&rear==NULL){
        printf("Queue is empty\n");
    }
    else{

    struct Node *ptr=front;
    while(ptr!=NULL){
        printf("%d\n",ptr->data);
        ptr=ptr->next;
    }
    }
}

void peek(){
    if(front==NULL&&rear==NULL){
        printf("Queue is empty\n");
    }
    else{
        printf(" %d",front->data);
    }
}

void Dequeue(){
    struct Node *ptr;
    ptr=front;
    if(front==NULL&&rear==NULL){
```

```
        printf("Queue is empty\n");
    }
    else{
        printf("popped element is: %d\n",front->data);
        front=front->next;
        free(ptr);
    }
}
```

Output:
The operations can be performed on Queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 1

Enter the element to be pushed into the Queue:1

The operations can be performed on Queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 1

Enter the element to be pushed into the Queue:2

The operations can be performed on Queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 1

Enter the element to be pushed into the Queue:3

The operations can be performed on Queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 4

1

2

3

The operations can be performed on Queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 2

popped element is: 1

The operations can be performed on Queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 4

2

3

The operations can be performed on Queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 5

Program has been ended

4 D)Queue using Array.

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

#define N 5
int queue[N];// a queue of max size 5 is declared
int front=-1,rear=-1;

void Enqueue(int x);
void Dequeue();
void Peek();
void Display();

int main(){
   int choice;
   int x;//This variable is created for Enqueue operation
   while(1){
   printf("The operations can be performed on queue are:\n");
   printf("1.Enqueue\n2.Dequeue\n3.peek\n4.display\n5.End\n");
   printf("Enter the choice: ");
   scanf("%d",&choice);
   switch(choice){
      case 1:
         printf("Enter the element to be inserted: ");
         scanf("%d",&x);
         Enqueue(x);
         break;
      case 2:
         Dequeue();
```

```
            break;
        case 3:
            Peek();
            break;
        case 4:
            Display();
            break;
        case 5:
            printf(“The program has been ended\n”);
           exit(0);
        default: printf("Enter a valid choice!\n");
    }
    }
}

void Enqueue(int x){
    if(rear==(N-1)){
        printf("Overflow!");
    }
    else if(front==-1 && rear==-1){
        front=rear=0;
        queue[rear]=x;
    }
    else{
        rear++;
        queue[rear]=x;
    }
}

void Dequeue(){
```

```
    if(front==-1 && rear==-1){
        printf("Underflow!");
    }
    else if(front==rear){
        front=rear=-1;
    }
    else{
        printf("The deleted element is %d\n",queue[front]);
        front++;
    }
}

void Peek(){
    if(front==-1 && rear==-1){
        printf("Queue is empty\n");
    }
    else{
        printf("The element peeked is %d\n",queue[front]);
    }
}

void Display()
{
    if(front==-1 && rear==-1){
       printf("Underflow!");
    }
    else{
        for(int i=front;i<=rear;i++){
            printf("%d\n",queue[i]);
        }
```

```
    }
}
```

Output:

The operations can be performed on queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 1

Enter the element to be inserted: 10

The operations can be performed on queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 1

Enter the element to be inserted: 11

The operations can be performed on queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 1

Enter the element to be inserted: 12

The operations can be performed on queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 4

10

11

12

The operations can be performed on queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 3

The element peeked is 10

The operations can be performed on queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 2

The deleted element is 10

The operations can be performed on queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 4

11

12

The operations can be performed on queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 5

The program has been ended

5) Circular queue using array

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

#define N 5
int queue[N];// a queue of max size 5 is declared
int front=-1,rear=-1;

void Enqueue(int x);
void Dequeue();
void Peek();
void Display();

int main(){
   int choice;
   long x;//This variable is created for Enqueue operation
   while(1){
   printf("The operations can be performed on circular queue are:\n");
   printf("1.Enqueue\n2.Dequeue\n3.peek\n4.display\n5.End\n");
   printf("Enter the choice: ");
   scanf("%d",&choice);
   switch(choice){
      case 1:
         printf("Enter the element to be inserted: ");
         scanf("%d",&x);
         Enqueue(x);
         break;
      case 2:
         Dequeue();
         break;
```

```
        case 3:
            Peek();
            break;
        case 4:
            Display();
            break;
        case 5:
printf(“The program has been ended\n”);
            exit(0);
        default: printf("Enter a valid choice!\n");
    }
    }
}

void Enqueue(int x){
    if(front==-1 && rear==-1){
        front=rear=0;
        queue[rear]=x;
    }
    else if((rear+1)%N==front){
        printf("Queue is full\n");
    }

    else{
        rear=(rear+1)%N;
        queue[rear]=x;
    }
}
```

```
void Dequeue(){
   if(front==-1 && rear==-1){
      printf("Underflow!");
   }
   else if(front==rear){
      printf("The dequeued element is %d",queue[front]);
      front=rear=-1;
   }
   else{
      printf("The deleted element is %d\n",queue[front]);
      front=(front+1)%N;
   }
}


void Peek()
{
   if(front==-1 && rear==-1)
   {
      printf("Queue is empty\n");
   }
   else
   {
      printf("The element peeked is %d\n",queue[front]);
   }
}

void Display()
{
   int i=front;
```

```
    if(front==-1 && rear==-1){
       printf("Underflow!");
    }
    else{
        while(i!=rear){
            printf("%d\n",queue[i]);
            i=(i+1)%N;
        }
        printf("%d\n",queue[rear]);
    }
}
```

Output:
The operations can be performed on circular queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 1

Enter the element to be inserted: 23

The operations can be performed on circular queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 1

Enter the element to be inserted: 24

The operations can be performed on circular queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 1

Enter the element to be inserted: 20

The operations can be performed on circular queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 4

23

24

20

The operations can be performed on circular queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 2

The deleted element is 23

The operations can be performed on circular queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 4

24

20

The operations can be performed on circular queue are:

1.Enqueue

2.Dequeue

3.peek

4.display

5.End

Enter the choice: 5

The program has been ended

6. Double ended queue using array

<u>CODE :</u>

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

#define MAX_SIZE 100

struct Deque {
    int arr[MAX_SIZE];
    int front, rear;
};

// Initialize the deque
void initializeDeque(struct Deque *deque) {
    deque->front = -1;
    deque->rear = -1;
}

// Check if the deque is empty
int isEmpty(struct Deque *deque) {
    return (deque->front == -1 && deque->rear == -1);
}

// Check if the deque is full
int isFull(struct Deque *deque) {
    return (deque->rear + 1) % MAX_SIZE == deque->front;
}

// Insert at front of the deque
void insertFront(struct Deque *deque, int data) {
```

```
    if (isFull(deque)) {
        printf("Deque is full. Cannot insert at front.\n");
        return;
    }

    if (isEmpty(deque)) {
        deque->front = 0;
        deque->rear = 0;
    } else {
        deque->front = (deque->front - 1 + MAX_SIZE) % MAX_SIZE;
    }

    deque->arr[deque->front] = data;
    printf("Inserted %d at the front of the deque.\n", data);
}

// Insert at rear of the deque
void insertRear(struct Deque *deque, int data) {
    if (isFull(deque)) {
        printf("Deque is full. Cannot insert at rear.\n");
        return;
    }

    if (isEmpty(deque)) {
        deque->front = 0;
        deque->rear = 0;
    } else {
        deque->rear = (deque->rear + 1) % MAX_SIZE;
    }
```

```c
    deque->arr[deque->rear] = data;
    printf("Inserted %d at the rear of the deque.\n", data);
}

// Delete from front of the deque
void deleteFront(struct Deque *deque) {
    if (isEmpty(deque)) {
        printf("Deque is empty. Cannot delete from front.\n");
        return;
    }

    if (deque->front == deque->rear) {
        // Last element in the deque
        printf("Deleted %d from the front of the deque.\n", deque->arr[deque->front]);
        deque->front = -1;
        deque->rear = -1;
    } else {
        printf("Deleted %d from the front of the deque.\n", deque->arr[deque->front]);
        deque->front = (deque->front + 1) % MAX_SIZE;
    }
}

// Delete from rear of the deque
void deleteRear(struct Deque *deque) {
    if (isEmpty(deque)) {
        printf("Deque is empty. Cannot delete from rear.\n");
        return;
    }

    if (deque->front == deque->rear) {
```

```
        // Last element in the deque
        printf("Deleted %d from the rear of the deque.\n", deque->arr[deque->rear]);
        deque->front = -1;
        deque->rear = -1;
    } else {
        printf("Deleted %d from the rear of the deque.\n", deque->arr[deque->rear]);
        deque->rear = (deque->rear - 1 + MAX_SIZE) % MAX_SIZE;
    }
}

// Display the deque
void displayDeque(struct Deque *deque) {
    if (isEmpty(deque)) {
        printf("Deque is empty.\n");
        return;
    }

    printf("Deque elements: ");
    int i = deque->front;
    while (1) {
        printf("%d ", deque->arr[i]);
        if (i == deque->rear)
            break;
        i = (i + 1) % MAX_SIZE;
    }
    printf("\n");
}

int main() {
    struct Deque deque;
```

```
    initializeDeque(&deque);

    insertFront(&deque, 10);
    insertRear(&deque, 20);
    insertFront(&deque, 5);

    displayDeque(&deque);
    deleteFront(&deque);
    displayDeque(&deque);
    deleteRear(&deque);
    displayDeque(&deque);

    return 0;
}
```

OUTPUT :

```
Inserted 10 at the front of the deque.
Inserted 20 at the rear of the deque.
Inserted 5 at the front of the deque.
Deque elements: 5 10 20
Deleted 5 from the front of the deque.
Deque elements: 10 20
Deleted 20 from the rear of the deque.
Deque elements: 10
```

7. Postfix expression evaluation

CODE :

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

#define MAX_SIZE 100

struct Stack {
    int arr[MAX_SIZE];
    int top;
};

// Initialize the stack
void initializeStack(struct Stack *stack) {
    stack->top = -1;
}

// Check if the stack is empty
int isEmpty(struct Stack *stack) {
    return stack->top == -1;
}

// Check if the stack is full
int isFull(struct Stack *stack) {
    return stack->top == MAX_SIZE - 1;
}

// Push an element onto the stack
void push(struct Stack *stack, int value) {
```

```
    if (isFull(stack)) {
        printf("Stack overflow. Cannot push element.\n");
        exit(EXIT_FAILURE);
    }

    stack->arr[++stack->top] = value;
}

// Pop an element from the stack
int pop(struct Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow. Cannot pop element.\n");
        exit(EXIT_FAILURE);
    }

    return stack->arr[stack->top--];
}

// Evaluate postfix expression
int evaluatePostfix(char *expression) {
    struct Stack stack;
    initializeStack(&stack);

    for (int i = 0; expression[i] != '\0'; ++i) {
        if (isdigit(expression[i])) {
            // If the current character is a digit, push it onto the stack
            push(&stack, expression[i] - '0');
        } else {
            // If the current character is an operator, pop two operands, perform the
operation, and push the result back
            int operand2 = pop(&stack);
```

```
            int operand1 = pop(&stack);

            switch (expression[i]) {
                case '+':
                    push(&stack, operand1 + operand2);
                    break;
                case '-':
                    push(&stack, operand1 - operand2);
                    break;
                case '*':
                    push(&stack, operand1 * operand2);
                    break;
                case '/':
                    push(&stack, operand1 / operand2);
                    break;
                default:
                    printf("Invalid operator: %c\n", expression[i]);
                    exit(EXIT_FAILURE);
            }
        }
    }

    // The final result is at the top of the stack
    return pop(&stack);
}

int main() {
    char expression[] = "235*+";
    int result = evaluatePostfix(expression);
    printf("Result of postfix expression %s is: %d\n", expression, result);
```

```
    return 0;
}
```

OUTPUT :

Result of postfix expression 235*+ is: 17

8. Infix to Postfix using stack

<u>CODE :</u>

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

#define MAX_SIZE 100

struct Stack {
    char arr[MAX_SIZE];
    int top;
};

// Initialize the stack
void initializeStack(struct Stack *stack) {
    stack->top = -1;
}

// Check if the stack is empty
int isEmpty(struct Stack *stack) {
    return stack->top == -1;
}

// Check if the stack is full
int isFull(struct Stack *stack) {
    return stack->top == MAX_SIZE - 1;
}

// Push an element onto the stack
```

```
void push(struct Stack *stack, char value) {
    if (isFull(stack)) {
        printf("Stack overflow. Cannot push element.\n");
        exit(EXIT_FAILURE);
    }

    stack->arr[++stack->top] = value;
}

// Pop an element from the stack
char pop(struct Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow. Cannot pop element.\n");
        exit(EXIT_FAILURE);
    }

    return stack->arr[stack->top--];
}

// Get the precedence of an operator
int getPrecedence(char operator) {
    switch (operator) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
```

```
        default:
            return -1;  // For parentheses or other characters
    }
}

// Convert infix expression to postfix expression
void infixToPostfix(char *infix, char *postfix) {
    struct Stack stack;
    initializeStack(&stack);

    int i, j;
    for (i = 0, j = 0; infix[i] != '\0'; ++i) {
        if (isalnum(infix[i])) {
            // If the current character is an operand, add it to the postfix expression
            postfix[j++] = infix[i];
        } else if (infix[i] == '(') {
            // If the current character is an opening parenthesis, push it onto the stack
            push(&stack, infix[i]);
        } else if (infix[i] == ')') {
            // If the current character is a closing parenthesis, pop and add operators
from the stack to the postfix expression until an opening parenthesis is encountered
            while (!isEmpty(&stack) && stack.arr[stack.top] != '(') {
                postfix[j++] = pop(&stack);
            }
            if (!isEmpty(&stack) && stack.arr[stack.top] == '(') {
                pop(&stack); // Pop the opening parenthesis
            } else {
                printf("Invalid expression. Mismatched parentheses.\n");
                exit(EXIT_FAILURE);
            }
        } else {
```

```
            // If the current character is an operator, pop and add operators from the stack to the postfix expression while they have equal or higher precedence
            while (!isEmpty(&stack) && getPrecedence(stack.arr[stack.top]) >= getPrecedence(infix[i])) {
                postfix[j++] = pop(&stack);
            }
            // Push the current operator onto the stack
            push(&stack, infix[i]);
        }
    }

    // Pop and add remaining operators from the stack to the postfix expression
    while (!isEmpty(&stack)) {
        if (stack.arr[stack.top] == '(') {
            printf("Invalid expression. Mismatched parentheses.\n");
            exit(EXIT_FAILURE);
        }
        postfix[j++] = pop(&stack);
    }

    // Add null terminator to the postfix expression
    postfix[j] = '\0';
}

int main() {
    char infix[] = "a+b*(c^d-e)^(f+g*h)-i";
    char postfix[MAX_SIZE];

    infixToPostfix(infix, postfix);

    printf("Infix expression: %s\n", infix);
```

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

    return 0;
}
```

<u>OUTPUT :</u>

Infix expression: a+b*(c^d-e)^(f+g*h)-i

Postfix expression: abcd^e-fgh*+^*+i-

9. Polynomial addition using linked list

CODE :

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

// Node structure for a term in a polynomial
struct Node {
    int coefficient;
    int exponent;
    struct Node *next;
};

// Function to create a new node with given coefficient and exponent
struct Node *createNode(int coefficient, int exponent) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }

    newNode->coefficient = coefficient;
    newNode->exponent = exponent;
    newNode->next = NULL;

    return newNode;
}

// Function to insert a term into the polynomial
void insertTerm(struct Node **poly, int coefficient, int exponent) {
```

```
    struct Node *newTerm = createNode(coefficient, exponent);

    // If the polynomial is empty, make the new term the head of the list
    if (*poly == NULL) {
        *poly = newTerm;
    } else {
        struct Node *current = *poly;
        struct Node *prev = NULL;

        // Traverse the list to find the correct position for the new term based on the
exponent
        while (current != NULL && current->exponent > exponent) {
            prev = current;
            current = current->next;
        }

        // Insert the new term
        if (prev == NULL) {
            // Insert at the beginning
            newTerm->next = *poly;
            *poly = newTerm;
        } else {
            // Insert in the middle or at the end
            prev->next = newTerm;
            newTerm->next = current;
        }
    }
}

// Function to add two polynomials
struct Node *addPolynomials(struct Node *poly1, struct Node *poly2) {
```

```
struct Node *result = NULL;

while (poly1 != NULL && poly2 != NULL) {
    if (poly1->exponent > poly2->exponent) {
        insertTerm(&result, poly1->coefficient, poly1->exponent);
        poly1 = poly1->next;
    } else if (poly1->exponent < poly2->exponent) {
        insertTerm(&result, poly2->coefficient, poly2->exponent);
        poly2 = poly2->next;
    } else {
        // Exponents are equal, add coefficients
        int sum = poly1->coefficient + poly2->coefficient;
        if (sum != 0) {
            insertTerm(&result, sum, poly1->exponent);
        }
        poly1 = poly1->next;
        poly2 = poly2->next;
    }
}

// Add any remaining terms from poly1
while (poly1 != NULL) {
    insertTerm(&result, poly1->coefficient, poly1->exponent);
    poly1 = poly1->next;
}

// Add any remaining terms from poly2
while (poly2 != NULL) {
    insertTerm(&result, poly2->coefficient, poly2->exponent);
    poly2 = poly2->next;
```

```
    }

    return result;
}

// Function to display a polynomial
void displayPolynomial(struct Node *poly) {
    while (poly != NULL) {
        printf("%dx^%d", poly->coefficient, poly->exponent);
        poly = poly->next;
        if (poly != NULL) {
            printf(" + ");
        }
    }
    printf("\n");
}

// Function to free the memory allocated for the polynomial
void freePolynomial(struct Node *poly) {
    struct Node *temp;
    while (poly != NULL) {
        temp = poly;
        poly = poly->next;
        free(temp);
    }
}

int main() {
    // Example polynomials
    struct Node *poly1 = NULL;
```

```c
    struct Node *poly2 = NULL;

    // Insert terms into the first polynomial: 3x^2 + 2x^1 + 5x^0
    insertTerm(&poly1, 3, 2);
    insertTerm(&poly1, 2, 1);
    insertTerm(&poly1, 5, 0);

    // Insert terms into the second polynomial: 1x^3 + 4x^2 + 2x^0
    insertTerm(&poly2, 1, 3);
    insertTerm(&poly2, 4, 2);
    insertTerm(&poly2, 2, 0);

    printf("Polynomial 1: ");
    displayPolynomial(poly1);

    printf("Polynomial 2: ");
    displayPolynomial(poly2);

    struct Node *sum = addPolynomials(poly1, poly2);

    printf("Sum of polynomials: ");
    displayPolynomial(sum);

    // Free the allocated memory
    freePolynomial(poly1);
    freePolynomial(poly2);
    freePolynomial(sum);

    return 0;
}
```

OUTPUT :

Polynomial 1: 3x^2 + 2x^1 + 5x^0

Polynomial 2: 1x^3 + 4x^2 + 2x^0

Sum of polynomials: 1x^3 + 7x^2 + 2x^1 + 7x^0

10 a. Binary Search

CODE :

```
#include <stdio.h>

// Binary search function
int binarySearch(int arr[], int low, int high, int target) {
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == target) {
            return mid;  // Element found, return its index
        } else if (arr[mid] < target) {
            low = mid + 1;  // Search in the right half
        } else {
            high = mid - 1;  // Search in the left half
        }
    }

    return -1;  // Element not found
}

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 6;

    int result = binarySearch(arr, 0, n - 1, target);

    if (result != -1) {
        printf("Element %d found at index %d.\n", target, result);
```

```
    } else {
        printf("Element %d not found in the array.\n", target);
    }

    return 0;
}
```

<u>OUTPUT :</u>

Element 6 found at index 5.

10b) Linear search

Code:

```
#include <stdio.h>
int linearSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}

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

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

    int target;
    printf("Enter the target value: ");
    scanf("%d", &target);

    int result = linearSearch(arr, n, target);

    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the array\n", target);
    }
    return 0;
}
```

Output

```
Enter the size of the array: 4
Enter 4 elements:
23
34
12
56
Enter the target value: 12
Element 12 found at index 2
```

11)Implementation of hashing
a) Separate chaining
code :

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

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

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

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode != NULL) {
        newNode->data = data;
        newNode->next = NULL;
    }
    return newNode;
}

struct HashTable* createHashTable(int size) {
    struct HashTable* hashTable = (struct HashTable*)malloc(sizeof(struct
HashTable));
    if (hashTable != NULL) {
        hashTable->size = size;
        hashTable->table = (struct Node**)malloc(sizeof(struct Node*) * size);

        // Initialize each bucket with NULL
        for (int i = 0; i < size; i++) {
            hashTable->table[i] = NULL;
        }
    }
    return hashTable;
}

// Function to calculate hash value
int hashFunction(struct HashTable* hashTable, int key) {
    return key % hashTable->size;
}

// Function to insert a key into the hash table, ensuring each bucket has at least one
key
void insert(struct HashTable* hashTable, int key) {
    // Calculate hash value
    int hashValue = hashFunction(hashTable, key);
```

```
    // Create a new node with the key
    struct Node* newNode = createNode(key);

    // If the bucket is empty, insert at the beginning
    if (hashTable->table[hashValue] == NULL) {
        hashTable->table[hashValue] = newNode;
    } else {
        // Find the first empty bucket using a separate loop
        int i = 1;
        while (hashTable->table[(hashValue + i) % hashTable->size] != NULL && i <
hashTable->size) {
            i++;
        }

        // If an empty bucket is found, insert the node
        if (i < hashTable->size) {
            hashTable->table[(hashValue + i) % hashTable->size] = newNode;
        }
    }
}

void displayHashTable(struct HashTable* hashTable) {
    for (int i = 0; i < hashTable->size; i++) {
        printf("Bucket %d:", i);
        struct Node* current = hashTable->table[i];
        while (current != NULL) {
            printf(" %d", current->data);
            current = current->next;
        }
        printf("\n");
    }
}

int main() {
    struct HashTable* hashTable = createHashTable(5);

    // Insert some keys into the hash table
    insert(hashTable, 12);
    insert(hashTable, 22);
    insert(hashTable, 7);
    insert(hashTable, 42);
    insert(hashTable, 32);

    displayHashTable(hashTable);
    for (int i = 0; i < hashTable->size; i++) {
        struct Node* current = hashTable->table[i];
        while (current != NULL) {
            struct Node* next = current->next;
            free(current);
```

```
            current = next;
        }
    }
    free(hashTable->table);
    free(hashTable);

    return 0;
}
```

Output

Bucket 0: 42
Bucket 1: 32
Bucket 2: 12
Bucket 3: 22
Bucket 4: 7

b) Linear probing
<u>Code :</u>

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

struct HashTable {
    int size;
    int *keys;
    int *values;
};

struct HashTable* createHashTable(int size) {
    struct HashTable* hashTable = (struct HashTable*)malloc(sizeof(struct
HashTable));
    if (hashTable != NULL) {
        hashTable->size = size;
        hashTable->keys = (int*)malloc(sizeof(int) * size);
        hashTable->values = (int*)malloc(sizeof(int) * size);

        for (int i = 0; i < size; i++) {
            hashTable->keys[i] = -1;
        }
    }
    return hashTable;
}

// Function to calculate hash value
int hashFunction(struct HashTable* hashTable, int key) {
    return key % hashTable->size;
}

// Function to insert a key-value pair into the hash table using linear probing
void insert(struct HashTable* hashTable, int key, int value) {
    int index = hashFunction(hashTable, key);

    // Linear probing to find the next available slot
    while (hashTable->keys[index] != -1) {
        index = (index + 1) % hashTable->size;
    }

    // Insert key-value pair into the found slot
    hashTable->keys[index] = key;
    hashTable->values[index] = value;
}

// Function to search for a key in the hash table
int search(struct HashTable* hashTable, int key) {
    int index = hashFunction(hashTable, key);

    // Linear probing to find the key or an empty slot
```

```
    while (hashTable->keys[index] != -1) {
        if (hashTable->keys[index] == key) {
            return hashTable->values[index];
        }
        index = (index + 1) % hashTable->size;
    }

    return -1;
}

// Function to display the hash table
void displayHashTable(struct HashTable* hashTable) {
    printf("Hash Table:\n");
    for (int i = 0; i < hashTable->size; i++) {
        printf("Bucket [%d] -> ", i);
        if (hashTable->keys[i] != -1) {
            printf("(%d, %d)", hashTable->keys[i], hashTable->values[i]);
        }
        printf("\n");
    }
}

int main() {
    struct HashTable* hashTable = createHashTable(5);

    insert(hashTable, 12, 120);
    insert(hashTable, 22, 220);
    insert(hashTable, 7, 70);
    insert(hashTable, 42, 420);
    insert(hashTable, 32, 320);

    displayHashTable(hashTable);

    int searchResult = search(hashTable, 7);
    if (searchResult != -1) {
        printf("Value for key 7: %d\n", searchResult);
    } else {
        printf("Key 7 not found in the hash table\n");
    }

    free(hashTable->keys);
    free(hashTable->values);
    free(hashTable);

    return 0;
}
```

Output:

Hash Table:
Bucket [0] -> (42, 420)
Bucket [1] -> (32, 320)
Bucket [2] -> (12, 120)
Bucket [3] -> (22, 220)
Bucket [4] -> (7, 70)
Value for key 7: 70

12. Recursive and iterative traversals on binary tree

Code:

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

// Structure for a binary tree node
struct TreeNode {
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
};

// Function to create a new node with a given value
struct TreeNode* createNode(int value) {
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    if (newNode != NULL) {
        newNode->data = value;
        newNode->left = NULL;
        newNode->right = NULL;
    }
    return newNode;
}

// Function to insert a new node with a given value into the binary search tree
struct TreeNode* insert(struct TreeNode* root, int value) {
    if (root == NULL) {
        return createNode(value);
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }

    return root;
}

// Recursive in-order traversal of a binary tree
void inOrderRecursive(struct TreeNode* root) {
    if (root != NULL) {
        inOrderRecursive(root->left);
        printf("%d ", root->data);
        inOrderRecursive(root->right);
    }
}

// Recursive pre-order traversal of a binary tree
void preOrderRecursive(struct TreeNode* root) {
    if (root != NULL) {
```

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

// Recursive post-order traversal of a binary tree
void postOrderRecursive(struct TreeNode* root) {
    if (root != NULL) {
        postOrderRecursive(root->left);
        postOrderRecursive(root->right);
        printf("%d ", root->data);
    }
}

// Iterative in-order traversal of a binary tree using a stack
void inOrderIterative(struct TreeNode* root) {
    struct TreeNode* stack[100];
    int top = -1;
    struct TreeNode* current = root;

    while (current != NULL || top != -1) {
        while (current != NULL) {
            stack[++top] = current;
            current = current->left;
        }

        current = stack[top--];
        printf("%d ", current->data);
        current = current->right;
    }
}

// Iterative pre-order traversal of a binary tree using a stack
void preOrderIterative(struct TreeNode* root) {
    if (root == NULL) {
        return;
    }

    struct TreeNode* stack[100];
    int top = -1;
    stack[++top] = root;

    while (top != -1) {
        struct TreeNode* current = stack[top--];
        printf("%d ", current->data);

        if (current->right != NULL) {
            stack[++top] = current->right;
        }
```

```
        if (current->left != NULL) {
            stack[++top] = current->left;
        }
    }
}

// Iterative post-order traversal of a binary tree using two stacks
void postOrderIterative(struct TreeNode* root) {
    if (root == NULL) {
        return;
    }

    struct TreeNode* stack1[100];
    struct TreeNode* stack2[100];
    int top1 = -1;
    int top2 = -1;

    stack1[++top1] = root;

    while (top1 != -1) {
        struct TreeNode* current = stack1[top1--];
        stack2[++top2] = current;

        if (current->left != NULL) {
            stack1[++top1] = current->left;
        }
        if (current->right != NULL) {
            stack1[++top1] = current->right;
        }
    }

    while (top2 != -1) {
        printf("%d ", stack2[top2--]->data);
    }
}

// Function to free memory of a binary tree
void freeTree(struct TreeNode* root) {
    if (root != NULL) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}

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

    // Insert nodes into the binary search tree
    root = insert(root, 10);
```

```
    insert(root, 5);
    insert(root, 15);
    insert(root, 3);
    insert(root, 7);
    insert(root, 12);
    insert(root, 18);

    // Recursive traversals
    printf("In-order (recursive): ");
    inOrderRecursive(root);
    printf("\n");

    printf("Pre-order (recursive): ");
    preOrderRecursive(root);
    printf("\n");

    printf("Post-order (recursive): ");
    postOrderRecursive(root);
    printf("\n");

    // Iterative traversals
    printf("In-order (iterative): ");
    inOrderIterative(root);
    printf("\n");

    printf("Pre-order (iterative): ");
    preOrderIterative(root);
    printf("\n");

    printf("Post-order (iterative): ");
    postOrderIterative(root);
    printf("\n");

    // Free memory
    freeTree(root);

    return 0;
}
```

Output:

```
In-order (recursive): 3 5 7 10 12 15 18
Pre-order (recursive): 10 5 3 7 15 12 18
Post-order (recursive): 3 7 5 12 18 15 10
In-order (iterative): 3 5 7 10 12 15 18
Pre-order (iterative): 10 5 3 7 15 12 18
Post-order (iterative): 3 7 5 12 18 15 10
```

13. Binary tree operations

Code:

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

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

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

struct TreeNode* insert(struct TreeNode* root, int value) {
    if (root == NULL) {
        return createNode(value);
    }
    if (root->left == NULL) {
        root->left = insert(root->left, value);
    } else if (root->right == NULL) {
        root->right = insert(root->right, value);
    } else {
        root->left = insert(root->left, value);
    }

    return root;
}

struct TreeNode* deleteNode(struct TreeNode* root, int value) {
    if (root == NULL) {
        return root;
    }

    if (root->data == value && root->left == NULL && root->right == NULL) {
        free(root);
        return NULL;
    }
    root->left = deleteNode(root->left, value);
    root->right = deleteNode(root->right, value);

    return root;
}
```

```
struct TreeNode* search(struct TreeNode* root, int value) {
    if (root == NULL || root->data == value) {
        return root;
    }
    struct TreeNode* leftResult = search(root->left, value);
    struct TreeNode* rightResult = search(root->right, value);

    return (leftResult != NULL) ? leftResult : rightResult;
}

struct TreeNode* createMirrorImage(struct TreeNode* root) {
    if (root == NULL) {
        return NULL;
    }

    struct TreeNode* newNode = createNode(root->data);
    newNode->left = createMirrorImage(root->right);
    newNode->right = createMirrorImage(root->left);

    return newNode;
}

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

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

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

void freeTree(struct TreeNode* root) {
    if (root != NULL) {
        freeTree(root->left);
        freeTree(root->right);
```

```
        free(root);
    }
}

int main() {
    struct TreeNode* root = NULL;
    root = insert(root, 1);
    insert(root, 2);
    insert(root, 3);
    insert(root, 4);
    insert(root, 5);

    printf("Original Binary Tree (In-order): ");
    inOrderTraversal(root);
    printf("\n");

    printf("Original Binary Tree (Pre-order): ");
    preOrderTraversal(root);
    printf("\n");

    printf("Original Binary Tree (Post-order): ");
    postOrderTraversal(root);
    printf("\n");

    int searchValue = 3;
    struct TreeNode* searchResult = search(root, searchValue);
    if (searchResult != NULL) {
        printf("Found %d in the binary tree.\n", searchValue);
    } else {
        printf("%d not found in the binary tree.\n", searchValue);
    }

    int deleteValue = 4;
    root = deleteNode(root, deleteValue);
    printf("Binary Tree after deleting %d (In-order): ", deleteValue);
    inOrderTraversal(root);
    printf("\n");

    struct TreeNode* mirrorRoot = createMirrorImage(root);
    printf("Mirror Image of Binary Tree (In-order): ");
    inOrderTraversal(mirrorRoot);
    printf("\n");

    freeTree(root);
    freeTree(mirrorRoot);

    return 0;
}
```

Output:
Original Binary Tree (In-order): 4 2 5 1 3
Original Binary Tree (Pre-order): 1 2 4 5 3
Original Binary Tree (Post-order): 4 5 2 3 1
Found 3 in the binary tree.
Binary Tree after deleting 4 (In-order): 2 5 1 3
Mirror Image of Binary Tree (In-order): 3 1 5 2

14.Binary Search Tree operations

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

// Definition of a node in BST
struct Node {
   int key;
   struct Node* left;
   struct Node* right;
};

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

// Function to insert a key into BST
struct Node* insert(struct Node* root, int key) {
   if (root == NULL)
      return createNode(key);

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

   return root;
```

```
}

// Function to find the minimum key in BST
struct Node* findMin(struct Node* root) {
    while (root->left != NULL)
        root = root->left;
    return root;
}

// Function to find the maximum key in BST
struct Node* findMax(struct Node* root) {
    while (root->right != NULL)
        root = root->right;
    return root;
}

// Function to search for a key in BST
struct Node* search(struct Node* root, int key) {
    if (root == NULL || root->key == key)
        return root;

    if (key < root->key)
        return search(root->left, key);
    else
        return search(root->right, key);
}

// Function to delete a node with a given key from BST
struct Node* deleteNode(struct Node* root, int key) {
    if (root == NULL)
```

```
        return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else {
        // Node with only one child or no child
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        // Node with two children, get the inorder successor (smallest in the right
subtree)
        struct Node* temp = findMin(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }

    return root;
}
```

```
// Function to print inorder traversal of BST
void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->key);
        inorderTraversal(root->right);
    }
}

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

    // Inserting elements into BST
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // Inorder traversal to display the BST
    printf("Inorder traversal of BST: ");
    inorderTraversal(root);
    printf("\n");

    // Finding minimum and maximum keys
    printf("Minimum key in BST: %d\n", findMin(root)->key);
    printf("Maximum key in BST: %d\n", findMax(root)->key);
```

```
    // Searching for a key
    int searchKey = 40;
    struct Node* searchResult = search(root, searchKey);
    if (searchResult != NULL)
        printf("Key %d found in BST.\n", searchKey);
    else
        printf("Key %d not found in BST.\n", searchKey);

    // Deleting a node with a given key
    int deleteKey = 30;
    root = deleteNode(root, deleteKey);
    printf("Inorder traversal after deleting node with key %d: ", deleteKey);
    inorderTraversal(root);
    printf("\n");

    return 0;
}
```

Output :

```
Inorder traversal of BST: 20 30 40 50 60 70 80
Minimum key in BST: 20
Maximum key in BST: 80
Key 40 found in BST.
Inorder traversal after deleting node with key 30: 20 40 50 60 70 80
```

15. Implement the following sorting algorithms:
a) Bubble sort b) Selection sort c) Insertion sort (d) Merge sort (e) Quick sort (f) Heap sort

a. Bubble Sort

```
#include<stdio.h>
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

void bubbleSort(int arr[], int n)
{
    for (int i = 0; i < n-1; i++)
    {
        for(int j = 0; j < n-i-1;j++)
        {
            if(arr[j] > arr[j+1])
            {
                swap(&arr[j], &arr[j+1]);
            }
        }
    }
}
void printArray(int arr[], int size) {
    for (int i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}


int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int size = sizeof(arr)/sizeof(arr[0]);

    bubbleSort(arr, size);
    printArray(arr, size);

}
```

OUTPUT:

```
Original Array:
64,34,25,12,22,11,90

Sorted Array:
11,12,22,25,34,64,90
```

b. Selection Sort

```
#include<stdio.h>

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

void selectionSort(int arr[], int n)
{
    int i, j, minIndex;
    for (int i = 0; i < n-1; i++)
    {
        minIndex = i;
        for(int j = 0; j < n; j++)
        {
            if (arr[j] < arr[minIndex])
            {
                minIndex = j;
            }
        }
        swap(&arr[minIndex], &arr[i]);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Original array: \n");
    printArray(arr, n);

    selectionSort(arr, n);

    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```

OUTPUT:

Original Array:
64,34,25,12,22,11,90

Sorted Array:
11,12,22,25,34,64,90

c. **Insertion Sort**

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        // Move elements of arr[0..i-1] that are greater than key to one position
ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

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

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: \n");
    printArray(arr, n);

    insertionSort(arr, n);

    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```

OUTPUT:

Original Array:
64,34,25,12,22,11,90

Sorted Array:
11,12,22,25,34,64,90

d. Merge Sort

```
#include <stdio.h>

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

    // Create temporary arrays
    int L[n1], R[n2];

    // Copy data to temporary arrays L[] and R[]
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    // Merge the temporary arrays back into arr[l..r]
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[], if there are any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[], if there are any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
```

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

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

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}

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

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: \n");
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);

    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```

OUTPUT:

```
Original Array:
64,34,25,12,22,11,90

Sorted Array:
11,12,22,25,34,64,90
```

e. Heap Sorts

```c
#include <stdio.h>

// Function to heapify a subtree rooted at node i, assuming that the subtrees are
already heapified
void heapify(int arr[], int n, int i) {
    int largest = i;  // Initialize largest as root
    int left = 2 * i + 1;  // Left child
    int right = 2 * i + 2;  // Right child

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // If largest is not the root
    if (largest != i) {
        // Swap the root with the largest element
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// Main function to perform Heap Sort
void heapSort(int arr[], int n) {
    // Build a max heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Extract elements from the heap one by one
    for (int i = n - 1; i > 0; i--) {
        // Move the current root to the end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Call heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

// Function to print an array
```

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

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: \n");
    printArray(arr, n);

    heapSort(arr, n);

    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```

OUTPUT:

```
Original Array:
64,34,25,12,22,11,90

Sorted Array:
11,12,22,25,34,64,90
```

f. Quick sort

```
#include <stdio.h>

// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
    int pivot = arr[low];
    int i = low + 1;
    int j = high;

    while (1) {
        while (i <= j && arr[i] <= pivot)
            i++;

        while (j >= i && arr[j] > pivot)
            j--;

        if (i <= j) {
            // Swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        } else {
            // Swap pivot with arr[j]
            int temp = arr[low];
            arr[low] = arr[j];
            arr[j] = temp;
            return j; // Return the pivot index
        }
    }
}
```

```c
// Function to perform QuickSort
void quicksort(int arr[], int low, int high) {
    if (low < high) {
        // Find pivot element such that elements smaller than pivot are on the left,
        // and elements greater than pivot are on the right
        int pivotIndex = partition(arr, low, high);

        // Recursively sort the sub-arrays
        quicksort(arr, low, pivotIndex - 1);
        quicksort(arr, pivotIndex + 1, high);
    }
}

int main() {
    int myArray[] = {3, 6, 8, 10, 1, 2, 1};
    int n = sizeof(myArray) / sizeof(myArray[0]);
    // Perform QuickSort
    quicksort(myArray, 0, n - 1);

    // Print the sorted array
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", myArray[i]);
    }

    return 0;
}
```

Output:

Sorted array: 1 1 2 3 6 8 10

16.AVL Tree Operations.

```
#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 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
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
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);
}

struct Node* insert(struct Node* node, int key) {
    /* 1.  Perform the normal BST rotation */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left  = insert(node->left, key);
    else
        node->right = insert(node->right, key);

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

```
    /* 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;
}
```

```c
/* Given a non-empty binary search tree, return the node with minimum
   key value found in that tree. Note that the entire tree does not
   need to be searched. */
struct Node * minValueNode(struct Node* node) {
    struct Node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

// Recursive function to delete a node with given key from subtree with
// given root. It returns root of the modified subtree.
struct Node* deleteNode(struct Node* root, int key) {
    // STEP 1: PERFORM STANDARD BST DELETE

    if (root == NULL)
        return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if ( key < root->key )
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if( key > root->key )
        root->right = deleteNode(root->right, key);
```

```
    }

    // if key is same as root's key, then This is the node
    // to be deleted
    else {
        // node with only one child or no child
        if( (root->left == NULL) || (root->right == NULL) ) {
            struct Node *temp = root->left ? root->left : root->right;

            // No child case
            if(temp == NULL) {
                temp = root;
                root = NULL;
            }
            else // One child case
                *root = *temp; // Copy the contents of the non-empty child

            free(temp);
        }
        else {
            // node with two children: Get the inorder successor (smallest
            // in the right subtree)
            struct Node* temp = minValueNode(root->right);

            // Copy the inorder successor's data to this node
            root->key = temp->key;

            // Delete the inorder successor
            root->right = deleteNode(root->right, temp->key);
        }
    }
```

```
// If the tree had only one node then return
if (root == NULL)
    return root;

// STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
root->height = max(height(root->left), height(root->right)) + 1;

// STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether
//  this node became unbalanced)
int balance = getBalance(root);

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

// Left Left Case
if (balance > 1 && getBalance(root->left) >= 0)
    return rightRotate(root);

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

// Right Right Case
if (balance < -1 && getBalance(root->right) <= 0)
    return leftRotate(root);

// Right Left Case
if (balance < -1 && getBalance(root->right) > 0) {
```

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

    return root;
}

// 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);
    }
}

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;
}
```

Output :

Preorder traversal of the constructed AVL tree is

30 20 10 25 40 50

17.Graph Traversal Methods

(A) Breadth first search

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

// Structure for representing a node in the adjacency list
struct Node {
    int data;
    struct Node* next;
};

// Structure for representing a graph
struct Graph {
    int vertices;
    struct Node** adjacencyList;
};

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

// Function to create a graph with a given number of vertices
struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->vertices = vertices;
```

```
    // Allocate memory for adjacency list
    graph->adjacencyList = (struct Node**)malloc(vertices * sizeof(struct Node*));

    // Initialize each adjacency list as empty
    for (int i = 0; i < vertices; ++i)
        graph->adjacencyList[i] = NULL;

    return graph;
}

// Function to add an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest) {
    // Add edge from source to destination
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjacencyList[src];
    graph->adjacencyList[src] = newNode;

    // Add edge from destination to source (since the graph is undirected)
    newNode = createNode(src);
    newNode->next = graph->adjacencyList[dest];
    graph->adjacencyList[dest] = newNode;
}

// Function to perform Breadth-First Search (BFS) traversal
void BFS(struct Graph* graph, int startVertex) {
    // Create an array to keep track of visited vertices
    int* visited = (int*)malloc(graph->vertices * sizeof(int));
    for (int i = 0; i < graph->vertices; ++i)
        visited[i] = 0; // Mark all vertices as not visited
```

```
// Create a queue for BFS
int* queue = (int*)malloc(graph->vertices * sizeof(int));
int front = 0, rear = 0;

// Enqueue the start vertex and mark it as visited
queue[rear++] = startVertex;
visited[startVertex] = 1;

// Perform BFS
while (front < rear) {
    // Dequeue a vertex from the queue
    int currentVertex = queue[front++];
    printf("%d ", currentVertex);

    // Traverse the adjacency list of the dequeued vertex
    struct Node* temp = graph->adjacencyList[currentVertex];
    while (temp != NULL) {
        int adjVertex = temp->data;
        // If the adjacent vertex is not visited, enqueue it and mark it as visited
        if (!visited[adjVertex]) {
            queue[rear++] = adjVertex;
            visited[adjVertex] = 1;
        }
        temp = temp->next;
    }
}

// Free allocated memory
free(visited);
```

```
    free(queue);
}

// Driver program
int main() {
    // Create a sample graph
    int vertices = 6;
    struct Graph* graph = createGraph(vertices);

    // Add edges
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 5);
    addEdge(graph, 4, 5);

    // Perform BFS starting from vertex 0
    printf("Breadth-First Search (BFS) starting from vertex 0:\n");
    BFS(graph, 0);

    // Free allocated memory for the graph
    for (int i = 0; i < vertices; ++i) {
        struct Node* temp = graph->adjacencyList[i];
        while (temp != NULL) {
            struct Node* next = temp->next;
            free(temp);
            temp = next;
        }
```

```
    }
    free(graph->adjacencyList);
    free(graph);

    return 0;
}
```

Output :

Breadth-First Search (BFS) starting from vertex 0:

0 2 1 4 3 5

17.Graph Traversal methods
(B) Depth first search

```
#include <stdio.h>

#include <stdlib.h>


// Structure for representing a node in the adjacency list

struct Node {

    int data;

    struct Node* next;

};


// Structure for representing a graph

struct Graph {

    int vertices;

    struct Node** adjacencyList;

};


// Function to create a new node

struct Node* createNode(int data) {

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    newNode->data = data;

    newNode->next = NULL;

    return newNode;

}


// Function to create a graph with a given number of vertices

struct Graph* createGraph(int vertices) {

    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));

    graph->vertices = vertices;
```

```
    // Allocate memory for adjacency list
    graph->adjacencyList = (struct Node**)malloc(vertices * sizeof(struct Node*));

    // Initialize each adjacency list as empty
    for (int i = 0; i < vertices; ++i)
        graph->adjacencyList[i] = NULL;

    return graph;
}

// Function to add an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest) {
    // Add edge from source to destination
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjacencyList[src];
    graph->adjacencyList[src] = newNode;

    // Add edge from destination to source (since the graph is undirected)
    newNode = createNode(src);
    newNode->next = graph->adjacencyList[dest];
    graph->adjacencyList[dest] = newNode;
}

// Recursive function for DFS traversal
void DFSUtil(struct Graph* graph, int vertex, int* visited) {
    // Mark the current vertex as visited
    visited[vertex] = 1;
    printf("%d ", vertex);

    // Traverse all the adjacent vertices
```

```
    struct Node* temp = graph->adjacencyList[vertex];
    while (temp != NULL) {
        int adjVertex = temp->data;
        if (!visited[adjVertex]) {
            DFSUtil(graph, adjVertex, visited);
        }
        temp = temp->next;
    }
}

// Function to perform Depth-First Search (DFS) traversal
void DFS(struct Graph* graph, int startVertex) {
    // Create an array to keep track of visited vertices
    int* visited = (int*)malloc(graph->vertices * sizeof(int));
    for (int i = 0; i < graph->vertices; ++i)
        visited[i] = 0; // Mark all vertices as not visited

    // Call the recursive utility function to perform DFS
    DFSUtil(graph, startVertex, visited);

    // Free allocated memory
    free(visited);
}

// Driver program
int main() {
    // Create a sample graph
    int vertices = 6;
    struct Graph* graph = createGraph(vertices);
```

```
    // Add edges
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 5);
    addEdge(graph, 4, 5);

    // Perform DFS starting from vertex 0
    printf("Depth-First Search (DFS) starting from vertex 0:\n");
    DFS(graph, 0);

    // Free allocated memory for the graph
    for (int i = 0; i < vertices; ++i) {
        struct Node* temp = graph->adjacencyList[i];
        while (temp != NULL) {
            struct Node* next = temp->next;
            free(temp);
            temp = next;
        }
    }
    free(graph->adjacencyList);
    free(graph);

    return 0;
}
```

Output :

Depth-First Search (DFS) starting from vertex 0:

0 2 4 5 3 1

18 a) Prim's algorithm to find out minimum spanning tree

<u>Program</u> :

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

#define V 5 // Number of vertices in the graph

int minKey(int key[], int mstSet[]) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++) {
        if (mstSet[v] == 0 && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }

    return min_index;
}

void printMST(int parent[], int graph[V][V]) {
    printf("Edge   Weight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d    %d \n", parent[i], i, graph[i][parent[i]]);
}

void primMST(int graph[V][V]) {
    int parent[V]; // Array to store constructed MST
    int key[V];    // Key values used to pick minimum weight edge in cut
```

```
    int mstSet[V]; // To represent set of vertices included in MST

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = 0;
    }

    // Always include the first vertex in MST
    key[0] = 0;     // Make key 0 so that this vertex is picked as the first vertex
    parent[0] = -1; // First node is always the root of MST

    // The MST will have V-1 edges
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum key vertex from the set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Include the picked vertex in MST
        mstSet[u] = 1;

        // Update key value and parent index of the adjacent vertices of the picked
vertex
        for (int v = 0; v < V; v++) {
            // Update key[v] only if the graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }
```

```
    // Print the constructed MST
    printMST(parent, graph);
}

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;
}
```

<u>Output</u> :

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

18.b) Krushkal's algorithm to find out minimum spanning tree

Program :

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

#define V 5 // Number of vertices in the graph

// Structure to represent an edge in the graph
struct Edge {
    int src, dest, weight;
};

// Structure to represent a subset for union-find
struct Subset {
    int parent;
    int rank;
};

// Function prototypes
int find(struct Subset subsets[], int i);
void Union(struct Subset subsets[], int x, int y);
int compare(const void* a, const void* b);
void kruskalMST(struct Edge edges[]);

int main() {
    struct Edge edges[] = {
        {0, 1, 2},
        {0, 3, 6},
        {1, 2, 3},
        {1, 3, 8},
```

```
        {1, 4, 5},
        {2, 4, 7},
        {3, 4, 9}
    };

    kruskalMST(edges);

    return 0;
}

// Find set of an element i
int find(struct Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// Perform union of two sets x and y
void Union(struct Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
```

```
    }
}

// Compare function for qsort to sort edges based on their weights
int compare(const void* a, const void* b) {
    return ((struct Edge*)a)->weight - ((struct Edge*)b)->weight;
}

// Kruskal's algorithm to find the minimum spanning tree
void kruskalMST(struct Edge edges[]) {
    // Allocate memory for the subsets
    struct Subset* subsets = (struct Subset*)malloc(V * sizeof(struct Subset));

    // Initialize each subset as a single element set with its rank as 0
    for (int i = 0; i < V; i++) {
        subsets[i].parent = i;
        subsets[i].rank = 0;
    }

    // Sort the edges in non-decreasing order of their weights
    qsort(edges, V - 1, sizeof(struct Edge), compare);

    printf("Edge   Weight\n");

    // Process each edge in sorted order and add it to the MST if it doesn't form a
cycle
    for (int i = 0; i < V - 1; i++) {
        int x = find(subsets, edges[i].src);
        int y = find(subsets, edges[i].dest);
     if (x != y) {
          printf("%d - %d    %d\n", edges[i].src, edges[i].dest, edges[i].weight);
```

```
            Union(subsets, x, y);
        }
    }
    // Free allocated memory
    free(subsets);
}
```

Output :

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

19) Dijkstra's algorithm

Program :

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

#define V 9 // Number of vertices in the graph

// Function to find the vertex with the minimum distance value
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;
}

// Function to print the constructed distance array
void printSolution(int dist[]) {
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t\t %d\n", i, dist[i]);
}

// Dijkstra's algorithm to find the shortest path from source to all vertices
```

```
void dijkstra(int graph[V][V], int src) {
    int dist[V];     // The output array dist[i] holds the shortest distance from src to i
    int sptSet[V];   // sptSet[i] is true if vertex i is included in the shortest
                     // path tree or the shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and sptSet[] as false
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = 0;
    }

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find the shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum distance vertex from the set of vertices
        // not yet processed. u is always equal to src in the first iteration.
        int u = minDistance(dist, sptSet);
        // Mark the picked vertex as processed
        sptSet[u] = 1;
        // Update dist value of the adjacent vertices of the picked vertex
        for (int v = 0; v < V; v++) {
            // Update dist[v] only if it is not in the sptSet, there is an
            // edge from u to v, and the total weight of path from src to
            // v through u is less than the current value of dist[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];
            }
```

```
        }
    }
    // Print the constructed distance array
    printSolution(dist);
}
int main() {
    int graph[V][V] = {
        {0, 4, 0, 0, 0, 0, 0, 8, 0},
        {4, 0, 8, 0, 0, 0, 0, 11, 0},
        {0, 8, 0, 7, 0, 4, 0, 0, 2},
        {0, 0, 7, 0, 9, 14, 0, 0, 0},
        {0, 0, 0, 9, 0, 10, 0, 0, 0},
        {0, 0, 4, 14, 10, 0, 2, 0, 0},
        {0, 0, 0, 0, 0, 2, 0, 1, 6},
        {8, 11, 0, 0, 0, 0, 1, 0, 7},
        {0, 0, 2, 0, 0, 0, 6, 7, 0}
    };
    dijkstra(graph, 0);
    return 0;
}
```

Output:

```
0       0
1       4
2       12
3       19
4       21
5       11
6       9
7       8
8       14
```

20)B-TREE operations

Program :

```
// Searching a key on a B-tree in C

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

#define MAX 3
#define MIN 2

struct BTreeNode {
  int val[MAX + 1], count;
  struct BTreeNode *link[MAX + 1];
};

struct BTreeNode *root;

// Create a node
struct BTreeNode *createNode(int val, struct BTreeNode *child) {
  struct BTreeNode *newNode;
  newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
  newNode->val[1] = val;
  newNode->count = 1;
  newNode->link[0] = root;
  newNode->link[1] = child;
  return newNode;
}

// Insert node
void insertNode(int val, int pos, struct BTreeNode *node,
```

```
      struct BTreeNode *child) {
  int j = node->count;
  while (j > pos) {
    node->val[j + 1] = node->val[j];
    node->link[j + 1] = node->link[j];
    j--;
  }
  node->val[j + 1] = val;
  node->link[j + 1] = child;
  node->count++;
}

// Split node
void splitNode(int val, int *pval, int pos, struct BTreeNode *node,
        struct BTreeNode *child, struct BTreeNode **newNode) {
  int median, j;

  if (pos > MIN)
    median = MIN + 1;
  else
    median = MIN;

  *newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
  j = median + 1;
  while (j <= MAX) {
    (*newNode)->val[j - median] = node->val[j];
    (*newNode)->link[j - median] = node->link[j];
    j++;
  }
  node->count = median;
```

```
  (*newNode)->count = MAX - median;

  if (pos <= MIN) {
    insertNode(val, pos, node, child);
  } else {
    insertNode(val, pos - median, *newNode, child);
  }
  *pval = node->val[node->count];
  (*newNode)->link[0] = node->link[node->count];
  node->count--;
}

// Set the value
int setValue(int val, int *pval,
           struct BTreeNode *node, struct BTreeNode **child) {
  int pos;
  if (!node) {
    *pval = val;
    *child = NULL;
    return 1;
  }

  if (val < node->val[1]) {
    pos = 0;
  } else {
    for (pos = node->count;
       (val < node->val[pos] && pos > 1); pos--)
      ;
    if (val == node->val[pos]) {
      printf("Duplicates are not permitted\n");
```

```
      return 0;
    }
  }
  if (setValue(val, pval, node->link[pos], child)) {
    if (node->count < MAX) {
      insertNode(*pval, pos, node, *child);
    } else {
      splitNode(*pval, pval, pos, node, *child, child);
      return 1;
    }
  }
  return 0;
}

// Insert the value
void insert(int val) {
  int flag, i;
  struct BTreeNode *child;

  flag = setValue(val, &i, root, &child);
  if (flag)
    root = createNode(i, child);
}

// Search node
void search(int val, int *pos, struct BTreeNode *myNode) {
  if (!myNode) {
    return;
  }
```

```
  if (val < myNode->val[1]) {
    *pos = 0;
  } else {
    for (*pos = myNode->count;
       (val < myNode->val[*pos] && *pos > 1); (*pos)--)
      ;
    if (val == myNode->val[*pos]) {
      printf("%d is found", val);
      return;
    }
  }
  search(val, pos, myNode->link[*pos]);

  return;
}

// Traverse then nodes
void traversal(struct BTreeNode *myNode) {
  int i;
  if (myNode) {
    for (i = 0; i < myNode->count; i++) {
      traversal(myNode->link[i]);
      printf("%d ", myNode->val[i + 1]);
    }
    traversal(myNode->link[i]);
  }
}

int main() {
  int val, ch;
```

```
  insert(8);
  insert(9);
  insert(10);
  insert(11);
  insert(15);
  insert(16);
  insert(17);
  insert(18);
  insert(20);
  insert(23);

  traversal(root);

  printf("\n");
  search(11, &ch, root);
}
```

Output :

8 9 10 11 15 16 17 18 20 23

11 is found