Department of Computer Science and Engineering- Data Science

DATA STRUCTURES

LAB MANUAL

Regulations: PR24 - JNTUH

Class: II Year B.Tech. CSE I Semester

LIST OF PROGRAMS

SNO	PROGRAMS

1	Write a program that uses functions to perform the following operations on singly linked list.: i) Creation ii) Insertion iii) Deletion iv) Traversal
2	Write a program that uses functions to perform the following operations on doubly linked list.: i) Creation ii) Insertion iii) Deletion iv) Traversal
3	Write a program that uses functions to perform the following operations on circular linked list.: i) Creation ii) Insertion iii) Deletion iv) Traversal
4	Write a program that implement stack (its operations) using i) Arrays ii) Pointers
5	Write a program that implement Queue (its operations) using i) Arrays ii) Pointers
6	Write a program that implements the following sorting methods to sort a given list of integers in ascending order i) Quick sort ii) Heap sort iii) Merge sort
7	Write a program to implement the tree traversal methods(Recursive and Non Recursive).
8	Write a program to implement i) Binary Search tree ii) B Trees iii) B+ Trees iv) AVL trees v) Red - Black trees.
9	Write a program to implement the graph traversal methods. 10. Implement a Pattern matching algorithms using Boyer- Moore, Knuth-Morris-Prat
10	Implement a Pattern matching algorithms using Boyer- Moore, Knuth-Morris-Prat

1. Write a program that uses functions to perform the following operations on singly linked list.:
i) Creation ii) Insertion iii) Deletion iv) Traversal

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

   int data;
   struct node *next;
};

struct node *head=NULL;
void begin_insert ();
void last_insert ();
void random_insert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search_key();
void main ()
{

   int choice =0;
   while(choice != 9)
   {

      printf("\n\n***Main Menu*\n");
      printf("\nChoose one option from the following list ...\n");
      printf("\n==========================================\n");
      printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete
from Beginning\n5.Delete from last\n6.Delete node after specified
location\n7.Show\n8.Search for an element\n9.Exit\n");
      printf("\nEnter your choice?\n");
      scanf("\n%d",&choice);
      switch(choice)
      {

         case 1:
         begin_insert();
         break;
         case 2:
         last_insert();
         break;
         case 3:
         random_insert();
         break;
         case 4:
         begin_delete();
         break;
         case 5:
```

```
            last_delete();
            break;
            case 6:
            random_delete();
            break;
            case 7: display(); break;
            case 8: search_key();
            break;
            case 9:exit(0);
             break; default:
            printf("Please enter valid choice..");
        }

    }

}

 void begin_insert()
 {

    struct node *newnode;
    newnode = (struct node *) malloc(sizeof(struct
    node *));
printf("\nEnter         value\n");
scanf("%d",&newnode->data);
    if(head== NULL)
    {

    Newnode->next=NULL;

    Head=newnode;

    }

    else

    {

        newnode ->next = head;
        head=newnode;
        printf("\nNode inserted");
    }

}

 void last_insert()
 {

    struct node *newnode,*temp;
    newnode = (struct node *) malloc(sizeof(struct
    node *));
printf("\nEnter         value\n");
scanf("%d",&newnode->data);
    if(head== NULL)
    {
```

```
    Newnode->next=NULL;

    Head=newnode;else
       else
       {

          temp = head;
          while (temp -> next != NULL)
          {

             temp = temp -> next;
          }

temp->next = newnode;
          ptr->next = NULL;
          printf("\nNode inserted");

       }

    }

}

void random_insert()
{

    int i,loc;
    struct node *newnode, *temp;
    newnode= (struct node *) malloc (sizeof(struct
    node));
       printf("\nEnter element value");
       scanf("%d",&ptr->data);
       printf("\nEnter the location after which you want to insert ");
       scanf("\n%d",&loc);
       temp=head;
       for(i=1;i<loc;i++)
       {

          temp = temp->next;
          if(temp == NULL)
          {

             printf("\ncan't insert\n");
             return;
          }

newnode ->next = temp -
>next;
temp ->next = newnode;
printf("\nNode inserted");
    }

}

void begin_delete()
{
```

```
struct node * temp;

temp=head;

if(head == NULL)

   {

      printf("\nList is empty\n");
   }

   Else if(temp->next==NULL)
   {

      Head=NULL;
      Free(temp);
      }
      Else
      {
      temp = head;
      head = temp ->next;
      free(temp);
      printf("\nNode deleted from the begining ...\n");
   }

}

void last_delete()
{

   struct node *temp,temp1;
   if(head == NULL)
   {

      printf("\nlist is empty");
   }

   else if(head -> next == NULL)
   {

      head = NULL;
      free(head);
      printf("\nOnly node of the list deleted ...\n");
   }

   else
   {

      temp = head;
      while(temp->next != NULL)
      {

         temp1 = temp;
         temp = temp ->next;
      }
```

```
        temp 1->next = NULL; free(temp);

        printf("\nDeleted Node from the last ...\n");
    }

}

void random_delete()
{

    struct node *ptr,*ptr1;
    int loc,i;
    printf("\n Enter the location of the node after which you want to perform deletion \n");
    scanf("%d",&loc);
    ptr=head;
    for(i=1;i<loc;i++)
    {

        ptr1 = ptr;
        ptr = ptr->next;

        if(ptr == NULL)
        {

            printf("\nCan't delete");
            return;
        }

    }

    ptr1 ->next = ptr ->next;
    free(ptr);
    printf("\nDeleted node from %d location ",loc);
}

void display()
{

    struct node *temp;
    temp = head;
    if(temp == NULL)
    {

        printf("Nothing to print");
    }

    else
    {

        printf("\nprinting values..... \n");
        while (temp!=NULL)
        {

            printf("\n%d",temp->data);
            temp = temp -> next;
        }
```

```
    }

}

void search_key()
{

    struct node *ptr;
    int key,i=0,flag=0;
    ptr = head;
    if(ptr == NULL)
    {

        printf("\nEmpty List\n");
    }

    else
    {

        printf("\nEnter key element which you want to search?\n");
        scanf("%d",&key);
        while (ptr!=NULL)
        {

            if(ptr->data == key)
            {

                flag=1;
                break;
            }

            i++;
            ptr=ptr->next;
        }

        if(flag==1)
        {

            printf("key element found at location %d", i+1);
        }

        else
        {

            printf("element is not found\n");
        }

      }
```

Output:

*****Main Menu***

Choose one option from the following list ...

==========

1. Insert in beginning
2. Insert at last
3. Insert at any random location
4. Delete from Beginning
5. Delete from last
6. Delete node after specified location
7.Show
8.Search for an element
9.Exit

Enter your choice?

1

Enter value
10

Node inserted
Enter your choice?
2

Enter value?

20

Node inserted
Enter your choice?
7

printing values . . . . .

10

2. Write a program that uses functions to perform the following operations on doubly linked list.:
i) Creation ii) Insertion iii) Deletion iv) Traversal

```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

int count=0;

struct node

{

int data;

struct node *next,*prev;

}*head,*last,*newn,*trav;

//-------------------------------------------------

void create_list()

{

struct node *temp;

int value;

temp=head;

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

printf("\n enter value");

scanf("%d",&value);

newn->data=value;

if(head==NULL)

{

head=last=newn;

head->prev=NULL;

head->next=NULL;

count++;
```

```
}
else
{
   while(temp->next!=NULL)
   {
   temp=temp->next;
   }
newn->next=NULL;
newn->prev=temp;
temp->next=newn;
temp=newn;
count++;
}
}
//-------------------------------------------------
void insert_at_begning(int value)
{
newn=(struct node *)malloc(sizeof (struct node));
newn->data=value;
if(head==NULL)
{
head=last=newn;
head->prev=NULL;
head->next=NULL;
count++;
```

```
}
else
{
newn->next=head;
head->prev=newn;
newn->prev=NULL;
head=newn;
count++;
}
}
//-------------------------------------------------------
void insert_at_end(int value)
{
struct node *temp;
temp=head;
newn=(struct node *)malloc(sizeof (struct node));
newn->data=value;
if(head==NULL)
{
head=last=newn;
head->prev=NULL;
head->next=NULL;
count++;
}
else
```

```
{
   while(temp->next!=NULL)
   {
      temp=temp->next;
   }
newn->next=NULL;
newn->prev=temp;
temp->next=newn;
temp=newn;
count++;
}
}
//-------------------------------------------------------
int insert_at_middle()
{
if(count>=2)
{
struct node *var2,*temp;
int loc,value;
printf("\nselect location where you want to insert the data");
scanf("%d",&loc);
printf("\nenter which value do u want to inserted");
scanf("%d",&value);
newn=(struct node *)malloc(sizeof (struct node));
newn->data=value;
```

```
temp=head;
while(temp->data!=loc)
{
temp=temp->next;
if(temp==NULL)
{
printf("\nSORRY...there is no %d element",loc);
return 0;
}
}
if(temp->next==NULL)
{
printf("\n%d is last node..please enter midddle node values ",loc) ;
return 0;
}
var2=temp->next;
temp->next=newn;
newn->next=var2;
newn->prev=temp;
var2->prev=newn;
count++;
}
else
{
printf("\nthe no of nodes must be >=2");
```

```
}
}
//-----------------------------------------------------------------
int delete_from_middle()
{
if(count>2)
{
struct node *temp,*var;
int value;
temp=head;
printf("\nenter the data that you want to delete from the list shown above");
scanf("%d",&value);
while(temp!=NULL)
{
if(temp->data == value)
{
if(temp->next==NULL)//if(temp==head)
{
printf("\n\n sorry %d is last node ..please enter middle nodes only",value);
return 0;
}
else
{
if(temp==head)
{
```

```
printf("\n\n %d is first node..plz enter middle nodes",value);
return 0;


}
var->next=temp->next;
temp->next->prev=var;
free(temp);
count--;
return 0;
}
}
else
{
var=temp;
temp=temp->next;
}
}
if(temp==NULL)
printf("\n%d is not avilable.. enter only middle elements..",value);
else
printf("\ndata deleted from list is %d",value);
}
else
{
printf("\nthere no middle elemts..only %d elemts is avilable",count);
```

```
}
}
//-----------------------------------------------------------------
int delete_from_front()
{
struct node *temp;
if(head==NULL)
{
printf("no elements for deletion in the list");
return 0;
}
else if(head->next==NULL)
{
printf("deleted element is :%d",head->data);
head=last=NULL;

}
else
{
temp=head->next;
printf("deleted element is :%d",head->data);
head->next=NULL;
temp->prev=NULL;
head=temp;
count--;
```

```
return 0;
}
}
//-------------------------------------------------------
int delete_from_end()
{
struct node *temp,*var;
temp=head;
if(head==NULL)
{
printf("no elemts in the list");
return 0;
}
else
while(temp->next!=NULL)
{
   var=temp;
   temp=temp->next;
}
var->next=NULL;
temp->prev=NULL;

printf("data deleted from list is %d",temp->data);
//head=last=NULL;
free(temp);
```

```
count--;
return 0;
}

int display()
{
trav=head;//head;
if(trav==NULL)
{
printf("\nList is Empty");
return 0;
}
else
{
printf("\n\nElements in the List is %d:\n",count);
while(trav!=NULL)
{
printf("%d<--> ",trav->data);
trav=trav->next;//next;
}
printf("\n");
}
}
//-------------------------------------------------------------
int main()
```

```
{
int ch=0;
char ch1;
// clrscr();
head=NULL;
last=NULL;
while(1)
{
printf("\n Double Linked List Operations");
printf("\n1.Create Double Linked List");
printf("\n2.insertion at begning of linked list");
printf("\n3.insertion at the end of linked list");
printf("\n4.insertion at the middle where you want");
printf("\n5.deletion from the front of linked list");
printf("\n6.deletion from the end of linked list ");
printf("\n7.deletion of the middle data that you want");
printf("\n8.display");
printf("\n9.exit\n");
printf("\nenter the choice of operation to perform on linked list");
scanf("%d",&ch);
switch(ch)
{
case 1:
{
do{
```

```
create_list();
display();
printf("do you want to create list ,y / n");
getchar();
scanf("%c",&ch1);
}while(ch1=='y');
break;
}
case 2:
{
int value;
printf("\nenter the value to be inserted");
scanf("%d",&value);
insert_at_begning(value);
display();
break;
}
case 3:
{
int value;
printf("\nenter value to be inserted");
scanf("%d",&value);
insert_at_end(value);
display();
break;
```

```
}
case 4:
{
insert_at_middle();
display();
break;
}
case 5:
{
delete_from_front();
display();
}break;
case 6:
{
delete_from_end();
display();
break;
}
case 7:
{
display();
delete_from_middle();
display();
break;
}
```

```
case 8:display();break;
case 9:
{
exit(0);
}
}
}
getch();
}
```

3. Write a program that uses functions to perform the following operations on circular linked list.:
i) Creation ii) Insertion iii) Deletion iv) Traversal

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *head=NULL;
void beginsert ();
void lastinsert ();
void insertatspecified();
void begin_delete();
void last_delete();
void delete_from_middle();
void display();
void main ()
{
int choice =0;
printf("\n*********Main Menu*********\n");
printf("\nChoose one option from the following list ...\n");
printf("\n===============================================\n");
printf("\n1.Insert in begining\n2.Insert at last\n 3.Insert at specified location 4.Delete from Beginning\n
5.Delete from last\n6.random_delete\n7.display\n8.Exit\n");
while(choice != 7)
{
printf("\nEnter your choice?\n");
scanf("\n%d",&choice);
switch(choice)
{
case 1:
beginsert();
display();
break;
case 2:
lastinsert();
display();
break;
case 3:
insertatspecified();
```

```
display();
break;
case 4:
begin_delete();
display();
break;
case 5:
last_delete();
display();
break;
case 6:
delete_from_middle();
display();
break;
case 7:
display();
break;
case 8:
exit(0);
break;
default:
printf("Please enter valid choice..");
}
}
}
void beginsert()
{
struct node *newn,*temp;
int value;
temp = head;
newn = (struct node *)malloc(sizeof(struct node));
printf("\nEnter the node data?");
scanf("%d",&value);
newn -> data = value;
if(head == NULL)
{
head = newn;
newn -> next = head;
}
else
{
while(temp->next != head)
```

```
{
temp = temp->next;
}
newn->next = head;
temp -> next = newn;
head = newn;
}
printf("\nnode inserted\n");
}
void lastinsert()
{
struct node *newn,*temp;
int value;
temp=head;
newn = (struct node *)malloc(sizeof(struct node));
printf("\nEnter Data?");
scanf("%d",&value);
newn->data = value;
if(head == NULL)
{
head = newn;
newn -> next = head;
}
else
{
 while(temp -> next != head)
{
temp = temp -> next;
}
temp -> next = newn;
newn -> next = head;
temp=newn;
}
printf("\nnode inserted\n");
}
void insertatspecified()
{
 struct node *newn,*temp,*var;
int value,loc;
temp=head;
newn = (struct node *)malloc(sizeof(struct node));
printf("enter the specified location value");
```

```
scanf("%d",&loc);
printf("\nEnter Data?");
scanf("%d",&value);
newn->data = value;
if(head == NULL)
{
head = newn;
newn -> next = head;
}
else
{
   while(temp->data!=loc)
   {
      var=temp;
      temp=temp->next;
   }
   newn->next=var->next;
   var->next=newn;
}
printf("node inserted");
}
void begin_delete()
{
struct node *temp,*var;
temp=var=head;
if(temp == NULL)
{
printf("\nLinked list is empty");
}
else
{
while(var->next != head)
{
var=var->next;
}
head=temp->next;
temp->next=NULL;
var->next=head;
printf("\nnode deleted\n");
}
}
```

```
void last_delete()
{
struct node *temp,*var;
temp=head;
if(temp==NULL)
{
printf("\nUNDERFLOW");
}
else
{
   while(temp->next!=head)
{
var=temp;
temp=temp->next;
}
var->next = temp -> next;
temp->next=NULL;
free(temp);
printf("\nnode deleted\n");
}
}
void delete_from_middle()
{
struct node *temp,*var;
int value;
temp=head;
printf("\nenter the data that you want to delete from the list shown above");
scanf("%d",&value);
if(temp==NULL)
{
printf("\nSORRY...there is no %d element",value);
return 0;
}
else
{
while(temp->data!=value)
{
var=temp;
temp=temp->next;
}
var->next=temp->next;
temp->next=NULL;
```

```
}
printf("\ndata deleted from list is %d",value);
free(temp);
}
void display()
{
struct node *trav;
trav=head;
if(head == NULL)
{
printf("\nnothing to print");
}
else
{
printf("\n printing values ... \n");
while(trav -> next != head)
{
printf("%d\n", trav -> data);
trav = trav -> next;
}
printf("%d\n", trav -> data);
}
}
```

4. Write a program that implement stack (its operations) using
i) Arrays ii) Pointers

Using Arrays

```
#include<stdio.
h>
#include<conio.
h>
#include<stdlib.
h> #define
SIZE 10 void
push(int); void
pop();

void
display();
void peek();

int stack[SIZE], top = -1;
void main()

{

   int value, choice;

   //clrscr();
   while(1){

      printf("\n\n*** MENU ***\n");

      printf("1. Push\n2. Pop\n3. Display\n4.peek\n5. Exit");
      printf("\nEnter your choice: ");

      scanf("%d",&choice);
      switch(choice)
```

```
      {
case 1: printf("Enter the value to
be insert: ");
scanf("%d",&value);
push(value);
break;
case 2: pop();
                    break;
case 3: display();
                    b
     reak; case 4:
     peek();
           break;
          case 5: exit(0);
                 break;
           default: printf("\nWrong selection!!! Try again!!!");
      }
   }
}
void push(int
   value){ if(top ==
   SIZE-1)
      printf("\nStack is Full!!! Insertion is not possible!!!"); else{
      top++;
      stack[top] = value;
      printf("\nInsertion
      success!!!");
   }
```

```
}
void pop()
{
if(top == -1)
{
      printf("\nStack is Empty!!! Deletion is not possible!!!");
      }
       Else
      {
      printf("\nDeleted : %d", stack[top]);
      top--;
   }
}
void peek()
{
    if(top == -1)
      printf("\nStack is Empty!!!");
      else
      printf("the peek/top element is %d", stack[top]);
}
void
   display()
{
if(top == -1)
{
      printf("\nStack is Empty!!!");
```

```
    }
    else
    {
    int i;
    printf("\nStack elements are:\n");
    for(i=top; i>=0; i--)
        printf("%d\n",stack[i]);
  }
}
```

Expected output:

***** MENU *****

1. Push
2. Pop
3. D
ispl
ay
4.pe
ek
5. Exit

Enter your choice: 1

Enter the value to be insert: 10

Insertion success!!!

***** MENU *****

1. Push
2. Pop
3. D
ispl
ay
4.pe
ek
5. Exit

Enter your choice: 1

Enter the value to be insert: 20

Insertion success!!!

***** MENU *****

1. Push
2. Pop
3. D
ispl
ay
4.pe
ek
5. Exit

Enter your choice: 1

Enter the value to be insert: 30

Insertion success!!!

***** MENU *****

1. Push
2. Pop
3. D
ispl
ay
4.pe
ek
5. Exit

Enter your choice: 1

Enter the value to be insert: 40

Insertion success!!!

***** MENU *****

1. Push
2. Pop
3. D

ispl
ay
4.pe
ek
5. Exit

Enter your choice: 1

Enter the value to be insert: 50

Insertion success!!!

***** MENU *****

1. Push
2. Pop
3. D
ispl
ay
4.pe
ek
5. Exit

Enter your

choice: 2

Deleted : 50

***** MENU *****

1. Push
2. Pop
3. D
ispl
ay
4.pe
ek
5. Exit

Enter your choice: 3

Stack
elements are:

40

30

20

10

Using Pointers

```
#include<std
io.h>
#include<co
nio.h>
```

```
#include<std
lib.h>

struct Node
{

  int data;
  struct Node *next;
}*top = NULL;

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

void main()
{

  int choice, value;
  //clrscr();
  printf("\n:: Stack using Linked List
  ::\n"); while(1){
    printf("\n** MENU **\n");
    printf("1. Push\n2. Pop\n3. Display\n4.Peek\n5.
    Exit\n"); printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
        case 1: printf("Enter the value to be insert: ");
                scanf("%d", &value);
                push(val
                ue);
                break;
        case 2: pop();
        break; case 3:
        display(); break;
        case 4:peek();
              bre
        ak; case
        5:exit(0
        );
        default: printf("\nWrong selection!!! Please try again!!!\n");
```

```
    }

  }

}

void push(int value)
{

  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct
  Node)); newNode->data = value;
  if(top == NULL)
    newNode->next =
  NULL; else
    newNode->next
  = top; top =
  newNode;
  printf("\nInsertion is Success!!!\n");
}

void pop()
{

  if(top == NULL)
    printf("\nStack is
  Empty!!!\n"); else{
    struct Node *temp = top;
    printf("\nDeleted element: %d", temp->data); top = temp->next;
    free(temp);
  }

}

void peek()
{

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

void display()
{

  if(top == NULL)
```

```
    printf("\nStack is
  Empty!!!\n"); else{
    struct Node *temp = top;
    while(temp!= NULL)
        {

        printf("%d\t",temp-
        >data); temp = temp -
        > next;
    }

  }

}
```

5. Write a program that implement queue (its operations using) i) Arrays ii) Pointers

program that implement queue using Arrays

```
#include <stdio.h>

      #define MAX 50

      void insert();
   void delete();
      void display();
      int queue_array[MAX];
      int rear = - 1;
      int front = - 1;
      main()
      {
          int choice;
         while (1)
         {
             printf("1.Insert element to queue \n");
             printf("2.Delete element from queue \n");
              printf("3.Display all elements of queue \n");
             printf("4.Quit \n");
              printf("Enter your choice : ");
              scanf("%d", &choice);
             switch (choice)
             {
                  case 1:
                  insert();
                  break;
                  case 2:
                  delete();
                 break;
                  case 3:
                  display();
                  break;
                  case 4:
                 exit(1);
                 default:
                 printf("Wrong choice \n");
             } /* End of switch */
         } /* End of while */
} /* End of main() */

      void insert()
      {
          int add_item;
         if (rear == MAX - 1)
          printf("Queue Overflow \n");
         else
         {
              if (front == - 1)
              /*If queue is initially empty */
              front = 0;
             printf("Inset the element in queue : ");
```

```
        scanf("%d", &add_item);
          rear = rear + 1;
          queue_array[rear] = add_item;
      }
    } /* End of insert() */

void delete()
{
    if (front == - 1 || front > rear)
    {
        printf("Queue Underflow \n");
        return ;
    }
    else
    {
        printf("Element    deleted    from    queue    is    :    %d\n",
queue_array[front]);
        front = front + 1;
    }
} /* End of delete() */

void display()
{
    int i;
    if (front == - 1)
        printf("Queue is empty \n");
    else
    {
        printf("Queue is : \n");
        for (i = front; i <= rear; i++)
            printf("%d ", queue_array[i]);
        printf("\n");
    }
} /* End of display() */
```

Ex:2

```
Queue using Array
1.Insertion
2.Deletion
3.Display
4.Exit
Enter the Choice:1

 Enter no 1:10

Enter the Choice:1

 Enter no 2:54

Enter the Choice:1

 Enter no 3:98
```

```
Enter the Choice:1

 Enter no 4:234

Enter the Choice:3

 Queue Elements are:
 10
54
98
234

Enter the Choice:2

 Deleted Element is 10
Enter the Choice:3

 Queue Elements are:
 54
98
234

Enter the Choice:4
```

Program that implement queue using pointers

```
#include<stdlib.h>

#include<stdio.h>

#include<conio.h>

struct queue

{

int data;

struct queue *next;

}*front=NULL,*rear=NULL;

void add();

int del();

void display();

void main()

{
```

```
int ch;
printf(" \n MENU \n");
printf("\n1.Insert an element in Queue\n");
printf("\n2.Delete an element from Queue\n");
printf("\n3.Display the Queue\n");
printf("\n4.Exit!\n");
while(1)
{
printf("\nEnter your choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:add();
display();
break;
case 2:del();
display();
break;
case 3:display();
getch();
break;
case 4:exit(0);
break;
default:printf("\nYou entered wrong choice");
getch();
```

```
break;
}
}
getch();
}
void add()
{
struct queue *newn;
int value;
newn=(struct queue*)malloc(sizeof(struct queue));
printf("\nEnter the element:");
scanf("%d",&value);
newn->data=value;
newn->next=NULL;
if(front==NULL&&rear==NULL)
{
rear=front=newn;
}
else
{
rear->next=newn;
rear=newn;
}
}
int del()
```

```
{
struct queue *temp;
temp=front;
if(front==NULL)
{
printf("\nQueue is Empty");
return(0);
}
else
{
printf("deleted data %d\n",front->data);
front=front->next;
free(temp);
}
return;
}
void display()
{
struct queue *temp;
if(front==NULL)
{
printf("queue is empty");
}
else
{
```

```
temp=front;
while(temp->next!=NULL)
{
printf("%d \t ",temp->data);
temp=temp->next;
}
printf("\t %d",temp->data);
}
getch();
}
```

6. Write a program that implements the following sorting methods to sort a given list of integers in ascending order

i) Quick sort ii) Heap sort iii) Merge sort

```
#include <stdio.h>
// Function to swap two elements
void swap (int* a, int* b) {
   int t = *a;
   *a = *b;
   *b = t;
}
int partition(int arr[], int low, int high) {
   int pivot = arr[high];
   int i = (low - 1);

   for (int j = low; j <= high - 1; j++) {
      if (arr[j] < pivot) {
         i++;
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}
void quickSort(int arr[], int low, int high) {
   if (low < high) {
      int pi = partition(arr, low, high);
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
   }
}
```

```
// Function to print the array
void printArray(int arr[], int size) {
   int i;
   for (i = 0; i < size; i++)
      printf("%d ", arr[i]);
   printf("\n");
}

int main() {
   int arr[] = { 12, 17, 6, 25, 1, 5 };
   int n = sizeof(arr) / sizeof(arr[0]);
   quickSort(arr, 0, n - 1);
   printf("Sorted array: \n");
   printArray(arr, n);
   return 0;
}
```

Output:

1 5 6 12 17 25

Heap sort:

```
#include <stdio.h>
/* function to heapify a subtree. Here 'i' is the
index of root node in array a[], and 'n' is the size of heap. */
void heapify(int a[], 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 && a[left] > a[largest])
      largest = left;
   // If right child is larger than root
   if (right < n && a[right] > a[largest])
```

```
      largest = right;
   // If root is not largest
   if (largest != i) {
      // swap a[i] with a[largest]
      int temp = a[i];
      a[i] = a[largest];
      a[largest] = temp;

      heapify(a, n, largest);
   }
}
/*Function to implement the heap sort*/
void heapSort(int a[], int n)
{
   for (int i = n / 2 - 1; i >= 0; i--)
      heapify(a, n, i);
   // One by one extract an element from heap
   for (int i = n - 1; i >= 0; i--) {
      /* Move current root element to end*/
      // swap a[0] with a[i]
      int temp = a[0];
      a[0] = a[i];
      a[i] = temp;

      heapify(a, i, 0);
   }
}
/* function to print the array elements */
void printArr(int arr[], int n)
{
   for (int i = 0; i < n; ++i)
   {
      printf("%d", arr[i]);
      printf(" ");
```

```
    }

  }
  int main()
  {
    int a[] = {48, 10, 23, 43, 28, 26, 1};
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    heapSort(a, n);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);
    return 0;
  }
```

Output:

Before sorting array elements are -

48 10 23 43 28 26 1

After sorting array elements are -

1 10 23 26 28 43 48

Merge sort:

```
#include <stdio.h>

/* Function to merge the subarrays of a[] */
void merge(int a[], int beg, int mid, int end)
{
  int i, j, k;
  int n1 = mid - beg + 1;
  int n2 = end - mid;

  int LeftArray[n1], RightArray[n2]; //temporary arrays
```

```
/* copy data to temp arrays */
for (int i = 0; i < n1; i++)
LeftArray[i] = a[beg + i];
for (int j = 0; j < n2; j++)
RightArray[j] = a[mid + 1 + j];

i = 0; /* initial index of first sub-array */
j = 0; /* initial index of second sub-array */
k = beg;  /* initial index of merged sub-array */

while (i < n1 && j < n2)
{
   if(LeftArray[i] <= RightArray[j])
   {
      a[k] = LeftArray[i];
      i++;
   }
   else
   {
      a[k] = RightArray[j];
      j++;
   }
   k++;
}
while (i<n1)
{
   a[k] = LeftArray[i];
   i++;
   k++;
}

while (j<n2)
{
```

```
      a[k] = RightArray[j];
      j++;
      k++;
   }
}

void mergeSort(int a[], int beg, int end)
{
   if (beg < end)
   {
      int mid = (beg + end) / 2;
      mergeSort(a, beg, mid);
      mergeSort(a, mid + 1, end);
      merge(a, beg, mid, end);
   }
}

/* Function to print the array */
void printArray(int a[], int n)
{
   int i;
   for (i = 0; i < n; i++)
      printf("%d ", a[i]);
   printf("\n");
}

int main()
{
   int a[] = { 12, 31, 25, 8, 32, 17, 40, 42 };
   int n = sizeof(a) / sizeof(a[0]);
   printf("Before sorting array elements are - \n");
   printArray(a, n);
   mergeSort(a, 0, n - 1);
   printf("After sorting array elements are - \n");
```

```
    printArray(a, n);
    return 0;
}
```

Output:

Before sorting array elements are -

12 31 25 8 32 17 40 42

After sorting array elements are -

8 12 17 25 31 32 40 42

7)Write a program to implement the tree traversal methods (Recursive and Non Recursive)

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

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

// Inorder traversal
void inorderTraversal(struct node* root) {
  if (root == NULL) return;
  inorderTraversal(root->left);
  printf("%d ->", root->item);
  inorderTraversal(root->right);
}

// preorderTraversal traversal
void preorderTraversal(struct node* root) {
  if (root == NULL) return;
  printf("%d ->", root->item);
  preorderTraversal(root->left);
  preorderTraversal(root->right);
}

// postorderTraversal traversal
void postorderTraversal(struct node* root) {
  if (root == NULL) return;
  postorderTraversal(root->left);
  postorderTraversal(root->right);
  printf("%d ->", root->item);
}
```

```
// Create a new Node
struct node* createNode(value) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->item = value;
  newNode->left = NULL;
  newNode->right = NULL;

  return newNode;
}

// Insert on the left of the node
struct node* insertLeft(struct node* root, int value) {
  root->left = createNode(value);
  return root->left;
}

// Insert on the right of the node
struct node* insertRight(struct node* root, int value) {
  root->right = createNode(value);
  return root->right;
}

int main() {
  struct node* root = createNode(1);
  insertLeft(root, 12);
  insertRight(root, 9);

  insertLeft(root->left, 5);
  insertRight(root->left, 6);

  printf("Inorder traversal \n");
  inorderTraversal(root);

  printf("\nPreorder traversal \n");
  preorderTraversal(root);

  printf("\nPostorder traversal \n");
  postorderTraversal(root);
}
```

Output:

Inorder traversal
5 ->12 ->6 ->1 ->9 ->
Preorder traversal
1 ->12 ->5 ->6 ->9 ->
Postorder traversal
5 ->6 ->12 ->9 ->1 ->

8) **Write a program to implement**
i) Binary Search tree ii) B Trees iii) B+ Trees iv) AVL trees v) Red - Black trees

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

struct node {
  int data;
  struct node *right_child;
  struct node *left_child;
};

struct node* new_node(int x){
  struct node *temp;
  temp = malloc(sizeof(struct node));
  temp->data = x;
  temp->left_child = NULL;
  temp->right_child = NULL;

  return temp;
}

struct node* search(struct node * root, int x){
  if (root == NULL || root->data == x)
    return root;
  else if (x > root->data)
    return search(root->right_child, x);
  else
    return search(root->left_child, x);
}

struct node* insert(struct node * root, int x){
  if (root == NULL)
    return new_node(x);
  else if (x > root->data)
    root->right_child = insert(root->right_child, x);
```

```
  else
    root -> left_child = insert(root->left_child, x);
  return root;
}
struct node* find_minimum(struct node * root) {
  if (root == NULL)
    return NULL;
  else if (root->left_child != NULL)
    return find_minimum(root->left_child);
  return root;
}

struct node* delete(struct node * root, int x) {

  if (root == NULL)
    return NULL;
  if (x > root->data)
    root->right_child = delete(root->right_child, x);
  else if (x < root->data)
    root->left_child = delete(root->left_child, x);
  else {
    if (root->left_child == NULL && root->right_child == NULL){
      free(root);
      return NULL;
    }
    else if (root->left_child == NULL || root->right_child == NULL){
      struct node *temp;
      if (root->left_child == NULL)
        temp = root->right_child;
      else
        temp = root->left_child;
      free(root);
      return temp;
    }
```

```
    else {
      struct node *temp = find_minimum(root->right_child);
      root->data = temp->data;
      root->right_child = delete(root->right_child, temp->data);
    }
  }
  return root;
}

void inorder(struct node *root){
  if (root != NULL)
  {
    inorder(root->left_child);
    printf(" %d ", root->data);
    inorder(root->right_child);
  }
}

int main() {
  struct node *root;
  root = new_node(20);
  insert(root, 5);
  insert(root, 1);
  insert(root, 15);
  insert(root, 9);
  insert(root, 7);
  insert(root, 12);
  insert(root, 30);
  insert(root, 25);
  insert(root, 40);
  insert(root, 45);
  insert(root, 42);

  inorder(root);
```

```
        printf("\n");

        root = delete(root, 1);

        root = delete(root, 40);

        root = delete(root, 45);

        root = delete(root, 9);

        inorder(root);
        printf("\n");

        return 0;
      }
```

Output:

1 5 7 9 12 15 20 25 30 40 42 45

5 7 12 15 20 25 30 42

B Trees :

```
// including required header files
#include <stdio.h>
#include <stdlib.h>

// defining required macros
#define MAX 3
#define MIN 2

// defining the structure of the node
struct btree_node {
```

```
  int data_item[MAX + 1], counter;
  struct btree_node *the_link[MAX + 1];
};

struct btree_node *root_node;

// creating a Node
struct btree_node *create_node(int data_item, struct btree_node *child_node) {
  struct btree_node *new_node;
  new_node = (struct btree_node *)malloc(sizeof(struct btree_node));
  new_node -> data_item[1] = data_item;
  new_node -> counter = 1;
  new_node -> the_link[0] = root_node;
  new_node -> the_link[1] = child_node;
  return new_node;
}

// insert
void insert_value(int data_item, int position, struct btree_node *the_node,
        struct btree_node *child_node) {
  int j = the_node -> counter;
  while (j > position) {
    the_node -> data_item[j + 1] = the_node -> data_item[j];
    the_node -> the_link[j + 1] = the_node -> the_link[j];
    j--;
```

```
  }
  the_node -> data_item[j + 1] = data_item;
  the_node -> the_link[j + 1] = child_node;
  the_node -> counter++;
}

// splitting the node
void splitNode(int data_item, int *p_value, int position, struct btree_node *the_node,
        struct btree_node *child_node, struct btree_node **new_node) {
  int median_key, j;

  if (position > MIN)
    median_key = MIN + 1;
  else
    median_key = MIN;

  *new_node = (struct btree_node *)malloc(sizeof(struct btree_node));
  j = median_key + 1;
  while (j <= MAX) {
    (*new_node)->data_item[j - median_key] = the_node -> data_item[j];
    (*new_node)->the_link[j - median_key] = the_node -> the_link[j];
    j++;
  }
  the_node -> counter = median_key;
  (*new_node) -> counter = MAX - median_key;
```

```
  if (position <= MIN) {
    insert_value(data_item, position, the_node, child_node);
  } else {
    insert_value(data_item, position - median_key, *new_node, child_node);
  }
  *p_value = the_node -> data_item[the_node->counter];
  (*new_node) -> the_link[0] = the_node -> the_link[the_node->counter];
  the_node -> counter--;
}

// setting the value of node
int set_node_value(int data_item, int *p_value,
          struct btree_node *the_node, struct btree_node **child_node) {
  int position;
  if (!the_node) {
    *p_value = data_item;
    *child_node = NULL;
    return 1;
  }

  if (data_item < the_node -> data_item[1]) {
    position = 0;
  } else {
    for (position = the_node -> counter;
```

```
      (data_item < the_node -> data_item[position] && position > 1); position--)
     ;
   if (data_item == the_node -> data_item[position]) {
    printf("Duplicates are not allowed\n");
    return 0;
   }
  }
  if (set_node_value(data_item, p_value, the_node->the_link[position], child_node)) {
   if (the_node->counter < MAX) {
    insert_value(*p_value, position, the_node, *child_node);
   } else {
    splitNode(*p_value, p_value, position, the_node, *child_node, child_node);
    return 1;
   }
  }
  return 0;
}

// insertion operation
void insertion_operation(int data_item) {
  int the_flag, i;
  struct btree_node *child_node;

  the_flag = set_node_value(data_item, &i, root_node, &child_node);
  if (the_flag)
```

```
   root_node = create_node(i, child_node);
}

// copying the successor
void copy_successor(struct btree_node *my_node, int position) {
  struct btree_node *dummy_node;
  dummy_node = my_node -> the_link[position];

  for (; dummy_node -> the_link[0] != NULL;)
    dummy_node = dummy_node -> the_link[0];
  my_node -> data_item[position] = dummy_node -> data_item[1];
}

// performing the rightshift
void right_shift(struct btree_node *my_node, int position) {
  struct btree_node *x = my_node -> the_link[position];
  int j = x -> counter;

  while (j > 0) {
    x -> data_item[j + 1] = x -> data_item[j];
    x -> the_link[j + 1] = x -> the_link[j];
  }
  x -> data_item[1] = my_node->data_item[position];
  x -> the_link[1] = x->the_link[0];
  x -> counter++;
```

```
  x = my_node -> the_link[position - 1];
  my_node -> data_item[position] = x -> data_item[x -> counter];
  my_node -> the_link[position] = x -> the_link[x -> counter];
  x -> counter--;
  return;
}

// performing the leftshift
void left_shift(struct btree_node *my_node, int position) {
  int j = 1;
  struct btree_node *x = my_node -> the_link[position - 1];

  x -> counter++;
  x -> data_item[x -> counter] = my_node -> data_item[position];
  x -> the_link[x -> counter] = my_node -> the_link[position] -> the_link[0];

  x = my_node -> the_link[position];
  my_node -> data_item[position] = x -> data_item[1];
  x -> the_link[0] = x -> the_link[1];
  x -> counter--;

  while (j <= x -> counter) {
    x -> data_item[j] = x -> data_item[j + 1];
    x -> the_link[j] = x -> the_link[j + 1];
```

```
   j++;
  }
  return;
}

// merging the nodes
void merge_nodes(struct btree_node *my_node, int position) {
  int j = 1;
  struct btree_node *x1 = my_node -> the_link[position], *x2 = my_node -> the_link[position - 1];

  x2 -> counter++;
  x2 -> data_item[x2 -> counter] = my_node -> data_item[position];
  x2 -> the_link[x2 -> counter] = my_node -> the_link[0];

  while (j <= x1 -> counter) {
    x2 -> counter++;
    x2 -> data_item[x2 -> counter] = x1 -> data_item[j];
    x2 -> the_link[x2 -> counter] = x1 -> the_link[j];
    j++;
  }

  j = position;
  while (j < my_node -> counter) {
    my_node -> data_item[j] = my_node -> data_item[j + 1];
    my_node -> the_link[j] = my_node -> the_link[j + 1];
```

```
    j++;
  }
  my_node -> counter--;
  free(x1);
}

// adjusting the node
void adjustNode(struct btree_node *my_node, int position) {
  if (!position) {
    if (my_node -> the_link[1] -> counter > MIN) {
      left_shift(my_node, 1);
    } else {
      merge_nodes(my_node, 1);
    }
  } else {
    if (my_node -> counter != position) {
      if (my_node -> the_link[position - 1] -> counter > MIN) {
        right_shift(my_node, position);
      } else {
        if (my_node -> the_link[position + 1] -> counter > MIN) {
          left_shift(my_node, position + 1);
        } else {
          merge_nodes(my_node, position);
        }
      }
```

```
    } else {
      if (my_node -> the_link[position - 1] -> counter > MIN)
        right_shift(my_node, position);
      else
        merge_nodes(my_node, position);
    }
  }
}

// Traversing the tree
void tree_traversal(struct btree_node *my_node) {
  int i;
  if (my_node) {
    for (i = 0; i < my_node -> counter; i++) {
      tree_traversal(my_node -> the_link[i]);
      printf("%d ", my_node -> data_item[i + 1]);
    }
    tree_traversal(my_node -> the_link[i]);
  }
}

// main function
int main() {
  int data_item, ch;
```

```
  insertion_operation(4);
  insertion_operation(6);
  insertion_operation(2);
  insertion_operation(8);
  insertion_operation(10);
  insertion_operation(9);
  insertion_operation(1);
  insertion_operation(3);
  insertion_operation(12);
  insertion_operation(11);
  insertion_operation(13);

  printf("The B Tree is : ");
  tree_traversal(root_node);
}
```

Output:

The B Tree is : 1 2 3 4 6 8 9 10 11 12 13.

B+ Trees:

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

```
// Default order
#define ORDER 3

typedef struct record {
  int value;
} record;

// Node
typedef struct node {
  void **pointers;
  int *keys;
  struct node *parent;
  bool is_leaf;
  int num_keys;
  struct node *next;
} node;

int order = ORDER;
node *queue = NULL;
bool verbose_output = false;

// Enqueue
void enqueue(node *new_node);

// Dequeue
```

```
node *dequeue(void);
int height(node *const root);
int pathToLeaves(node *const root, node *child);
void printLeaves(node *const root);
void printTree(node *const root);
void findAndPrint(node *const root, int key, bool verbose);
void findAndPrintRange(node *const root, int range1, int range2, bool verbose);
int findRange(node *const root, int key_start, int key_end, bool verbose,
      int returned_keys[], void *returned_pointers[]);
node *findLeaf(node *const root, int key, bool verbose);
record *find(node *root, int key, bool verbose, node **leaf_out);
int cut(int length);

record *makeRecord(int value);
node *makeNode(void);
node *makeLeaf(void);
int getLeftIndex(node *parent, node *left);
node *insertIntoLeaf(node *leaf, int key, record *pointer);
node *insertIntoLeafAfterSplitting(node *root, node *leaf, int key,
            record *pointer);
node *insertIntoNode(node *root, node *parent,
        int left_index, int key, node *right);
node *insertIntoNodeAfterSplitting(node *root, node *parent,
            int left_index,
            int key, node *right);
```

```
node *insertIntoParent(node *root, node *left, int key, node *right);
node *insertIntoNewRoot(node *left, int key, node *right);
node *startNewTree(int key, record *pointer);
node *insert(node *root, int key, int value);

// Enqueue
void enqueue(node *new_node) {
  node *c;
  if (queue == NULL) {
    queue = new_node;
    queue->next = NULL;
  } else {
    c = queue;
    while (c->next != NULL) {
      c = c->next;
    }
    c->next = new_node;
    new_node->next = NULL;
  }
}

// Dequeue
node *dequeue(void) {
  node *n = queue;
  queue = queue->next;
```

```
  n->next = NULL;
  return n;
}

// Print the leaves
void printLeaves(node *const root) {
  if (root == NULL) {
    printf("Empty tree.\n");
    return;
  }
  int i;
  node *c = root;
  while (!c->is_leaf)
    c = c->pointers[0];
  while (true) {
    for (i = 0; i < c->num_keys; i++) {
      if (verbose_output)
        printf("%p ", c->pointers[i]);
      printf("%d ", c->keys[i]);
    }
    if (verbose_output)
      printf("%p ", c->pointers[order - 1]);
    if (c->pointers[order - 1] != NULL) {
      printf(" | ");
      c = c->pointers[order - 1];
```

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

// Calculate height
int height(node *const root) {
  int h = 0;
  node *c = root;
  while (!c->is_leaf) {
    c = c->pointers[0];
    h++;
  }
  return h;
}

// Get path to root
int pathToLeaves(node *const root, node *child) {
  int length = 0;
  node *c = child;
  while (c != root) {
    c = c->parent;
    length++;
  }
```

```
  return length;
}

// Print the tree
void printTree(node *const root) {
  node *n = NULL;
  int i = 0;
  int rank = 0;
  int new_rank = 0;

  if (root == NULL) {
    printf("Empty tree.\n");
    return;
  }
  queue = NULL;
  enqueue(root);
  while (queue != NULL) {
    n = dequeue();
    if (n->parent != NULL && n == n->parent->pointers[0]) {
      new_rank = pathToLeaves(root, n);
      if (new_rank != rank) {
        rank = new_rank;
        printf("\n");
      }
    }
```

```
    if (verbose_output)
      printf("(%p)", n);
    for (i = 0; i < n->num_keys; i++) {
      if (verbose_output)
        printf("%p ", n->pointers[i]);
      printf("%d ", n->keys[i]);
    }
    if (!n->is_leaf)
      for (i = 0; i <= n->num_keys; i++)
        enqueue(n->pointers[i]);
    if (verbose_output) {
      if (n->is_leaf)
        printf("%p ", n->pointers[order - 1]);
      else
        printf("%p ", n->pointers[n->num_keys]);
    }
    printf("| ");
  }
  printf("\n");
}

// Find the node and print it
void findAndPrint(node *const root, int key, bool verbose) {
  node *leaf = NULL;
  record *r = find(root, key, verbose, NULL);
```

```
  if (r == NULL)
    printf("Record not found under key %d.\n", key);
  else
    printf("Record at %p -- key %d, value %d.\n",
        r, key, r->value);
}

// Find and print the range
void findAndPrintRange(node *const root, int key_start, int key_end,
           bool verbose) {
  int i;
  int array_size = key_end - key_start + 1;
  int returned_keys[array_size];
  void *returned_pointers[array_size];
  int num_found = findRange(root, key_start, key_end, verbose,
              returned_keys, returned_pointers);
  if (!num_found)
    printf("None found.\n");
  else {
    for (i = 0; i < num_found; i++)
      printf("Key: %d   Location: %p  Value: %d\n",
          returned_keys[i],
          returned_pointers[i],
          ((record *)
           returned_pointers[i])
```

```
          ->value);
  }
}

// Find the range
int findRange(node *const root, int key_start, int key_end, bool verbose,
       int returned_keys[], void *returned_pointers[]) {
  int i, num_found;
  num_found = 0;
  node *n = findLeaf(root, key_start, verbose);
  if (n == NULL)
    return 0;
  for (i = 0; i < n->num_keys && n->keys[i] < key_start; i++)
    ;
  if (i == n->num_keys)
    return 0;
  while (n != NULL) {
    for (; i < n->num_keys && n->keys[i] <= key_end; i++) {
      returned_keys[num_found] = n->keys[i];
      returned_pointers[num_found] = n->pointers[i];
      num_found++;
    }
    n = n->pointers[order - 1];
    i = 0;
  }
```

```
  return num_found;
}

// Find the leaf
node *findLeaf(node *const root, int key, bool verbose) {
  if (root == NULL) {
    if (verbose)
      printf("Empty tree.\n");
    return root;
  }
  int i = 0;
  node *c = root;
  while (!c->is_leaf) {
    if (verbose) {
      printf("[");
      for (i = 0; i < c->num_keys - 1; i++)
        printf("%d ", c->keys[i]);
      printf("%d] ", c->keys[i]);
    }
    i = 0;
    while (i < c->num_keys) {
      if (key >= c->keys[i])
        i++;
      else
        break;
```

```
    }
    if (verbose)
      printf("%d ->\n", i);
    c = (node *)c->pointers[i];
  }
  if (verbose) {
    printf("Leaf [");
    for (i = 0; i < c->num_keys - 1; i++)
      printf("%d ", c->keys[i]);
    printf("%d] ->\n", c->keys[i]);
  }
  return c;
}

record *find(node *root, int key, bool verbose, node **leaf_out) {
  if (root == NULL) {
    if (leaf_out != NULL) {
      *leaf_out = NULL;
    }
    return NULL;
  }

  int i = 0;
  node *leaf = NULL;
```

```
  leaf = findLeaf(root, key, verbose);


  for (i = 0; i < leaf->num_keys; i++)
    if (leaf->keys[i] == key)
      break;
  if (leaf_out != NULL) {
    *leaf_out = leaf;
  }
  if (i == leaf->num_keys)
    return NULL;
  else
    return (record *)leaf->pointers[i];
}


int cut(int length) {
  if (length % 2 == 0)
    return length / 2;
  else
    return length / 2 + 1;
}


record *makeRecord(int value) {
  record *new_record = (record *)malloc(sizeof(record));
  if (new_record == NULL) {
    perror("Record creation.");
```

```
    exit(EXIT_FAILURE);
  } else {
    new_record->value = value;
  }
  return new_record;
}

node *makeNode(void) {
  node *new_node;
  new_node = malloc(sizeof(node));
  if (new_node == NULL) {
    perror("Node creation.");
    exit(EXIT_FAILURE);
  }
  new_node->keys = malloc((order - 1) * sizeof(int));
  if (new_node->keys == NULL) {
    perror("New node keys array.");
    exit(EXIT_FAILURE);
  }
  new_node->pointers = malloc(order * sizeof(void *));
  if (new_node->pointers == NULL) {
    perror("New node pointers array.");
    exit(EXIT_FAILURE);
  }
  new_node->is_leaf = false;
```

```
  new_node->num_keys = 0;
  new_node->parent = NULL;
  new_node->next = NULL;
  return new_node;
}

node *makeLeaf(void) {
  node *leaf = makeNode();
  leaf->is_leaf = true;
  return leaf;
}

int getLeftIndex(node *parent, node *left) {
  int left_index = 0;
  while (left_index <= parent->num_keys &&
      parent->pointers[left_index] != left)
    left_index++;
  return left_index;
}

node *insertIntoLeaf(node *leaf, int key, record *pointer) {
  int i, insertion_point;

  insertion_point = 0;
  while (insertion_point < leaf->num_keys && leaf->keys[insertion_point] < key)
```

```
  insertion_point++;

 for (i = leaf->num_keys; i > insertion_point; i--) {
  leaf->keys[i] = leaf->keys[i - 1];
  leaf->pointers[i] = leaf->pointers[i - 1];
 }
 leaf->keys[insertion_point] = key;
 leaf->pointers[insertion_point] = pointer;
 leaf->num_keys++;
 return leaf;
}

node *insertIntoLeafAfterSplitting(node *root, node *leaf, int key, record *pointer) {
 node *new_leaf;
 int *temp_keys;
 void **temp_pointers;
 int insertion_index, split, new_key, i, j;

 new_leaf = makeLeaf();

 temp_keys = malloc(order * sizeof(int));
 if (temp_keys == NULL) {
  perror("Temporary keys array.");
  exit(EXIT_FAILURE);
 }
```

```
temp_pointers = malloc(order * sizeof(void *));
if (temp_pointers == NULL) {
  perror("Temporary pointers array.");
  exit(EXIT_FAILURE);
}

insertion_index = 0;
while (insertion_index < order - 1 && leaf->keys[insertion_index] < key)
  insertion_index++;

for (i = 0, j = 0; i < leaf->num_keys; i++, j++) {
  if (j == insertion_index)
    j++;
  temp_keys[j] = leaf->keys[i];
  temp_pointers[j] = leaf->pointers[i];
}

temp_keys[insertion_index] = key;
temp_pointers[insertion_index] = pointer;

leaf->num_keys = 0;

split = cut(order - 1);
```

```
for (i = 0; i < split; i++) {
  leaf->pointers[i] = temp_pointers[i];
  leaf->keys[i] = temp_keys[i];
  leaf->num_keys++;
}

for (i = split, j = 0; i < order; i++, j++) {
  new_leaf->pointers[j] = temp_pointers[i];
  new_leaf->keys[j] = temp_keys[i];
  new_leaf->num_keys++;
}

free(temp_pointers);
free(temp_keys);

new_leaf->pointers[order - 1] = leaf->pointers[order - 1];
leaf->pointers[order - 1] = new_leaf;

for (i = leaf->num_keys; i < order - 1; i++)
  leaf->pointers[i] = NULL;
for (i = new_leaf->num_keys; i < order - 1; i++)
  new_leaf->pointers[i] = NULL;

new_leaf->parent = leaf->parent;
new_key = new_leaf->keys[0];
```

```
  return insertIntoParent(root, leaf, new_key, new_leaf);
}


node *insertIntoNode(node *root, node *n,
        int left_index, int key, node *right) {
  int i;


  for (i = n->num_keys; i > left_index; i--) {
    n->pointers[i + 1] = n->pointers[i];
    n->keys[i] = n->keys[i - 1];
  }
  n->pointers[left_index + 1] = right;
  n->keys[left_index] = key;
  n->num_keys++;
  return root;
}


node *insertIntoNodeAfterSplitting(node *root, node *old_node, int left_index,
                int key, node *right) {
  int i, j, split, k_prime;
  node *new_node, *child;
  int *temp_keys;
  node **temp_pointers;
```

```
temp_pointers = malloc((order + 1) * sizeof(node *));
if (temp_pointers == NULL) {
  exit(EXIT_FAILURE);
}
temp_keys = malloc(order * sizeof(int));
if (temp_keys == NULL) {
  exit(EXIT_FAILURE);
}

for (i = 0, j = 0; i < old_node->num_keys + 1; i++, j++) {
  if (j == left_index + 1)
    j++;
  temp_pointers[j] = old_node->pointers[i];
}

for (i = 0, j = 0; i < old_node->num_keys; i++, j++) {
  if (j == left_index)
    j++;
  temp_keys[j] = old_node->keys[i];
}

temp_pointers[left_index + 1] = right;
temp_keys[left_index] = key;

split = cut(order);
```

```
  new_node = makeNode();
  old_node->num_keys = 0;
  for (i = 0; i < split - 1; i++) {
    old_node->pointers[i] = temp_pointers[i];
    old_node->keys[i] = temp_keys[i];
    old_node->num_keys++;
  }
  old_node->pointers[i] = temp_pointers[i];
  k_prime = temp_keys[split - 1];
  for (++i, j = 0; i < order; i++, j++) {
    new_node->pointers[j] = temp_pointers[i];
    new_node->keys[j] = temp_keys[i];
    new_node->num_keys++;
  }
  new_node->pointers[j] = temp_pointers[i];
  free(temp_pointers);
  free(temp_keys);
  new_node->parent = old_node->parent;
  for (i = 0; i <= new_node->num_keys; i++) {
    child = new_node->pointers[i];
    child->parent = new_node;
  }

  return insertIntoParent(root, old_node, k_prime, new_node);
}
```

```
node *insertIntoParent(node *root, node *left, int key, node *right) {
  int left_index;
  node *parent;

  parent = left->parent;

  if (parent == NULL)
    return insertIntoNewRoot(left, key, right);

  left_index = getLeftIndex(parent, left);

  if (parent->num_keys < order - 1)
    return insertIntoNode(root, parent, left_index, key, right);

  return insertIntoNodeAfterSplitting(root, parent, left_index, key, right);
}

node *insertIntoNewRoot(node *left, int key, node *right) {
  node *root = makeNode();
  root->keys[0] = key;
  root->pointers[0] = left;
  root->pointers[1] = right;
  root->num_keys++;
  root->parent = NULL;
```

```
  left->parent = root;
  right->parent = root;
  return root;
}

node *startNewTree(int key, record *pointer) {
  node *root = makeLeaf();
  root->keys[0] = key;
  root->pointers[0] = pointer;
  root->pointers[order - 1] = NULL;
  root->parent = NULL;
  root->num_keys++;
  return root;
}

node *insert(node *root, int key, int value) {
  record *record_pointer = NULL;
  node *leaf = NULL;

  record_pointer = find(root, key, false, NULL);
  if (record_pointer != NULL) {
    record_pointer->value = value;
    return root;
  }
```

```
  record_pointer = makeRecord(value);

  if (root == NULL)
    return startNewTree(key, record_pointer);

  leaf = findLeaf(root, key, false);

  if (leaf->num_keys < order - 1) {
    leaf = insertIntoLeaf(leaf, key, record_pointer);
    return root;
  }

  return insertIntoLeafAfterSplitting(root, leaf, key, record_pointer);
}

int main() {
  node *root;
  char instruction;

  root = NULL;

  root = insert(root, 5, 33);
  root = insert(root, 15, 21);
  root = insert(root, 25, 31);
  root = insert(root, 35, 41);
```

```
  root = insert(root, 45, 10);


  printTree(root);


  findAndPrint(root, 15, instruction = 'a');
}
```

Output:

25 |

15 | 35 |

5 | 15 | 25 | 35 45 |

[25] 0 ->

[15] 1 ->

Leaf [15] ->

Record at 0x1b6d330 -- key 15, value 21.

AVL trees:

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

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

int max(int a, int b);

// Calculate height
int height(struct Node *N) {
```

```
  if (N == NULL)
    return 0;
  return N->height;
}

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

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

// Right rotate
struct Node *rightRotate(struct Node *y) {
  struct Node *x = y->left;
  struct Node *T2 = x->right;

  x->right = y;
  y->left = T2;

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

  return x;
}

// Left rotate
struct Node *leftRotate(struct Node *x) {
  struct Node *y = x->right;
  struct Node *T2 = y->left;

  y->left = x;
  x->right = T2;

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

  return y;
}
```

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

// Insert node
struct Node *insertNode(struct Node *node, int key) {
  // Find the correct position to insertNode the node and insertNode it
  if (node == NULL)
    return (newNode(key));

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

  // Update the balance factor of each node and
  // Balance the tree
  node->height = 1 + max(height(node->left),
               height(node->right));

  int balance = getBalance(node);
  if (balance > 1 && key < node->left->key)
    return rightRotate(node);

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

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

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

  return node;
}

struct Node *minValueNode(struct Node *node) {
  struct Node *current = node;
```

```
  while (current->left != NULL)
    current = current->left;

  return current;
}

// Delete a nodes
struct Node *deleteNode(struct Node *root, int key) {
  // Find the node and delete it
  if (root == NULL)
    return root;

  if (key < root->key)
    root->left = deleteNode(root->left, key);

  else if (key > root->key)
    root->right = deleteNode(root->right, key);

  else {
    if ((root->left == NULL) || (root->right == NULL)) {
      struct Node *temp = root->left ? root->left : root->right;

      if (temp == NULL) {
        temp = root;
        root = NULL;
      } else
        *root = *temp;
      free(temp);
    } else {
      struct Node *temp = minValueNode(root->right);

      root->key = temp->key;

      root->right = deleteNode(root->right, temp->key);
    }
  }

  if (root == NULL)
    return root;

  // Update the balance factor of each node and
  // balance the tree
  root->height = 1 + max(height(root->left),
                         height(root->right));

  int balance = getBalance(root);
  if (balance > 1 && getBalance(root->left) >= 0)
    return rightRotate(root);
```

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

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

  if (balance < -1 && getBalance(root->right) > 0) {
    root->right = rightRotate(root->right);
    return leftRotate(root);
  }

  return root;
}

// Print the tree
void printPreOrder(struct Node *root) {
  if (root != NULL) {
    printf("%d ", root->key);
    printPreOrder(root->left);
    printPreOrder(root->right);
  }
}

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

  root = insertNode(root, 2);
  root = insertNode(root, 1);
  root = insertNode(root, 7);
  root = insertNode(root, 4);
  root = insertNode(root, 5);
  root = insertNode(root, 3);
  root = insertNode(root, 8);

  printPreOrder(root);

  root = deleteNode(root, 3);

  printf("\nAfter deletion: ");
  printPreOrder(root);

  return 0;
}
```

Output:

4 2 1 3 7 5 8

After deletion: 4 2 1 7 5 8

Red - Black trees:

```
#include <stdio.h>

#include <stdlib.h>

enum nodeColor {

  RED,

  BLACK

};


struct rbNode {

  int data, color;

  struct rbNode *link[2];

};


struct rbNode *root = NULL;


// Create a red-black tree

struct rbNode *createNode(int data) {

  struct rbNode *newnode;

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

  newnode->data = data;

  newnode->color = RED;

  newnode->link[0] = newnode->link[1] = NULL;
```

```
  return newnode;
}

// Insert an node
void insertion(int data) {
  struct rbNode *stack[98], *ptr, *newnode, *xPtr, *yPtr;
  int dir[98], ht = 0, index;
  ptr = root;
  if (!root) {
    root = createNode(data);
    return;
  }

  stack[ht] = root;
  dir[ht++] = 0;
  while (ptr != NULL) {
    if (ptr->data == data) {
      printf("Duplicates Not Allowed!!\n");
      return;
    }
    index = (data - ptr->data) > 0 ? 1 : 0;
    stack[ht] = ptr;
    ptr = ptr->link[index];
    dir[ht++] = index;
  }
```

```
stack[ht - 1]->link[index] = newnode = createNode(data);
while ((ht >= 3) && (stack[ht - 1]->color == RED)) {
  if (dir[ht - 2] == 0) {
    yPtr = stack[ht - 2]->link[1];
    if (yPtr != NULL && yPtr->color == RED) {
      stack[ht - 2]->color = RED;
      stack[ht - 1]->color = yPtr->color = BLACK;
      ht = ht - 2;
    } else {
      if (dir[ht - 1] == 0) {
        yPtr = stack[ht - 1];
      } else {
        xPtr = stack[ht - 1];
        yPtr = xPtr->link[1];
        xPtr->link[1] = yPtr->link[0];
        yPtr->link[0] = xPtr;
        stack[ht - 2]->link[0] = yPtr;
      }
      xPtr = stack[ht - 2];
      xPtr->color = RED;
      yPtr->color = BLACK;
      xPtr->link[0] = yPtr->link[1];
      yPtr->link[1] = xPtr;
      if (xPtr == root) {
        root = yPtr;
```

```
    } else {
      stack[ht - 3]->link[dir[ht - 3]] = yPtr;
    }
    break;
  }
} else {
  yPtr = stack[ht - 2]->link[0];
  if ((yPtr != NULL) && (yPtr->color == RED)) {
    stack[ht - 2]->color = RED;
    stack[ht - 1]->color = yPtr->color = BLACK;
    ht = ht - 2;
  } else {
    if (dir[ht - 1] == 1) {
      yPtr = stack[ht - 1];
    } else {
      xPtr = stack[ht - 1];
      yPtr = xPtr->link[0];
      xPtr->link[0] = yPtr->link[1];
      yPtr->link[1] = xPtr;
      stack[ht - 2]->link[1] = yPtr;
    }
    xPtr = stack[ht - 2];
    yPtr->color = BLACK;
    xPtr->color = RED;
    xPtr->link[1] = yPtr->link[0];
```

```
      yPtr->link[0] = xPtr;
      if (xPtr == root) {
        root = yPtr;
      } else {
        stack[ht - 3]->link[dir[ht - 3]] = yPtr;
      }
      break;
    }
   }
  }
  root->color = BLACK;
}

// Delete a node
void deletion(int data) {
  struct rbNode *stack[98], *ptr, *xPtr, *yPtr;
  struct rbNode *pPtr, *qPtr, *rPtr;
  int dir[98], ht = 0, diff, i;
  enum nodeColor color;

  if (!root) {
    printf("Tree not available\n");
    return;
  }
```

```
ptr = root;
while (ptr != NULL) {
  if ((data - ptr->data) == 0)
    break;
  diff = (data - ptr->data) > 0 ? 1 : 0;
  stack[ht] = ptr;
  dir[ht++] = diff;
  ptr = ptr->link[diff];
}

if (ptr->link[1] == NULL) {
  if ((ptr == root) && (ptr->link[0] == NULL)) {
    free(ptr);
    root = NULL;
  } else if (ptr == root) {
    root = ptr->link[0];
    free(ptr);
  } else {
    stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];
  }
} else {
  xPtr = ptr->link[1];
  if (xPtr->link[0] == NULL) {
    xPtr->link[0] = ptr->link[0];
    color = xPtr->color;
```

```
  xPtr->color = ptr->color;
  ptr->color = color;

  if (ptr == root) {
    root = xPtr;
  } else {
    stack[ht - 1]->link[dir[ht - 1]] = xPtr;
  }

  dir[ht] = 1;
  stack[ht++] = xPtr;
 } else {
  i = ht++;
  while (1) {
    dir[ht] = 0;
    stack[ht++] = xPtr;
    yPtr = xPtr->link[0];
    if (!yPtr->link[0])
      break;
    xPtr = yPtr;
  }

  dir[i] = 1;
  stack[i] = yPtr;
  if (i > 0)
```

```
      stack[i - 1]->link[dir[i - 1]] = yPtr;

    yPtr->link[0] = ptr->link[0];

    xPtr->link[0] = yPtr->link[1];
    yPtr->link[1] = ptr->link[1];

    if (ptr == root) {
      root = yPtr;
    }

    color = yPtr->color;
    yPtr->color = ptr->color;
    ptr->color = color;
  }
}

if (ht < 1)
  return;

if (ptr->color == BLACK) {
  while (1) {
    pPtr = stack[ht - 1]->link[dir[ht - 1]];
    if (pPtr && pPtr->color == RED) {
      pPtr->color = BLACK;
```

```
  break;
}

if (ht < 2)
  break;

if (dir[ht - 2] == 0) {
  rPtr = stack[ht - 1]->link[1];

  if (!rPtr)
    break;

  if (rPtr->color == RED) {
    stack[ht - 1]->color = RED;
    rPtr->color = BLACK;
    stack[ht - 1]->link[1] = rPtr->link[0];
    rPtr->link[0] = stack[ht - 1];

    if (stack[ht - 1] == root) {
      root = rPtr;
    } else {
      stack[ht - 2]->link[dir[ht - 2]] = rPtr;
    }
    dir[ht] = 0;
    stack[ht] = stack[ht - 1];
```

```
  stack[ht - 1] = rPtr;
  ht++;

  rPtr = stack[ht - 1]->link[1];
 }

if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
  (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
  rPtr->color = RED;
 } else {
  if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) {
    qPtr = rPtr->link[0];
    rPtr->color = RED;
    qPtr->color = BLACK;
    rPtr->link[0] = qPtr->link[1];
    qPtr->link[1] = rPtr;
    rPtr = stack[ht - 1]->link[1] = qPtr;
  }
  rPtr->color = stack[ht - 1]->color;
  stack[ht - 1]->color = BLACK;
  rPtr->link[1]->color = BLACK;
  stack[ht - 1]->link[1] = rPtr->link[0];
  rPtr->link[0] = stack[ht - 1];
  if (stack[ht - 1] == root) {
    root = rPtr;
```

```
    } else {
      stack[ht - 2]->link[dir[ht - 2]] = rPtr;
    }
    break;
  }
} else {
  rPtr = stack[ht - 1]->link[0];
  if (!rPtr)
    break;

  if (rPtr->color == RED) {
    stack[ht - 1]->color = RED;
    rPtr->color = BLACK;
    stack[ht - 1]->link[0] = rPtr->link[1];
    rPtr->link[1] = stack[ht - 1];

    if (stack[ht - 1] == root) {
      root = rPtr;
    } else {
      stack[ht - 2]->link[dir[ht - 2]] = rPtr;
    }
    dir[ht] = 1;
    stack[ht] = stack[ht - 1];
    stack[ht - 1] = rPtr;
    ht++;
```

```
  rPtr = stack[ht - 1]->link[0];
 }
 if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
  (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
  rPtr->color = RED;
 } else {
  if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) {
   qPtr = rPtr->link[1];
   rPtr->color = RED;
   qPtr->color = BLACK;
   rPtr->link[1] = qPtr->link[0];
   qPtr->link[0] = rPtr;
   rPtr = stack[ht - 1]->link[0] = qPtr;
  }
  rPtr->color = stack[ht - 1]->color;
  stack[ht - 1]->color = BLACK;
  rPtr->link[0]->color = BLACK;
  stack[ht - 1]->link[0] = rPtr->link[1];
  rPtr->link[1] = stack[ht - 1];
  if (stack[ht - 1] == root) {
   root = rPtr;
  } else {
   stack[ht - 2]->link[dir[ht - 2]] = rPtr;
  }
```

```
        break;
      }
     }
     ht--;
   }
  }
}

// Print the inorder traversal of the tree
void inorderTraversal(struct rbNode *node) {
  if (node) {
    inorderTraversal(node->link[0]);
    printf("%d  ", node->data);
    inorderTraversal(node->link[1]);
  }
  return;
}

// Driver code
int main() {
  int ch, data;
  while (1) {
    printf("1. Insertion\t2. Deletion\n");
    printf("3. Traverse\t4. Exit");
    printf("\nEnter your choice:");
```

```
  scanf("%d", &ch);
  switch (ch) {
    case 1:
      printf("Enter the element to insert:");
      scanf("%d", &data);
      insertion(data);
      break;
    case 2:
      printf("Enter the element to delete:");
      scanf("%d", &data);
      deletion(data);
      break;
    case 3:
      inorderTraversal(root);
      printf("\n");
      break;
    case 4:
      exit(0);
    default:
      printf("Not available\n");
      break;
  }
  printf("\n");
}
return 0;
```

}

Output:

1. Insertion 2. Deletion

3. Traverse 4. Exit

Enter your choice:1

Enter the element to insert:4

1. Insertion 2. Deletion

3. Traverse 4. Exit

Enter your choice:1

Enter the element to insert:5

1. Insertion 2. Deletion

3. Traverse 4. Exit

Enter your choice:1

Enter the element to insert:6

1. Insertion 2. Deletion

3. Traverse 4. Exit

Enter your choice:3

4 5 6

9.Write a program to implement the graph traversal methods.

BFS:

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

// A structure to represent a node in the adjacency list
struct node {
   int vertex;
   struct node* next;
};

// A structure to represent the adjacency list
struct adj_list {
   struct node* head;
};
```

```
// A structure to represent the graph
struct graph {
   int num_vertices;
   struct adj_list* adj_lists;
   int* visited;
};

// Create a new node in the adjacency list
struct node* new_node(int vertex) {
   struct node* new_node = (struct node*)malloc(sizeof(struct node));
new_node->vertex = vertex;
new_node->next = NULL;
   return new_node;
}

// Create a graph with n vertices
struct graph* create_graph(int n) {
   struct graph* graph = (struct graph*)malloc(sizeof(struct graph));
   graph->num_vertices = n;
   graph->adj_lists = (struct adj_list*)malloc(n * sizeof(struct adj_list));
   graph->visited = (int*)malloc(n * sizeof(int));

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

   return graph;
}

// Add an edge to the graph
void add_edge(struct graph* graph, int src, int dest) {
   // Add an edge from src to dest
   struct node* new_node1 = new_node(dest);
   new_node1->next = graph->adj_lists[src].head;
   graph->adj_lists[src].head = new_node1;

   // Add an edge from dest to src
   struct node* new_node2 = new_node(src);
   new_node2->next = graph->adj_lists[dest].head;
   graph->adj_lists[dest].head = new_node2;
}

// BFS traversal of the graph starting from vertex v
void bfs(struct graph* graph, int v) {
   // Create a queue for BFS
```

```
   int queue[1000];
   int front = -1;
   int rear = -1;

   // Mark the current node as visited and enqueue it
   graph->visited[v] = 1;
   queue[++rear] = v;

   // Loop to visit all the vertices in the graph
   while (front != rear) {
      // Dequeue a vertex from the queue and print it
      int current_vertex = queue[++front];
printf("%d ", current_vertex);

      // Get all the neighbors of the dequeued vertex
      struct node* temp = graph->adj_lists[current_vertex].head;
      while (temp != NULL) {
         int adj_vertex = temp->vertex;

         // If the neighbor has not been visited, mark it as visited and enqueue it
         if (graph->visited[adj_vertex] == 0) {
            graph->visited[adj_vertex] = 1;
            queue[++rear] = adj_vertex;
         }

         temp = temp->next;
      }
   }
}

int main() {
   // Create a graph with 6 vertices
   struct graph* graph = create_graph(6);

   // Add edges to the graph
add_edge(graph, 0, 1);
add_edge(graph, 0, 2);
add_edge(graph, 1, 3);
add_edge(graph, 1, 4);
add_edge(graph, 2, 4);
add_edge(graph, 3, 4);
add_edge(graph, 3, 5);
add_edge(graph, 4,5);
      // Perform BFS traversal starting from vertex 0
printf("BFS traversal starting from vertex 0: ");
bfs(graph, 0);

   return 0;
```

```
}
```

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

DFS:

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

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

// Graph structure to store number
// of vertices and edges and
// Adjacency matrix
struct Graph {
   int V;
   int E;
   int** Adj;
};

// Function to input data of graph
struct Graph* adjMatrix()
{
   struct Graph* G = (struct Graph*)
      malloc(sizeof(struct Graph));
   if (!G) {
      printf("Memory Error\n");
      return NULL;
   }
   G->V = 7;
   G->E = 7;

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

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

```
   G->Adj[1][4] = G->Adj[4][1] = 1;
   G->Adj[1][5] = G->Adj[5][1] = 1;
   G->Adj[1][6] = G->Adj[6][1] = 1;
   G->Adj[6][2] = G->Adj[2][6] = 1;

   return G;
}

// DFS function to print DFS traversal of graph
void DFS(struct Graph* G, int u)
{
   vis[u] = 1;
   printf("%d ", u);
   for (int v = 0; v < G->V; v++) {
      if (!vis[v] && G->Adj[u][v]) {
         DFS(G, v);
      }
   }
}

// Function for DFS traversal
void DFStraversal(struct Graph* G)
{
   for (int i = 0; i < 100; i++) {
      vis[i] = 0;
   }
   for (int i = 0; i < G->V; i++) {
      if (!vis[i]) {
         DFS(G, i);
      }
   }
}

// Driver code
void main()
{
   struct Graph* G;
   G = adjMatrix();
   DFStraversal(G);
}
```

Output:
0 1 3 4 5 6 2

10.Implement a Pattern matching algorithms using Boyer- Moore, Knuth-Morris-Pratt

Boyer- Moore:

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

#define ALPHABET_SIZE 256

// Function to compute the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to preprocess the pattern and generate the bad character heuristic table
void generateBadCharHeuristic(char *pattern, int patternLength, int badCharHeuristic[ALPHABET_SIZE])
{
    int i;

    for (i = 0; i < ALPHABET_SIZE; i++) {
        badCharHeuristic[i] = -1;  // Initialize all entries to -1
    }

    for (i = 0; i < patternLength; i++) {
        badCharHeuristic[(int)pattern[i]] = i;
    }
}

// Function to perform Boyer-Moore search
void searchBoyerMoore(char *text, char *pattern) {
```

```
    int textLength = strlen(text);
    int patternLength = strlen(pattern);

    int badCharHeuristic[ALPHABET_SIZE];
    generateBadCharHeuristic(pattern, patternLength, badCharHeuristic);

    int shift = 0;  // Initialize the shift to 0

    while (shift <= textLength - patternLength) {
        int j = patternLength - 1;

        while (j >= 0 && pattern[j] == text[shift + j]) {
            j--;
        }

        if (j < 0) {
            // Pattern found at index 'shift'
            printf("Pattern found at index %d\n", shift);

            // Move the window
            shift += (shift + patternLength < textLength) ? patternLength - badCharHeuristic[(int)text[shift + patternLength]] : 1;
        } else {
            // Shift the pattern according to bad character heuristic
            shift += max(1, j - badCharHeuristic[(int)text[shift + j]]);
        }
    }
}
```

```
int main() {
    char text[] = "AABAACAADAABAABA";
    char pattern[] = "AABA";

    printf("Text: %s\n", text);
    printf("Pattern: %s\n", pattern);

    searchBoyerMoore(text, pattern);

    return 0;
}
```

Output:

Text: AABAACAADAABAABA

Pattern: AABA

Pattern found at index 0

Pattern found at index 9

Pattern found at index 12

Knuth-Morris-Pratt:

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

// Function to compute the LPS (Longest Prefix Suffix) array
void computeLPSArray(char *pattern, int patternLength, int *lps) {
    int len = 0;
```

```
    int i = 1;

    lps[0] = 0;

    while (i < patternLength) {
        if (pattern[i] == pattern[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

// Function to perform KMP search
void searchKMP(char *text, char *pattern) {
    int textLength = strlen(text);
    int patternLength = strlen(pattern);

    int *lps = (int *)malloc(sizeof(int) * patternLength);

    computeLPSArray(pattern, patternLength, lps);
```

```
    int i = 0; // index for text[]
    int j = 0; // index for pattern[]

    while (i < textLength) {
        if (pattern[j] == text[i]) {
            j++;
            i++;
        }

        if (j == patternLength) {
            // Pattern found at index 'i - j'
            printf("Pattern found at index %d\n", i - j);
            j = lps[j - 1];
        } else if (i < textLength && pattern[j] != text[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }

    free(lps);
}

int main() {
    char text[] = "AABAACAADAABAABA";
```

```
    char pattern[] = "AABA";

    printf("Text: %s\n", text);
    printf("Pattern: %s\n", pattern);

    searchKMP(text, pattern);

    return 0;
}
```

Output:

Text: AABAACAADAABAABA

Pattern: AABA

Pattern found at index 0

Pattern found at index 9

Pattern found at index 12