L	T	P	C
0	0	3	1.5

III Semester Syllabus

CS351PC: DATA STRUCTURES LAB

[Common to CSE, IT, CSB & CSD]

Prerequisites:

1. A Course on "Programming for problem solving".

Course Objectives

To Learn

- It covers various concepts of C programming language
- It introduces searching and sorting algorithms
- It provides an understanding of data structures such as stacks and queues.

Course Outcomes

Student will be able to:

- Ability to develop C programs for computing and real-life applications using basic elements like control statements, arrays, functions, pointers and strings, and data structures like stacks, queues and linked lists.
- Ability to Implement searching and sorting algorithms.

List of Experiments:

1. AIM:

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

CODE:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *head;
void create();
void insert_begin();
void insert_after();
void insert_end();
void delete_begin();
```

```
void delete_after();
void delete_end();
void display();
void main()
{
int ch;
system("clear");
while(1)
{
printf("\n_______________");
printf("\n single liked list ADT operations are:\n");
printf("_______________");
printf("\n\t1.create");
printf("\n\t2.Insertion at the beginning");
printf("\n\t3.Insertion after the given info:");
printf("\n\t4.Insertion at the end");
printf("\n\t5.deletion at the beginning");
printf("\n\t6.Deletion the given info:");
printf("\n\t7.Deletion at the end");
printf("\n\t8.Display");
printf("\n\t9.Exit");
printf("\n Enter ur choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:create();
 break;
case 2:insert_begin();
break;
case 3:
insert_after();
break;
case 4:
```

```
insert_end();
break;
case 5:
delete_begin();
break;
case 6:
delete_after();
break;
case 7:
delete_end();
break;
case 8:
display();
break;
case 9:
exit(0);
break;
default:
printf("\n wrong choice\n");
}
}
}
void create()
{
struct node *ptr,*cptr;
int c;
ptr=(struct node*)malloc(sizeof(struct node));
printf("\n Enter first node information:");
scanf("%d",&ptr->data);
head=ptr;
printf("\n Eneter 0/1 for more nodes:");
scanf("%d",&c);
```

```
while(c==1)
{
cptr=(struct node*)malloc(sizeof(struct node));
ptr->next=cptr;
ptr=cptr;
printf("\n Enter next node information:");
scanf("%d",&cptr->data);
printf("\n enter 0/1 for more nodes:");
scanf("%d",&c);
}
ptr->next=NULL;
}
void insert_begin()
{
struct node *ptr;
ptr=(struct node*)malloc(sizeof(struct node));
printf("\n Enter node information to be inserted:");
scanf("%d",&ptr->data);
ptr->next=head;
head=ptr;
}
void insert_end()
{
struct node *ptr,*cptr;
ptr=(struct node*)malloc(sizeof(struct node));
printf("\n Enter node information to be inserted:");
scanf("%d",&ptr->data);
cptr=head;
while(cptr->next!=NULL)
cptr=cptr->next;
cptr->next=ptr;
ptr->next=NULL;
}
```

```
void insert_after()
{
struct node *ptr,*cptr;
int d;
ptr=(struct node*)malloc(sizeof(struct node));
scanf("%d",&ptr->data);
printf("\n enter node info after which you want to inserted:");
scanf("%d",&d);
cptr=head;
while(cptr->data!=d)
cptr=cptr->next;
ptr->next=cptr->next;
cptr->next=ptr;
}

void delete_begin()
{
struct node *ptr;
if(head==NULL)
printf("\n linked list underflow\n");
else
{
ptr=head;
printf("\n deleted element is:%d",ptr->data);
head=ptr->next;
free(ptr);
}
}
void delete_end()
{
struct node *ptr,*cptr;
ptr=head;
while(ptr->next!=NULL)
```

```
{
cptr=ptr;
ptr=ptr->next;
}
cptr->next=NULL;
printf("\n deleted elements is:%d",ptr->data);
free(ptr);
}
void  delete_after()
{
struct node *ptr,*cptr;
int d;
if(head==NULL)
printf("\n Linked list  underflow\n");
else
{
ptr=head;
printf("\n Enter node info to be deleted:");
scanf("%d",&d);
while(ptr->data!=d)
{
cptr=ptr;
ptr=ptr->next;
}
cptr->next=ptr->next;
printf("\n deleted element is:%d",ptr->data);
free(ptr);
}
}
void display()
{
struct node *ptr;
ptr=head;
```

```
if(head==NULL)
printf("\n Linked list is empty\n");
else
{
while(ptr!=NULL)
{
printf("%d->",ptr->data);
ptr=ptr->next;
}
}
}
```

OUTPUT :

```
________________
 single liked list ADT operations are:
_______________
        1.create
        2.Insertion at the beginning
        3.Insertion after the given info:
        4.Insertion at the end
        5.deletion at the beginning
        6.Deletion the given info:
        7.Deletion at the end
        8.Display
        9.Exit
 Enter ur choice:1
 Enter first node information:10
 Eneter 0/1 for more nodes:1
 Enter next node information:20
 enter 0/1 for more nodes:0
________________
 single liked list ADT operations are:
_______________
        1.create
```

2.Insertion at the beginning

3.Insertion after the given info:

4.Insertion at the end

5.deletion at the beginning

6.Deletion the given info:

7.Deletion at the end

8.Display

9.Exit

Enter ur choice:2

Enter node information to be inserted:30

---

single liked list ADT operations are:

1.create

2.Insertion at the beginning

3.Insertion after the given info:

4.Insertion at the end

5.deletion at the beginning

6.Deletion the given info:

7.Deletion at the end

8.Display

9.Exit

Enter ur choice:8

30->10->20->

---

single liked list ADT operations are:

1.create

2.Insertion at the beginning

3.Insertion after the given info:

4.Insertion at the end

5.deletion at the beginning

6.Deletion the given info:

7.Deletion at the end

8.Display

9.Exit

Enter ur choice:5

deleted element is:30

---

single liked list ADT operations are:

1.create

2.Insertion at the beginning

3.Insertion after the given info:

4.Insertion at the end

5.deletion at the beginning

6.Deletion the given info:

7.Deletion at the end

8.Display

9.Exit

Enter ur choice:8

10->20->

---

single liked list ADT operations are:

1.create

2.Insertion at the beginning

3.Insertion after the given info:

4.Insertion at the end

5.deletion at the beginning

6.Deletion the given info:

7.Deletion at the end

8.Display

9.Exit

Enter ur choice:9

2 .AIM:

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

CODE:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
struct node *prev;
int data;
struct node *next;
};
struct node *head;
void create();
void insert_begin();
void insert_after();
void insert_end();
void delete_begin();
void delete_info();
void delete_end();
void display();
void main()
{
int ch;
system("clear");
while(1)
{
printf("_________________");
printf("\n doubly liked list ADT operations are:\n");
printf("________________");
printf("\n\t1.create");
printf("\n\t2.Insertion at the bieginning");
printf("\n\t3.Insertion after the given info:");
printf("\n\t4.Insertion at the end");
```

```
printf("\n\t5.deletion at the beginning");
printf("\nt6.Deletion the given info:");
printf("\nt7.Deletion at the end");
printf("\n\t8.Display");
printf("\n\t9.Exit");
printf("\n Enter ur choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:create();
 break;
case 2:insert_begin();
break;
case 3:
insert_after();
break;
case 4:

insert_end();
break;
case 5:
delete_begin();
break;
case 6:
delete_info();
break;
case 7:
delete_end();
break;
case 8:
display();
break;
case 9:
```

```
exit(0);
break;
default:
printf("\n wrong choice\n");
}
}
}
void create()
{
struct node *ptr,*cptr;
int c;
ptr=(struct node*)malloc(sizeof(struct node));
printf("\n Enter first node information:");
scanf("%d",&ptr->data);
head=ptr;
ptr->prev=NULL;
printf("\n Eneter 0/1 for more nodes:");
scanf("%d",&c);
while(c==1)
{
cptr=(struct node*)malloc(sizeof(struct node));
ptr->next=cptr;
cptr->prev=ptr;
ptr=cptr;
printf("\n Enter next node information:");
scanf("%d",&cptr->data);
printf("\n eneter 0/1 for more nodes:");
scanf("%d",&c);
}
ptr->next=NULL;
}
void insert_begin()
{
```

```
struct node *ptr;
ptr=(struct node*)malloc(sizeof(struct node));
printf("\n Eneter node information to be inserted:");
scanf("%d",&ptr->data);
ptr->next=head;
ptr->prev=NULL;
head->prev=ptr;
head=ptr;
}
void insert_end()
{
struct node *ptr,*cptr;
ptr=(struct node*)malloc(sizeof(struct node));
printf("\n Eneter node information to be inserted:");
scanf("%d",&ptr->data);
cptr=head;
while(cptr->next!=NULL)
cptr=cptr->next;
cptr->next=ptr;
ptr->prev=cptr;
ptr->next=NULL;
}
void insert_after()
{
struct node *ptr,*cptr;
int d;
ptr=(struct node*)malloc(sizeof(struct node));
printf("enter node information to insert\n");
scanf("%d",&ptr->data);
printf("\n eneter node info after which you want to insert:");
scanf("%d",&d);
cptr=head;
while(cptr->data!=d)
```

```
cptr=cptr->next;
ptr->next=cptr->next;
(cptr->next)->prev=ptr;
cptr->next=ptr;
ptr->prev=cptr;
}

void delete_begin()
{
struct node *ptr;
if(head==NULL)
printf("\n Doubly linked list underflow\n");
else if(head->prev==NULL && head->next==NULL)
{
ptr = head;
printf("\ndeleted element is %d ",ptr->data);
head=NULL;
free(ptr);
}
else
{
ptr=head;
printf("\n deleted element is:%d",ptr->data);
head=ptr->next;
head->prev=NULL;
free(ptr);
}
}
void delete_end()
{
struct node *ptr,*cptr;
if(head==NULL) printf("\nDLL underflow");
else if(head->prev=NULL && head->next==NULL)
```

```
{
ptr=head;
printf("\ndeleted element is %d",ptr->data);
head=NULL;
free(ptr);
}
else
{
ptr=head;
while(ptr->next!=NULL)
{
cptr=ptr;
ptr=ptr->next;
}
cptr->next=NULL;
printf("\n deleted elements is:%d",ptr->data);
free(ptr);
}
}
void  delete_info()
{
struct node *ptr,*cptr;
int d;
if(head==NULL)
printf("\nDoubly Linked list  underflow\n");
else
{
ptr=head;
printf("\n Enter node info to be deleted:");
scanf("%d",&d);
while(ptr->data!=d)
{
cptr=ptr;
```

```
ptr=ptr->next;
}
cptr->next=ptr->next;
(ptr->next)->prev=cptr;
printf("\n deleted element is:%d",ptr->data);
free(ptr);
}
}
void display()
{
struct node *ptr,*cptr,*revptr;
ptr=head;
if(head==NULL)
printf("\n Doubly Linked list is empty\n");
else
{
while(ptr!=NULL)
{
printf("%d->",ptr->data);
cptr=ptr;
ptr=ptr->next;
}
while(cptr!=NULL)
{
printf("%d<-->",cptr->data);
cptr=cptr->prev;
}
}
}
```

OUTPUT:

```
________________

 doubly liked list ADT operations are:

_______________

        1.create
        2.Insertion at the bieginning
        3.Insertion after the given info:
        4.Insertion at the end
        5.deletion at the beginning
        6.Deletion the given info:
        7.Deletion at the end
        8.Display
        9.Exit
 Enter ur choice:1
 Enter first node information:10
 Eneter 0/1 for more nodes:1
 Enter next node information:20
 eneter 0/1 for more nodes:0

________________

 doubly liked list ADT operations are:

_______________

        1.create
        2.Insertion at the bieginning
        3.Insertion after the given info:
        4.Insertion at the end
        5.deletion at the beginning
        6.Deletion the given info:
        7.Deletion at the end
        8.Display
        9.Exit
 Enter ur choice:8
10->20->20<-->10<-->
```

doubly liked list ADT operations are:

1.create

2.Insertion at the bieginning

3.Insertion after the given info:

4.Insertion at the end

5.deletion at the beginning

6.Deletion the given info:

7.Deletion at the end

8.Display

9.Exit

Enter ur choice:3

enter node information to insert

30

eneter node info after which you want to insert:10

doubly liked list ADT operations are:

1.create

2.Insertion at the bieginning

3.Insertion after the given info:

4.Insertion at the end

5.deletion at the beginning

6.Deletion the given info:

7.Deletion at the end

8.Display

9.Exit

Enter ur choice:8

```
10->30->20->20<-->30<-->10<-->
```

```
 ________________
 doubly liked list ADT operations are:
 _______________
        1.create
        2.Insertion at the bieginning
        3.Insertion after the given info:
        4.Insertion at the end
        5.deletion at the beginning
        6.Deletion the given info:
        7.Deletion at the end
        8.Display
        9.Exit
 Enter ur choice:7

 deleted elements is:20
 ________________
 doubly liked list ADT operations are:
 _______________
        1.create
        2.Insertion at the bieginning
        3.Insertion after the given info:
        4.Insertion at the end
        5.deletion at the beginning
        6.Deletion the given info:
        7.Deletion at the end
        8.Display
        9.Exit
 Enter ur choice:8
10->30->30<-->10<-->
```

---

doubly liked list ADT operations are:

1.create

2.Insertion at the bieginning

3.Insertion after the given info:

4.Insertion at the end

5.deletion at the beginning

6.Deletion the given info:

7.Deletion at the end

8.Display

9.Exit

Enter ur choice:9

3.AIM:

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

CODE:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *head;
void create();
void insert_begin();
void insert_after();
void insert_end();
void delete_begin();
void delete_info();
void delete_end();
void display();
```

```
void main()
{
int ch;
system("clear");
while(1)
{
printf("________________");
printf("\n circular linked list ADT operations are:\n");
printf("_______________");
printf("\n\t1.create");
printf("\n\t2.Insertion at the beginning");
printf("\n\t3.Insertion after the given info:");
printf("\n\t4.Insertion at the end");
printf("\n\t5.deletion at the beginning");
printf("\n\t6.Deletion the given info:");
printf("\n\t7.Deletion at the end");
printf("\n\t8.Display");
printf("\n\t9.Exit");
printf("\n Enter ur choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:create();
 break;
case 2:insert_begin();
break;
case 3:
insert_after();
break;
case 4:

insert_end();
break;
```

```
case 5:
delete_begin();
break;
case 6:
delete_info();
break;
case 7:
delete_end();
break;
case 8:
display();
break;
case 9:
exit(0);
break;
default:
printf("\n wrong choice\n");
}
}
}
void create()
{
struct node *ptr,*cptr;
int c;
ptr=(struct node*)malloc(sizeof(struct node));
printf("\n Enter first node information:");
scanf("%d",&ptr->data);
head=ptr;
printf("\n Eneter 0/1 for more nodes:");
scanf("%d",&c);
while(c==1)
{
cptr=(struct node*)malloc(sizeof(struct node));
```

```
ptr->next=cptr;
ptr=cptr;
printf("\n Enter next node information:");
scanf("%d",&cptr->data);
printf("\n eneter 0/1 for more nodes:");
scanf("%d",&c);
}
ptr->next=head;
}
void insert_begin()
{
struct node *ptr,*cptr;
ptr=(struct node*)malloc(sizeof(struct node));
printf("\n Enter node information to be inserted:");
scanf("%d",&ptr->data);
cptr=head;
while(cptr->next!=head)
cptr=cptr->next;
ptr->next=head;
head=ptr;
cptr->next=head;
}
void insert_end()
{
struct node *ptr,*cptr;
ptr=(struct node*)malloc(sizeof(struct node));
printf("\n Eneter node information to be inserted:");
scanf("%d",&ptr->data);
cptr=head;
while(cptr->next!=head)
cptr=cptr->next;
ptr->next=head;
cptr->next=ptr;
```

```
}
void insert_after()
{
struct node *ptr,*cptr;
int d;
ptr=(struct node*)malloc(sizeof(struct node));
printf("\n Eneter node information to be inserted:");
scanf("%d",&ptr->data);
printf("\n enter node info after which you want to inserted:");
scanf("%d",&d);
cptr=head;
while(cptr->data!=d)
cptr=cptr->next;
ptr->next=cptr->next;
cptr->next=ptr;
}

void delete_begin()
{
struct node *ptr,*cptr;
if(head==NULL)
printf("\n Circular Linked list underflow\n");
else
{
ptr=head;
cptr=head;
printf("\n deleted element is:%d",ptr->data);
while(cptr->next!=head)
cptr=cptr->next;
head=ptr->next;
free(ptr);
cptr->next=head;
}
```

```
}
void delete_end()
{
struct node *ptr,*cptr;
if(head==NULL)
printf("Circular Linked list empty\n");
else
{
ptr=head;
while(ptr->next!=NULL)
{
cptr=ptr;
ptr=ptr->next;
}
cptr->next=head;
printf("\n deleted elements is:%d",ptr->data);
free(ptr);
}
}
void  delete_info()
{
struct node *ptr,*cptr;
int d;
if(head==NULL)
printf("\n Circular Linked list  underflow\n");
else
{
ptr=head;
printf("\n Eneter node info to be deleted:");
scanf("%d",&d);
while(ptr->data!=d)
{
cptr=ptr;
```

```
ptr=ptr->next;
}
cptr->next=ptr->next;
printf("\n deleted element is:%d",ptr->data);
free(ptr);
}
}
void display()
{
struct node *ptr;
if(head==NULL)
printf("\n Circular Linked list is empty\n");
else
{
ptr=head;
do
{
printf("%d->",ptr->data);
ptr=ptr->next;
}while(ptr!=head);
}
}
```

OUTPUT:

```
_________________
 circular linked list ADT operations are:
________________
        1.create
        2.Insertion at the beginning
        3.Insertion after the given info:
        4.Insertion at the end
        5.deletion at the beginning
        6.Deletion the given info:
        7.Deletion at the end
```

8.Display

9.Exit

Enter ur choice:1

Enter first node information:10

Eneter 0/1 for more nodes:1

Enter next node information:20

eneter 0/1 for more nodes:0

---

circular linked list ADT operations are:

1.create

2.Insertion at the beginning

3.Insertion after the given info:

4.Insertion at the end

5.deletion at the beginning

6.Deletion the given info:

7.Deletion at the end

8.Display

9.Exit

Enter ur choice:4

Eneter node information to be inserted:30

---

circular linked list ADT operations are:

1.create

2.Insertion at the beginning

3.Insertion after the given info:

4.Insertion at the end

5.deletion at the beginning

6.Deletion the given info:

7.Deletion at the end

8.Display

9.Exit

```
 Enter ur choice:8
10->20->30->
________________
 circular linked list ADT operations are:
_______________
        1.create
        2.Insertion at the beginning
        3.Insertion after the given info:
        4.Insertion at the end
        5.deletion at the beginning
        6.Deletion the given info:
        7.Deletion at the end
        8.Display
        9.Exit
 Enter ur choice:5
 deleted element is:10
________________
 circular linked list ADT operations are:
_______________
        1.create
        2.Insertion at the beginning
        3.Insertion after the given info:
        4.Insertion at the end
        5.deletion at the beginning
        6.Deletion the given info:
        7.Deletion at the end
        8.Display
        9.Exit
 Enter ur choice:8
20->30->
```

circular linked list ADT operations are:

1.create

2.Insertion at the beginning

3.Insertion after the given info:

4.Insertion at the end

5.deletion at the beginning

6.Deletion the given info:

7.Deletion at the end

8.Display

9.Exit

Enter ur choice:9

4.1.AIM:

Write a program that implement stack (its operations) using Arrays

CODE:

```
#include<stdio.h>
#include<stdlib.h>
#define max 50

int top =-1;
int stack[50];

void push();
void pop();
void display();
void main()
{
int ch;
system("clear");
while(1)
{
```

```
printf("\n__________");
printf("\n stack ADT operations");
printf("\n_________");
printf("\n\t1.push");
printf("\n\t2.pop");
printf("\n\t3.display");
printf("\n\t4.exit");
printf("\n Enter ur choice");
scanf("%d",&ch);
switch(ch)
{
case 1:push();
break;
case 2:pop();
break;
case 3:display();
break;
case 4:exit(0);
break;
default:printf("\n wrong choice");
}
}
}
void push()
{
int element;
if(top==max-1)
printf("\nSTACK overflow");
else
{
printf("\n enter elmnt to be inserted:");
scanf("%d",&element);
top=top+1;
```

```
stack[top]=element;
}
}

void pop()
{
if(top==-1)
printf("\n  stack underflow\n");
else
{
printf("\n deleted element is:%d\n",stack[top]);
top=top-1;
}
}

void display()
{
int i;
if(top==-1)
printf("\n stack is empty\n");
else
{
printf("Stack elements are : \n");
for(i=top;i>=0;i--)
printf("%d->",stack[i]);
}
}
```

OUTPUT:

```
__________
 stack ADT operations
_________
     1.push
     2.pop
```

```
          3.display
          4.exit
 Enter ur choice:1
 enter elmnt to be inserted:10
__________
 stack ADT operations
_________
          1.push
          2.pop
          3.display
          4.exit
 Enter ur choice:1
 enter elmnt to be inserted:20
__________
 stack ADT operations
_________
          1.push
          2.pop
          3.display
          4.exit
 Enter ur choice1
 enter elmnt to be inserted:30
__________
 stack ADT operations
_________
          1.push
          2.pop
          3.display
          4.exit
 Enter ur choice3
Stack elements are :
30       20       10
```

---

stack ADT operations

1.push

2.pop

3.display

4.exit

Enter ur choice:2

deleted element is:30

---

stack ADT operations

1.push

2.pop

3.display

4.exit

Enter ur choice:3

Stack elements are :

20 10

---

stack ADT operations

1.push

2.pop

3.display

4.exit

Enter ur choice:4

4.2.AIM:

Write a program that implement stack (its operations) using Pointers

CODE:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *top;
void push();
void pop();
void display();
void main()
{
int ch;
system("clear");
while(1)
{
printf("__________");
printf("\n stack usinf sll");
printf("_________\n");
printf("\nt1.push");
printf("\nt2.pop");
printf("\nt3.display");
printf("\nt4.exit");
printf("\n Eneter ur choice");
scanf("%d",&ch);
switch(ch)
{
case 1:push();
break;
```

```
case 2:pop();
break;
case 3:display();
break;
case 4:exit(0);
break;
default:printf("\n wrong choice");
}
}
}
void push()
{
struct node *ptr;
ptr=(struct node*)malloc(sizeof(struct node));
printf("\n enter elmnt to be inserted:");
scanf("%d",&ptr->data);
ptr->next=top;
top=ptr;
}
void pop()
{
struct node *ptr;
if(top==NULL)
printf("\n  stack underflow\n");
else
{
ptr=top;
printf("\n deleted element is:%d\n",ptr->data);
top=ptr->next;
free(ptr);
}
}
```

```
void display()
{
struct node *ptr;
ptr=top;
if(top==NULL)
printf("\n stack is empty\n");
{
while(ptr!=NULL)
{
printf("%d->",ptr->data);
ptr=ptr->next;
}
}
}
```

OUTPUT :

```
___________
 stack using sll
__________
1.push
2.pop
3.display
4.exit
Enter ur choice 1
 enter element to be inserted:10
___________
 stack using sll
__________
1.push
2.pop
3.display
4.exit
 Enter ur choice 1
```

enter element to be inserted:20

____

stack using sll

____

1.push

2.pop

3.display

4.exit

Enter ur choice 1

enter element to be inserted:30

____

stack using sll

____

1.push

2.pop

3.display

4.exit

Enter ur choice 3

30->20->10->

____

stack using sll

____

1.push

2.pop

3.display

4.exit

Enter ur choice 2

deleted element is:30

____

stack using sll

____

1.push

2.pop

3.display

4.exit

Enter ur choice 3

20->10->

____

stack using sll

____

1.push

2.pop

3.display

4.exit

Enter ur choice 4

5.1. AIM:

Write a program that implement Queue (its operations) using Arrays

CODE:

```
#include<stdio.h>
#include<stdlib.h>
#define max 50

int front =-1;
int rear=-1;
int queue[max];

void insertion();
void deletion();
void display();
void main()
{
int ch;
system("clear");
while(1)
{
```

```
printf("\n__________");
printf("\n queue ADT operations");
printf("\n_________");
printf("\n\t1.insertion");
printf("\n\t2.deletion");
printf("\n\t3.display");
printf("\n\t4.exit");
printf("\n Eneter ur choice");
scanf("%d",&ch);
switch(ch)
{
case 1:insertion();
break;
case 2:deletion();
break;
case 3:display();
break;
case 4:exit(0);
break;
default:printf("\n wrong choice");
}
}
}
void insertion()
{
int element;
if(front==-1)
front=0;
if(rear==max-1)
printf("\nQueue overflow");
else
{
printf("\n enter elmnt to be inserted:");
```

```
scanf("%d",&element);
rear=rear+1;
queue[rear]=element;
}
}
void deletion()
{
if(front==-1 || front>rear)
printf("\n  queue underflow\n");
else
{
printf("\n deleted element is:%d\n",queue[front]);
front=front+1;
}
}
void display()
{
int i;
if(front==-1 || front>rear)
printf("\n queue is empty\n");
else
{
printf("Queue elements are : \n");
for(i=front;i<=rear;i++)
printf("%d->",queue[i]);
}
}
```

OUTPUT:

```
__________
 queue ADT operations
_________
          1.insertion
          2.deletion
          3.display
          4.exit
 Enter ur choice:1
 enter elmnt to be inserted:10


__________
 queue ADT operations
_________
          1.insertion
          2.deletion
          3.display
          4.exit
 Eneter ur choice1
 enter elmnt to be inserted:20
__________
 queue ADT operations
_________
          1.insertion
          2.deletion
          3.display
          4.exit
 Eneter ur choice1
 enter elmnt to be inserted:30
__________
 queue ADT operations
_________
          1.insertion
```

2.deletion

3.display

4.exit

Eneter ur choice:3

Queue elements are :

10 20 30

---

queue ADT operations

1.insertion

2.deletion

3.display

4.exit

Enter ur choice:2

deleted element is:10

---

queue ADT operations

1.insertion

2.deletion

3.display

4.exit

Enter ur choice:3

Queue elements are :

20 30

---

queue ADT operations

1.insertion

2.deletion

3.display

4.exit

Enter ur choice:4

5.2.AIM:

Write a program that implement Queue (its operations) using Pointers

CODE:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *front,*rear;
void insert();
void del();
void display();
void main()
{
int ch;
struct node *ptr;
system("clear");
while(1)
{
printf("\n ________");
printf("\n queue ADT using SSL operations are:\n");
printf("\n _");
printf("\n\t1.Insert");
printf("\n\t2.Delete");
printf("\n\t3.Display");
printf("\n\t4.Exit");
printf("\n Eneter ur choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:insert();
```

```
break;
case 2:del();
break;
case 3:display();
break;
case 4:exit(0);
break;
default:printf("\n wrong choice");
}
}
}
void insert()
{
struct node *ptr;
ptr=(struct node*)malloc(sizeof(struct node));
printf("\n Enter node information:");
scanf("%d",&ptr->data);
if(front==NULL)
{
front=ptr;
rear=ptr;
front->next=NULL;
rear->next=NULL;
}
else
{
rear->next=ptr;
rear=ptr;
rear->next=NULL;
}
}
void del()
{
```

```
struct node *ptr;
if(front==NULL)
{
printf("\n Queue underflow");
int count=1;
}
else
{
ptr=front;
printf("\n deleted element is:%d",ptr->data);
front=ptr->next;
}
}
void display()
{
struct node *ptr;
ptr=front;
if(front=NULL)
printf("\n Queue Empty\n");
else
{
while(ptr!=NULL)
{
printf("%d->",ptr->data);
ptr=ptr->next;
}
}
}
```

OUTPUT :

____

queue ADT using SLL operations are:

_

1.Insert

2.Delete

3.Display

4.Exit

Enter ur choice:1

Enter node information:10

____

queue ADT using SLL operations are:

_

1.Insert

2.Delete

3.Display

4.Exit

Enter ur choice:1

Enter node information:20

____

queue ADT using SLL operations are:

_

1.Insert

2.Delete

3.Display

4.Exit

Enter ur choice:1

Enter node information:30

____

queue ADT using SLL operations are:

_

1.Insert

2.Delete

3.Display

4.Exit

Enter ur choice:3

10->20->30->

---

queue ADT using SLL operations are:

_

1.Insert

2.Delete

3.Display

4.Exit

Enter ur choice:4

6.1.AIM:

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

CODE:

```
#include<stdio.h>

void quicksort(int a[],int ,int );
int partition(int a[],int ,int );

void main()
{
int a[20],n,i;
printf("\nEnter number of elements :");
scanf("%d",&n);
printf("\nEnter the elements : ");
for(i=0;i<n;i++) scanf("%d",&a[i]);
quicksort(a,0,n-1);
printf("\nSorted elements are :");
```

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

void quicksort(int a[10],int first,int last)
{
int p;
if(first<last)
{
p=partition(a,first,last);
quicksort(a,first,p-1);
quicksort(a,p+1,last);
}
}

int partition(int a[],int first,int last)
{
int pivot,i,j,temp;
pivot=first;
i=first;
j=last;
while(i<j)
{
while(a[i]<=a[pivot] && i<last) i++;
while(a[j]>a[pivot]) j--;
if(i<j)
{
temp = a[i];
a[i]=a[j];
a[j]=temp;
}
}
temp=a[pivot];
a[pivot]=a[j];
```

```
a[j]=temp;
return j;
}
```

OUTPUT:

Enter number of elements :5

Enter the elements : 5 3 2 4 1

Sorted elements are :1 2 3 4 5

6.2.AIM:

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

CODE:

```
#include<stdio.h>
void swap(int *a,int *b)
{
int temp=*a;
*a=*b;
*b=temp;
}
void heapify(int arr[],int N,int i)
{
int largest=i;
int left,right;
left=2*i+1;
right=2*i+2;
if(left<N&&arr[left]>arr[largest])
largest=left;
if(right<N&&arr[right]>arr[largest])
largest=right;
if(largest!=i)
{
```

```
swap(&arr[i],&arr[largest]);
heapify(arr,N,largest);
}
}
void heapsort(int arr[],int N)
{
for(int i=(N/2-1);i>=0;i--)
heapify(arr,N,i);
for(int i=(N-1);i>=0;i--)
{
swap(&arr[0],&arr[i]);
heapify(arr,i,0);
}
}
void printArray(int arr[],int N)
{
for(int i=0;i<N;i++)
{
printf("%d",arr[i]);
printf("\n");
}
}
int main()
{
int arr[]={12,11,13,5,6,7};
int N=sizeof(arr)/sizeof(arr[0]);
heapsort(arr,N);
printf("sorted array\n");
printArray(arr,N);
}
```

OUTPUT:

sorted array

5

6

7

11

12

13

6.3.AIM:

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

CODE:

```
#include<stdio.h>
#include<stdlib.h>
void merge(int[],int,int,int);
void partition(int[],int,int);
void main()
{
int  a[30],i,n;
printf("\n enter no. of elements:");
scanf("%d",&n);
printf("\n enter elements:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
partition(a,0,n-1);
printf("\n sorted elements are:");
for(i=0;i<n;i++)
printf("%d\t",a[i]);
}
```

```
void partition(int a[],int first,int last)
{
int mid;
if(first<last)
{
mid=(first+last)/2;
partition(a,first,mid);
partition(a,mid+1,last);
merge(a,first,mid,last);
}
}
void merge(int a[],int first,int mid,int last)
{
int b[30],i,j,k,l,size;
i=first;
j=mid+1;
k=0;
size=last-first+1;
while(i<=mid&&j<=last)
{
if(a[i]<a[j])
b[k++]=a[i++];
else
b[k++]=a[j++];
}
while(i<=mid)
b[k++]=a[i++];
while(j<=last)
b[k++]=a[j++];
for(l=0;l<size;l++)
a[first+l]=b[l];
}
```

OUTPUT :

enter no. of elements:5

enter elements:5 3 2 4 1

sorted elements are:1 2 3 4 5

7.AIM:

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

CODE:

```
#include<stdio.h>
#include<stdlib.h>
struct node* create();
void preorder(struct node *);
void postorder(struct node *);
void inorder(struct node *);

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

void main()
{
     struct node* root;
         int ch;
         system("clear");
        while(1)
        {
         printf("\n ____________________________________");
         printf("\n TREE TRAVERSAL METHODS ARE:\n");
         printf("______________________________________");
         printf("\n\t1.CREATE");
         printf("\n\t2.PREORDER");
         printf("\n\t3.INORDER");
         printf("\n\t4.POSTORDER");
         printf("\n\t5.EXIT");
         printf("\n Enter ur choice:");
         scanf("%d",&ch);
         switch(ch)
```

```
            {
             case 1:  root=create();
                          break;
              case 2:  printf("\n The preorder traversal of tree is:");
                        preorder(root);
                          break;
              case 3:  printf("\n The inorder traversal of tree is:");
                       inorder(root);
                           break;
              case 4: printf("\n The postorder traversal of tree is:");
                      postorder(root);
                       break;
              case 5: exit(0);
                          break;
              default: printf("\n wrong choice\n");
             }

        }
    }

struct node* create()
{
  struct node *p;
    int x;
    printf("enter node data(-1 for no data):");
    scanf("%d",&x);
    if(x==-1)
        return NULL;

    p=(struct node*)malloc(sizeof(struct node));
    p->data=x;
    printf("\nEnter left child of %d:\n",x);
    p->left=create();
    printf("\nEnter right child of %d:\n",x);
    p->right=create();
    return p;
}
void preorder(struct node *t)
{
    if(t!=NULL)
    {
         printf("\n%d",t->data);
        preorder(t->left);
        preorder(t->right);

    }
}

void inorder(struct node *t)
```

```
{
    if(t!=NULL)
    {
       inorder(t->left);
           printf("\n%d",t->data);
        inorder(t->right);
    }

}

void postorder(struct node *t)
{
    if(t!=NULL)
    {
        postorder(t->left);
         postorder(t->right);
           printf("\n%d",t->data);
    }
}
```

OUTPUT:

```
 ____________________________________
  TREE TRAVERSAL METHODS ARE:
______________________________________
          1.CREATE
          2.PREORDER
          3.INORDER
          4.POSTORDER
          5.EXIT
 Enter ur choice:1
enter node data(-1 for no data):10
Enter left child of 10:
enter node data(-1 for no data):5
Enter left child of 5:
enter node data(-1 for no data):3
Enter left child of 3:
enter node data(-1 for no data):-1
Enter right child of 3:
enter node data(-1 for no data):4
Enter left child of 4:
```

enter node data(-1 for no data):-1

Enter right child of 4:

enter node data(-1 for no data):-1

Enter right child of 5:

enter node data(-1 for no data):-1

Enter right child of 10:

enter node data(-1 for no data):12

Enter left child of 12:

enter node data(-1 for no data):11

Enter left child of 11:

enter node data(-1 for no data):-1

Enter right child of 11:

enter node data(-1 for no data):-1

Enter right child of 12:

enter node data(-1 for no data):-1

TREE TRAVERSAL METHODS ARE:

1.CREATE

2.PREORDER

3.INORDER

4.POSTORDER

5.EXIT

Enter ur choice:2

The preorder traversal of tree is:

10

5

3

4

12

11

TREE TRAVERSAL METHODS ARE:

1.CREATE

2.PREORDER

3.INORDER

4.POSTORDER

5.EXIT

Enter ur choice:3

The inorder traversal of tree is:

3

4

5

10

11

12

TREE TRAVERSAL METHODS ARE:

1.CREATE

2.PREORDER

3.INORDER

4.POSTORDER

5.EXIT

Enter ur choice:4

The postorder traversal of tree is:

4

3

5

11

12

10

TREE TRAVERSAL METHODS ARE:

```
        1.CREATE
        2.PREORDER
        3.INORDER
        4.POSTORDER
        5.EXIT
 Enter ur choice:5
```

8.1.AIM:

Write a program to implement Binary Search tree

CODE:

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

struct BSTNode
{
int data;
struct BSTNode *left;
struct BSTNode *right;
};
//for creating new Node
struct BSTNode* GetNewNode(int x)
{
struct BSTNode* newNode
= (struct BSTNode*)malloc(sizeof(struct BSTNode));
newNode->data = x;
newNode->left = NULL;
newNode->right = NULL;
return newNode;
}
struct BSTNode *insertTree(struct BSTNode *p,int key);
struct BSTNode *search(struct BSTNode *root,int key);
struct BSTNode *deleteTree(struct BSTNode *root,int key);
void inorder(struct BSTNode *p);
```

```
void preorder(struct BSTNode *p);
void postorder(struct BSTNode *p);
struct BSTNode *insertTree(struct BSTNode *p,int key)
{
if(p==NULL)
p=GetNewNode(key);
else if(key<p->data)
p->left=insertTree(p->left,key);
else
p->right=insertTree(p->right,key);
return p;
}
struct BSTNode* search(struct BSTNode *root,int key)
{
struct BSTNode *p=root;
while(p!=NULL)
{
if(key==p->data) return p;
else if(key<p->data)
p=p->left;
else
p=p->right;
}
return NULL;
}
struct BSTNode* deleteTree(struct BSTNode *root,int key)
{
struct BSTNode *p;
struct BSTNode *parent=root;
struct BSTNode *inorderSucc;
if(root==NULL)
{
printf("can't delete tree is empty");
```

```
return NULL;
}
p=root;
//tree having only one node
if(root->data==key&&root->left==NULL&&root->right==NULL)
{
root=NULL;
return root;
}
//key matching root and having either left and right child
if(p!=NULL &&p->data==key)
{
if(p->right!=NULL&&p->left==NULL)
{
root=p->right;
return root;
}
else
if(p->left!=NULL&&p->right==NULL)
{
root=p->left;
return root;
}
}
while(p!=NULL&&p->data!=key)
{
parent =p;
if(key<p->data)
p=p->left;
else
p=p->right;
}
if(p==NULL)
```

```
{
printf("%d node not found",key);
return NULL;
}
/* find inorder successor of the node being deleted
and its parent*/
if(p->left!=NULL &&p->right!=NULL)
{
parent=p;
inorderSucc=p->right;
while(inorderSucc->left!=NULL)
{
parent=inorderSucc;
inorderSucc=inorderSucc->left;
}
p->data=inorderSucc->data;
p=inorderSucc;
}
if(p->left==NULL &&p->right==NULL)
{
if(parent->left==p)
parent->left=NULL;
else
parent->right=NULL;
}
if(p->left==NULL &&p->right!=NULL)
{
if(parent->left==p)
parent->left=p->right;
else parent->right=p->right;
}
if(p->left!=NULL &&p->right==NULL)
{
```

```
if(parent->left==p)
parent->left=p->left;
else parent->right=p->left;
}
return root;
}
void inorder(struct BSTNode *p)
{
if(p!=NULL)
{
inorder(p->left);
printf("%d\t",p->data);
inorder(p->right);
}
}
void preorder(struct BSTNode *p)
{
if(p!=NULL)
{
printf("%d\t",p->data);
preorder(p->left);
preorder(p->right);
}
}
void postorder(struct BSTNode *p)
{
if(p!=NULL)
{
postorder(p->left);
postorder(p->right);
printf("%d\t",p->data);
}
}
```

```
void main()
{
struct BSTNode *item,*root=NULL;
int ch;
int element;
while(ch!=5)
{
printf("\n 1.Insert 2.Delete 3.Search 4 .Traversal 5.Exit \n");
printf("\nEnter your choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("\nenter element to insert");
scanf("%d",&element);
root=insertTree(root,element); break;
case 2:
printf("\nenter element to be deleted");
scanf("%d",&element);
root=deleteTree(root,element); break;
case 3:
printf("\nenter element to search");
scanf("%d",&element);
item=search(root,element);
if(item!=NULL)
printf("\nitem found in tree: %d",item->data);
else
printf("\nitem not found");
break;
case 4:
printf("\nPreorder:");preorder(root);
printf("\ninorder:");inorder(root);
printf("\npostorder:");postorder(root);
```

```
break;
case 5:exit(0);
}
}
}
```

OUTPUT :

```
 1.Insert 2.Delete 3.Search 4 .Traversal 5.Exit
Enter your choice:1
enter element to insert20
 1.Insert 2.Delete 3.Search 4 .Traversal 5.Exit
Enter your choice:1
enter element to insert18
 1.Insert 2.Delete 3.Search 4 .Traversal 5.Exit
Enter your choice:1
enter element to insert19
 1.Insert 2.Delete 3.Search 4 .Traversal 5.Exit
Enter your choice:1
enter element to insert21
 1.Insert 2.Delete 3.Search 4 .Traversal 5.Exit
Enter your choice:3
enter element to search19
item found in tree: 19
 1.Insert 2.Delete 3.Search 4 .Traversal 5.Exit
Enter your choice:2
enter element to be deleted20
 1.Insert 2.Delete 3.Search 4 .Traversal 5.Exit
Enter your choice:4
Preorder:21   18      19
inorder:18    19      21
postorder:19 18       21
 1.Insert 2.Delete 3.Search 4 .Traversal 5.Exit
Enter your choice:5
```

8.2.AIM:

Write a program to implement B Trees

CODE:

```
#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

8.3.AIM:

Write a program to implement B+ Trees

CODE:

```
#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 0x564d1f40a330 -- key 15, value 21.
```

8.4.AIM:

Write a program to implement AVL trees

CODE:

```
#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

9.AIM:

Write a program to implement the graph traversal methods.

CODE:

```
#include<stdio.h>
#include<stdlib.h>
void create_adjacency();
void dfs(int);
void bfs(int);
int v,n,adjm[20][20],visited[20];
void main()
{
int i,ch;
while(1)
{
printf("\n\t________________");
printf("|n\t Graph ADT operations are:");
printf("\n\t1.create adjacency matrix");
printf("\n\t2.Dept first search(DFS)");
printf("\n\t3.Breadth  first search(BFS)");
printf("\n\t4.exit");
printf("\n enter ur choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:create_adjacency();
break;
case 2:printf("\n  enetr starting node for DFS:");
scanf("%d",&v);
for(i=1;i<=n;i++)
visited[i]=0;
dfs(v);
break;
case 3:printf("\n enter starting bode for BFS:");
scanf("%d",&v);
```

```
for(i=1;i<=n;i++)
visited[i]=0;
bfs(v);
break;
case 4:exit(0);
break;
}
}
}
void create_adjacency()
{
int max_edges,i,j,origin,destin;
char graphtype;
printf("\n enter no-of  nodes:");
scanf("%d",&n);
getchar();
printf("\n enter graph type,directed or undirected(d/u):");

scanf("%c",&graphtype);
if(graphtype=='u')
max_edges=(n*(n-1))/2;
else
max_edges=n*(n-1);
for(i=1;i<=max_edges;i++)
{
printf("\n enter edges %d(0 0 to quit):",i);
scanf("%d%d",&origin,&destin);
if(origin==0&&destin==0)
break;
if((origin>n)||(destin>n)||(origin<=0)||(destin<=0))
{
printf("\n Ivalid edges!");
}
```

```
else
{
if(graphtype=='d')
adjm[origin][destin]=1;
else
{
adjm[origin][destin]=1;
adjm[destin][origin]=1;
}
}
}
printf("\n the adjacency matrix is:\n");
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
printf("%d",adjm[i][j]);
printf("\n");
}
}
void dfs(int v)
{
int stack[30],top=-1,node,i,j,t;
top++;
stack[top]=v;
while(top>=0)
{
node=stack[top];
top--;
if(visited[node]==0)
{
printf("%d\t",node);
visited[node]=1;
}
```

```
else
continue;
for(i=n;i>=1;i--)
{
if((adjm[node][i]==1)&&(visited[i]==0))
{
top++;
stack[top]=i;
}
}
}
}
void bfs(int v)
{
int i, front=-1,rear=-1,queue[30];
printf("%d\t",v);
visited[v]=1;
front++;rear++;
queue[rear]=v;
while(front<=rear)
{
v=queue[front];
front++;
for(i=1;i<=n;i++)
{
if((adjm[v][i]==1)&&(visited[i]==0))
{
printf("%d\t",i);
visited[i]=1;
rear++;
queue[rear]=i;
}
}
```

}

}

OUTPUT (for undirected graph):

```
        ________________
 Graph ADT operations are:
        1.create adjacency matrix
        2.Dept first search(DFS)
        3.Breadth  first search(BFS)
        4.exit
 enter ur choice:1
 enter no-of  nodes:3
 enter graph type,directed or undirected(d/u):u
 enter edges 1(0 0 to quit):1 2
 enter edges 2(0 0 to quit):1 3
 enter edges 3(0 0 to quit):3 2
 the adjacency matrix is:
011
101
110
        ________________
 Graph ADT operations are:
        1.create adjacency matrix
        2.Dept first search(DFS)
        3.Breadth  first search(BFS)
        4.exit
 enter ur choice:2
  enetr starting node for DFS:1
1       2       3
        ________________
         Graph ADT operations are:
        1.create adjacency matrix
        2.Dept first search(DFS)
        3.Breadth  first search(BFS)
```

4.exit

enter ur choice:3

enter starting bode for BFS:

2 1 3

________________|n Graph ADT operations are:

1.create adjacency matrix

2.Dept first search(DFS)

3.Breadth first search(BFS)

4.exit

enter ur choice:4

10.1.AIM:

Implement a Pattern matching algorithms using Boyer- Moore

CODE:

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

# define NO_OF_CHARS 256

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

// The preprocessing function for Boyer Moore's bad character heuristic
void badCharHeuristic(char *str, int size, int badchar[NO_OF_CHARS]) {
```

```
    int i;

// Initialize all occurrences as -1
for (i = 0; i < NO_OF_CHARS; i++)
badchar[i] = -1;

// Fill the actual value of last occurrence of a character
for (i = 0; i < size; i++)
badchar[(int) str[i]] = i;
}

void search(char *txt, char *pat) {
    int m = strlen(pat);
    int n = strlen(txt);

    int badchar[NO_OF_CHARS];

    badCharHeuristic(pat, m, badchar);

    int s = 0; // s is shift of the pattern with respect to text
    while (s <= (n - m)) {
        int j = m - 1;

        while (j >= 0 && pat[j] == txt[s + j])
            j--;

        if (j < 0) {
            printf("\n pattern occurs at shift = %d", s);

            s += (s + m < n) ? m - badchar[txt[s + m]] : 1;

        }
```

```
        else
s += max(1, j - badchar[txt[s + j]]);
}
}

int main() {
   char txt[] = "ABAAABCD";
   char pat[] = "ABC";
   search(txt, pat);
   return 0;
}
```

OUTPUT :

Pattern occurs at shift = 4

10.2.AIM:

Implement a Pattern matching algorithms using KMP

CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
void computeLPSArray(char* pat,int M,int* lps);
void KMPSearch(char* pat,char* txt)
{
int M=strlen(txt);
int N=strlen(pat);
int lps[N];
computeLPSArray(pat,N,lps);
int i=0;
int j=0;
while(i<M)
```

```
{
if(txt[i]==pat[j])
i++,j++;
else
{
if(j!=0)
j=lps[j-1];
else
i=i+1;
}
if(j==N)
printf("Found pattern at index  %d",i);
}
}
void computeLPSArray(char* pat,int N,int* lps)
{
int i=0;
lps[0]=0;
int j=1;
while(j<N)
{
if(pat[i]==pat[j])
{
lps[j]=i+1;
i++;j++;
}
else
{
if(i!=0)
{
i=lps[i-1];
}
else
```

```
{
lps[j]=0;
j++;
}
}
}
}
int main()
{
char txt[]="AAAABAAAX";
char pat[]="AAAX";
KMPSearch(pat,txt);
return 0;
}
```

OUTPUT :

Found pattern at index 5

10.3.AIM:

Implement a Pattern matching algorithms using Brute force technique

CODE:

```
#include<stdio.h>
#include<string.h>
int search( char * t,char* p)
{
int M=strlen(t);
int N=strlen(p);
for(int i=0;i<M-N;i++)
{
int j;
for(j=0;j<N;j++)
{
if(p[j]!=t[i+j])
break;
```

```
}
if(j==N)
return i;
}
return -1;
}
int main()
{
int i;
char text[100],pat[50];
printf("enter the string \n");
gets(text);
printf("enter the pattern \n");
gets(pat);
i=search(text,pat);
if(i==-1)
printf("pattern  not found\n");
else
printf("pattern fount at indes %d\n",i);
return 0;
}
```

OUTPUT :

```
enter the string
abdcegbdch
enter the pattern
bdc
pattern fount at index 1
```