

Ex no: 1	ARRAY IMPLEMENTATION OF LIST ADT
Date:	

AIM:

To write a program for List using array implementation.

DESCRIPTION:

A linked list is a sequence of data structures, which are connected together via links.
 Linked List is a sequence of links which contains items. Each link contains a connection to another link.
 Linked list is the second most-used data structure after array.

A linked list is a sequence of data structures, which are connected together via links.
 Linked List is a sequence of links which contains items. Each link contains a connection to another link.
 Linked list is the second most-used data structure after array.

Following are the important terms to understand the concept of Linked List.

- Link – each link of a linked list can store a data called an element.
- Next – each link of a linked list contains a link to the next link called Next.
- Linked List – A Linked List contains the connection link to the first link called First.

ALGORITHM:

Step 1: Create nodes first, last; next, prev and cur then set the value as NULL.

Step 2: Read the list operation type.

Step 3: If operation type is create then process the following steps.

- Allocate memory for node cur.
- Read data in cur's data area.
- Assign cur node as NULL.
- Assign first=last=cur.

Step 4: If operation type is Insert then process the following steps.

- Allocate memory for node cur.
- Read data in cur's data area.
- Read the position the Data to be insert.
 - Availability of the position is true then assign cur's node as first and first=cur. If availability of position is false then do following steps.
- Assign next as cur and count as zero.
- Repeat the following steps until count less than position. 1
- Assign prev as next
- Next as prev of node.
- Add count by one.
- If prev as NULL then display the message INVALID POSITION.
- If prev not equal to NULL then do the following steps.
- Assign cur's node as prev's node.
- Assign prev's node as cur.

Step5: If operation type is delete then do the following steps.

- Read the position .
- Check list is Empty .If it is true display the message List empty.
- If position is first.
- Assign cur as first.
- Assign First as first of node.
- Reallocate the cur from memory.
- If position is last.

- Move the current node to prev.
- cur's node as Null.
- Reallocate the Last from memory.
- Assign last as cur.
- If position is enter Mediate.
- Move the cur to required position.
- Move the Previous to cur's previous position
- Move the Next to cur's Next position.
- Now Assign previous of node as next.
- Reallocate the cur from memory.

Step 6: If operation is traverse.

- Assign current as first.
- Repeat the following steps until cur becomes NULL.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#define MAX 10

void create();
void insert();
void deletion();
void search();
void display();
int a,b[20], n, p, e, f, i, pos;

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

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

switch(ch)
{
case 1:
create();
break;

case 2:
deletion();
break;

case 3:
search();
break;

case 4:
```

```
insert();
break;

case 5:
display();
break;

case 6:
exit();
break;

default:
printf("\n Enter the correct choice:");
}
printf("\n Do u want to continue::");
scanf("\n%c", &g);
}
while(g=='y' || g=='Y');
getch();
}
```

```
void create()
{
printf("\n Enter the number of nodes");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n Enter the Element:",i+1);
scanf("%d", &b[i]);
}
}
```

```
void deletion()
{
printf("\n Enter the position u want to delete::");
scanf("%d", &pos);
if(pos>=n)
{
printf("\n Invalid Location::");
}
else
{
for(i=pos+1;i<n;i++)
{
b[i-1]=b[i];
}
n--;
}
printf("\n The Elements after deletion");
for(i=0;i<n;i++)
{
```

```
printf("\t%d", b[i]);  
}  
}
```

```
void search()  
{  
    printf("\n Enter the Element to be searched:");  
    scanf("%d", &e);  
    f = 0; // flag variable to check if element is found  
    for(i = 0; i < n; i++)  
    {  
        if(b[i] == e)  
        {  
            printf("Value is in the %d Position", i);  
            f = 1;  
            break; // exit loop when element is found  
        }  
    }  
  
    if(f == 0)  
    {  
        printf("Value %d is not found in the list\n", e);  
    }  
}
```

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

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

OUTPUT:

RESULT:

Thus the C program for array implementation of List ADT was created, executed and output was verified successfully.

Ex no: 2	LINKED LIST IMPLEMENTATION OF LIST
Date:	

AIM:

To write a program for List using Linked List implementation.

ALGORITHM:

1. Define a struct for each node in the List. Each node in the List contains data and link to the next node.
2. The operations on the stack are
 - a. INSERT data into the LIST
 - b. DELETE data out of LIST
 - c. TRAVERSE data out of LIST
3. INSERT DATA INTO THE LIST
 - a. Enter the data to be inserted into LIST.
 - b. If LIST is NULL
 - i. The input data is the first node in list.
 - ii. The link of the node is NULL.
 - iii. TOP points to that node.
 - c. If TOP is NOT NULL
 - i. The link of TOP points to the new node.
 - ii. TOP points to that node.
4. POP DATA FROM LIST

PROGRAM:

```
#include
<stdio.h>
#include
<stdlib.h> struct
node {
    int data;
    struct node *next;
};

struct node *start =
NULL; void
insert_at_begin(int); void
insert_at_end(int); void
traverse();
void
delete_from_begin();
void delete_from_end();
int count = 0;
```

```

int main ()
{ int i,
  data;

  for (;;) {
    printf("1. Insert an element at the beginning of linked list.\n");

    printf("2. Insert an element at the end of linked
    list.\n"); printf("3. Traverse linked list.\n");
    printf("4. Delete an element from
    beginning.\n"); printf("5. Delete an element
    from end.\n"); printf("6. Exit\n");
    scanf("%d",
    &i); if (i == 1) {
      printf("Enter value of
      element\n"); scanf("%d",
      &data); insert_at_begin(data);
    }
    else if (i == 2) {
      printf("Enter value of
      element\n"); scanf("%d",
      &data); insert_at_end(data);
    }
    else if (i == 3)
      traverse();
    else if (i == 4)
      delete_from_begin(
    ); else if (i == 5)
      delete_from_end();
    else if (i == 6)
      break
    ; else
      printf("Please enter valid input.\n");
    }

  return 0;
}

void insert_at_begin(int x)
{   struct node *t;

  t = (struct node*)malloc(sizeof(struct
  node)); t->data = x;
  count++;

  if (start == NULL)
  { start = t;
    start->next =
    NULL; return;
  }
}

```

```

t->next =
start; start =
t;
}

```

```

void insert_at_end(int x)
{ struct node *t, *temp;

t = (struct node*)malloc(sizeof(struct
node)); t->data = x;
count++;

```

```

if (start == NULL)
{ start = t;
start->next =
NULL; return;
}

```

```

temp = start;

```

```

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

```

```

temp->next = t;
t->next = NULL;
}

```

```

void traverse()
{ struct node
*t;

t = start;

if (t == NULL) {
printf("Linked list is
empty.\n"); return;
}
printf("There are %d elements in linked list.\n",
count); while (t->next != NULL) {
printf("%d\n", t-
>data); t = t->next;
}
printf("%d\n", t->data); // Print last node
}

```

```

void delete_from_begin()
{ struct node *t;
int n;

if (start == NULL) {
printf("Linked list is
empty.\n"); return;
}

```



```

}

n = start-
>data; t =
start->next;
free(start);
start = t;
count--;

printf("%d deleted from the beginning successfully.\n", n);
}

void delete_from_end()
{ struct node *t, *u;
int n;

if (start == NULL) {
    printf("Linked list is
    empty.\n"); return;
}

count--;

if (start->next == NULL)
    { n = start->data;
    free(start);
    start = NULL;
    printf("%d deleted from end successfully.\n",
    n); return;
}

t = start;

while (t->next != NULL)
    { u = t;
    t = t->next;
    }

n = t->data;
u->next =
NULL; free(t);

printf("%d deleted from end successfully.\n", n);
}

```

OUTPUT:

RESULT:

Thus the C program for Linked List implementation of List was created, executed and

output was verified successfully.

Ex no: 3	POLYNOMIAL MANIPULATION USING LIST ADT
Date:	

AIM:

To write program in C for polynomial addition using List ADT

DESCRIPTION:

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

Example:

$$3x^2+5x+7$$

Linked list representation:

The main fields of polynomial are coefficient and exponent, in linked list it will have one more field called „link“ field to point to next term in the polynomial. If there are “n” terms in the polynomial then “n” such nodes have to be created.

ALGORITHM:

STEP 1: Get the two polynomials. First polynomial is P1 and second polynomial is P2

STEP 2: For addition of two polynomials if exponents of both the polynomials are same then we add the coefficients.

STEP 3: For storing the result we will create the third linked lists say P3. 3: If Exponent of P2 is greater than exponent of P1 then keep the P3 as P2

STEP 4: If Exponent of P2 is greater than exponent of P1 then keep the P3 as P1

STEP 5: If Exponent of P2 is equal to the exponent of P1 then add the coefficient of P1 and coefficient of P2 as coefficient of P3.

STEP 6: Continue the above step from 3 to 5 until end of the two polynomials.

STEP 7: If any of the polynomial is ended keep P3 as the remaining polynomial.

PROGRAM:

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

```
struct poly* link(struct poly*);
void display(struct poly*);
struct poly* addlink(struct poly*,struct poly*, struct poly*);
```

```

struct poly
{
    int coeff, pow;
    struct poly *ptr;
};

int main()
{
    struct poly *head1,*head2,*head3;
    int num;
    head1=NULL;
    head2=NULL;
    head3=NULL;

    while(1)
    {
        printf("\nEnter\n1->To create first polynomial function\n2->To create second polynomial
function\n3->To display first polynomial function\n4->To display second polynomial function\n5->To
add both the polynomial functions\n6->To display the new polynomial created after addition\n");
        scanf("%d",&num);

        switch(num)
        {

            case 1: head1=link(head1);
            break;

            case 2: head2=link(head2);
            break;

            case 3: display(head1);
            break;

            case 4: display(head2);
            break;

            case 5: head3=addlink(head1,head2,head3);
            break;

            case 6: display(head3);
            break;

            default: printf("\nBoss ! Enter the correct value\n");

        }
    }
}

struct poly* link(struct poly *head1)
{
    struct poly *newnode;
    int value,degree;
    newnode=NULL;
    printf("\nEnter the coefficient of the polynomial\n");
    scanf("%d",&value);
    printf("\nEnter the degree for variable x\n");

```

```

scanf("%d",&degree);

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

newnode->pow=degree;
newnode->coeff=value;

if(head1==NULL)
{
    printf("\nThe first node is created\n");
    newnode->ptr=NULL;
}
else
{
    printf("\nThe next node is being linked\n");
    newnode->ptr=head1;
}
return newnode;
}

void display(struct poly* head)
{
    if(head == NULL)
    {
        printf("\nThe polynomial function is not created.\nPlease create a new polynomial function.\n");
    }

    while(head != NULL && head->ptr != NULL)
    {
        printf("The polynomial function has been detected\n");
        printf("%dx^%d+",head->coeff,head->pow);
        head=head->ptr;
    }

    if(head->ptr == NULL)
    {
        printf("\nEntering the last node\n");
        printf("%dx^%d",head->coeff,head->pow);
        head=head->ptr;
    }
}

struct poly* addlink(struct poly* head,struct poly* tail,struct poly *head3 )
{
    struct poly *temp1,*temp,*newnode;
    temp=newnode=temp1=head3=NULL;

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

    if( head == NULL || tail == NULL)
    {
        printf("\nYou have no node to get added\n");
    }
    if(head !=NULL && tail == NULL)
    {
        printf("\nThe existing first polynomial is the final polynomial\n");
    }
}

```

```

    return head;
}
if(head == NULL && tail !=NULL)
{
    printf("\nThe existing second polynomial is the final polynomial\n");
    return tail;
}

head3=newnode;

if(head3==NULL)
{
    printf("\nEnter a node first\n");
}

if(head3 != NULL)
{
while(head !=NULL && tail !=NULL)
{
    if(head->pow > tail->pow)
    {
        temp=tail;
        temp1=head;
        while((head->pow) > (temp->pow))
        {
            printf("\nThe element's degree in first poly. is greater than the second\n");

            if(temp->ptr == NULL)
            {
                newnode->coeff=temp->coeff;
                newnode->pow=temp->pow;
                newnode->ptr=NULL;
                head3=newnode;
                return head3;
                temp1=temp1->ptr;

            }

            temp=temp->ptr;
        }
        printf("\nYou have come to the end of the tail pointer\n");
    }

    if(tail->pow > head->pow)
    {
        temp=head;
        temp1=tail;
        while((tail->pow) > (temp->pow))
        {
            printf("\nThe element's degree in second poly. is greater than the first\n");

            if(temp->ptr ==NULL)
            {
                newnode->coeff=temp->coeff;
                newnode->pow=temp->pow;
                newnode->ptr=NULL;
                head3=newnode;
            }
        }
    }
}
}

```

```

    return head3;
    temp1=temp1->ptr;
}

temp=temp->ptr;
}

printf("\nYou have come to the end of the tail pointer\n");
}

if(head->pow == tail->pow)
{
    temp=head;
    temp1=tail;

    while((temp->pow == temp1->pow))
    {
        newnode->coeff = (temp->coeff + tail->coeff);
        newnode->pow=temp->pow;
        head3=newnode;
        return head3;
        temp=temp->ptr;
        temp1=temp1->ptr;
    }
}
}
}
}

```

OUTPUT:

RESULT:

Thus the C program for polynomial addition using List ADT

Ex no: 4A	ARRAY IMPLEMENTATION OF STACK ADT
Date:	

AIM:

Aim is to write a program in C to implement the stack ADT using array concept that performs all the operations of stack.

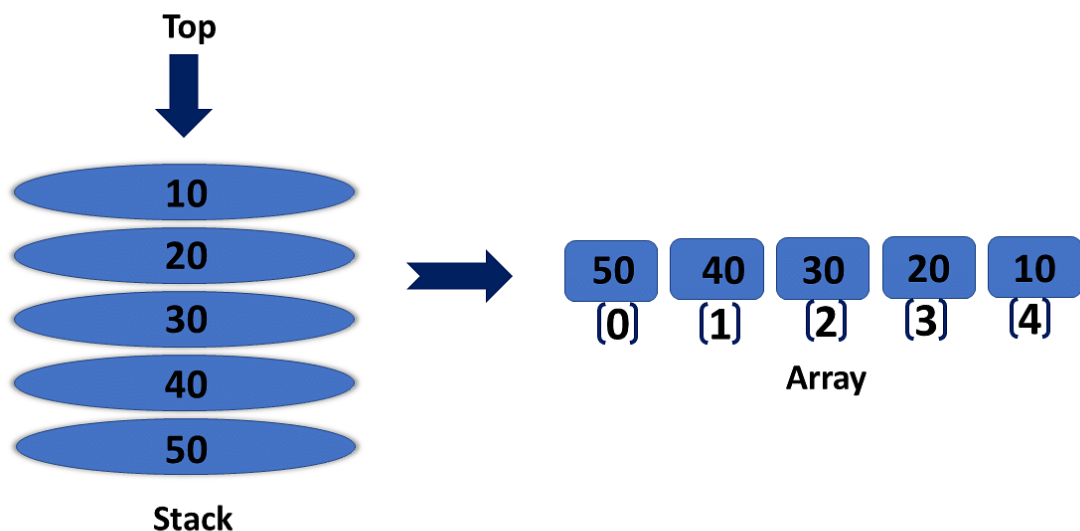
DESCRIPTION:

A stack data structure can be implemented using one dimensional array. But stack implemented using array, can store only fixed number of data values. This implementation is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using LIFO principle with the help of a variable 'top'. Initially top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

PROCEDURE:

Stack Implementation Using Array:

In Stack implementation using arrays, it forms the stack using the arrays. All the operations regarding the stack implementation using arrays.



ALGORITHM:

STEP 1: Define an array to store the element.

STEP 2: Get the users' choice.

STEP 3: If the option is 1 perform creation operation and goto step4.

If the option is 2 perform insertion operation and goto step5.

If the option is 3 perform deletion operation and goto step6.

If the option is 4 perform display operation and goto step7.

STEP 4: Create the stack. Initially get the limit of stack and the get the items. If the limit stack is exceeds print the message unable to create the stack.

STEP 5: Get the element to be pushed. If top pointer exceeds stack capacity. Print Error message that the stack overflow. If not, increment the top pointer by one and store the element in the position which is denoted by top pointer.

STEP 6: If the stack is empty, then print error message that stack is empty. If not fetch element from the position which is denoted by top pointer and decrement the top pointer by one

STEP 7: If the top value is not less than the 0 the stack is display otherwise print the message "stack is empty".

STEP 8: Stop the execution.

PROGRAM:

```
#include<stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
    //clrscr();
    top=-1;
    printf("\n Enter the size of STACK[MAX=100]:");
    scanf("%d",&n);
    printf("\n\t STACK OPERATIONS USING ARRAY");
    printf("\n\t-----");
    printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
    do
    {
        printf("\n Enter the Choice:");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                display();
                break;
            }
        }
    }
```

```

        case 4:
        {
            printf("\n\t EXIT POINT ");
            break;
        }
        default:
        {
            printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
        }

    }
}
while(choice!=4);
return 0;
}
void push()
{
    if(top>=n-1)
    {
        printf("\n\tSTACK is over flow");

    }
    else
    {
        printf(" Enter a value to be pushed:");
        scanf("%d",&x);
        top++;
        stack[top]=x;
    }
}
void pop()
{
    if(top<=-1)
    {
        printf("\n\t Stack is under flow");
    }
    else
    {
        printf("\n\t The popped elements is %d",stack[top]);
        top--;
    }
}
void display()
{
    if(top>=0)
    {
        printf("\n The elements in STACK \n");
        for(i=top; i>=0; i--)
            printf("\n%d",stack[i]);
        printf("\n Press Next Choice");
    }
    else
    {
        printf("\n The STACK is empty");
    }
}

```

}
}
OUTPUT:

RESULT:

Thus a C program for Stack using ADT was implemented successfully.

Ex no: 4B	LINKED LIST IMPLEMENTATION OF STACK ADT
Date:	

AIM:

To write a C program for stack ADT using linked list implementation.

DESCRIPTION:

The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means, stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as 'top' element. That means every newly inserted element is pointed by 'top'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by 'top' by moving 'top' to its next node in the list. The next field of the first element must be always NULL.

ALGORITHM:

1. Define a struct for each node in the stack. Each node in the stack contains data and link to the next node. TOP pointer points to last node inserted in the stack.
2. The operations on the stack are
 - a. PUSH data into the stack
 - b. POP data out of stack
3. PUSH DATA INTO STACK
 - a. Enter the data to be inserted into stack.
 - b. If TOP is NULL
 - i. The input data is the first node in stack.
 - ii. The link of the node is NULL.
 - iii. TOP points to that node.
 - c. If TOP is NOT NULL
 - i. The link of TOP points to the new node.
 - ii. TOP points to that node.
4. POP DATA FROM STACK
 - a. 4a.If TOP is NULL
 - i. the stack is empty
 - b. 4b.If TOP is NOT NULL
 - i. The link of TOP is the current TOP.
 - ii. The pervious TOP is popped from stack.
5. The stack represented by linked list is traversed to display its content.

PROGRAM:

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

struct node
{
    int info;
    struct node *ptr;
}*top,*top1,*temp;

int topelement();
void push(int data);
void pop();
void empty();
void display();
void destroy();
void stack_count();
void create();

int count = 0;

void main()
{
    int no, ch, e;

    printf("\n 1 - Push");
    printf("\n 2 - Pop");
    printf("\n 3 - Top");
    printf("\n 4 - Empty");
    printf("\n 5 - Exit");
    printf("\n 6 - Dipslay");
    printf("\n 7 - Stack Count");
    printf("\n 8 - Destroy stack");

    create();

    while (1)
    {
        printf("\n Enter choice : ");
        scanf("%d", &ch);

        switch (ch)
        {
            case 1:
                printf("Enter data : ");
                scanf("%d", &no);
                push(no);
                break;
            case 2:
                pop();
                break;
            case 3:
                if (top == NULL)
```

```

        printf("No elements in stack");
    else
    {
        e = topelement();
        printf("\n Top element : %d", e);
    }
    break;
case 4:
    empty();
    break;
case 5:
    exit(0);
case 6:
    display();
    break;
case 7:
    stack_count();
    break;
case 8:
    destroy();
    break;
default :
    printf(" Wrong choice, Please enter correct choice ");
    break;
}
}
}

```

/ Create empty stack */*

```
void create()
```

```
{
    top = NULL;
}
```

/ Count stack elements */*

```
void stack_count()
```

```
{
    printf("\n No. of elements in stack : %d", count);
}
```

/ Push data into stack */*

```
void push(int data)
```

```
{
    if (top == NULL)
    {
        top =(struct node *)malloc(1*sizeof(struct node));
        top->ptr = NULL;
        top->info = data;
    }
    else
    {
        temp =(struct node *)malloc(1*sizeof(struct node));
        temp->ptr = top;
        temp->info = data;
        top = temp;
    }
    count++;
}
```

```

}

/* Display stack elements */
void display()
{
    top1 = top;

    if (top1 == NULL)
    {
        printf("Stack is empty");
        return;
    }

    while (top1 != NULL)
    {
        printf("%d ", top1->info);
        top1 = top1->ptr;
    }
}

/* Pop Operation on stack */
void pop()
{
    top1 = top;

    if (top1 == NULL)
    {
        printf("\n Error : Trying to pop from empty stack");
        return;
    }
    else
        top1 = top1->ptr;
    printf("\n Popped value : %d", top->info);
    free(top);
    top = top1;
    count--;
}

/* Return top element */
int topelement()
{
    return(top->info);
}

/* Check if stack is empty or not */
void empty()
{
    if (top == NULL)
        printf("\n Stack is empty");
    else
        printf("\n Stack is not empty with %d elements", count);
}

/* Destroy entire stack */
void destroy()
{
    top1 = top;

```



```
while (top1 != NULL)
{
    top1 = top->ptr;
    free(top);
    top = top1;
    top1 = top1->ptr;
}
free(top1);
top = NULL;

printf("\n All stack elements destroyed");
count = 0;
}
```

OUTPUT:

RESULT:

Thus a C program for Stack ADT using LINKED LIST was implemented successfully.

Ex no: 5A	ARRAY IMPLEMENTATION OF QUEUE ADT
Date:	

AIM:

To write a program for Queue using array implementation.

DESCRIPTION:

A queue data structure can be implemented using one dimensional array. But, queue implemented using array can store only fixed number of data values. The implementation of queue data structure using array is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using FIFO (First In First Out) principle with the help of variables 'front' and 'rear'. Initially both 'front' and 'rear' are set to -1.

Whenever, we want to insert a new value into the queue, increment 'rear' value by one and then insert at that position. Whenever we want to delete a value from the queue, then increment 'front' value by one and then display the value at 'front' position as deleted element.

ALGORITHM:

1. Define a array which stores queue elements..
2. The operations on the queue are
 - a. a)INSERT data into the queue
 - b. b)DELETE data out of queue
3. INSERT DATA INTO queue
 - a. Enter the data to be inserted into queue.
 - b. If TOP is NULL
 - i. The input data is the first node in queue.
 - ii. The link of the node is NULL.
 - iii. TOP points to that node.
 - c. If TOP is NOT NULL
 - i. The link of TOP points to the new node.
 - ii. TOP points to that node.
4. DELETE DATA FROM queue
 - a. If TOP is NULL
 - i. the queue is empty
 - b. If TOP is NOT NULL
 - i. The link of TOP is the current TOP.
 - ii. The pervious TOP is popped from queue.
5. The queue represented by linked list is traversed to display its content.

PROGRAM:

```
#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");
    }
}

```

OUTPUT:

RESULT:

Thus a C program for Queue using Array implementation was implemented successfully

Ex no: 5B	LINKED LIST IMPLEMENTATION OF QUEUE ADT
Date:	

AIM:

To write a C program for Queue using Linked implementation.

DESCRIPTION:

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data(enqueue) and the other is used to remove data(dequeue). Queue follows First-In-First-Out Methodology, i.e.,the data item stored first will be accessed first.

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues.

- enqueue () – add (store) an item to the queue.
- dequeue () – remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are

- peek() – Gets the element at the front of the queue without removing it.
- isfull() – Checks if the queue is full.
- isempty() – Checks if the queue is empty.

ALGORITHM:

1. Define a struct for each node in the queue. Each node in the queue contains data and link to the next node. Front and rear pointer points to first and last node inserted in the queue.
2. The operations on the queue are
 - a. INSERT data into the queue
 - b. DELETE data out of queue
3. INSERT DATA INTO queue
 - a. Enter the data to be inserted into queue.
 - b. If TOP is NULL
 - i. The input data is the first node in queue.
 - ii. The link of the node is NULL.
 - iii. TOP points to that node.
 - c. If TOP is NOT NULL
 - i. The link of TOP points to the new node.

- ii. TOP points to that node.
- 4. DELETE DATA FROM queue a, If TOP is NULL
 - i. The queue is empty
- b. If TOP is NOT NULL
 - i. The link of TOP is the current TOP.
 - ii. The pervious TOP is popped from queue.

The queue represented by linked list is traversed to display its content of the Queue ADT

PROGRAM:

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

struct node
{
    int info;
    struct node *ptr;
}*front,*rear,*temp,*front1;

int frontelement();
void enq(int data);
void deq();
void empty();
void display();
void create();
void queuesize();

int count = 0;

void main()
{
    int no, ch, e;

    printf("\n 1 - Enque");
    printf("\n 2 - Deque");
    printf("\n 3 - Front element");
    printf("\n 4 - Empty");
    printf("\n 5 - Exit");
    printf("\n 6 - Display");
    printf("\n 7 - Queue size");
    create();
    while (1)
    {
        printf("\n Enter choice : ");
        scanf("%d", &ch);
        switch (ch)
        {
            case 1:
                printf("Enter data : ");
                scanf("%d", &no);
                enq(no);
                break;
            case 2:
```

```

        deq();
        break;
case 3:
    e = frondelement();
    if (e != 0)
        printf("Front element : %d", e);
    else
        printf("\n No front element in Queue as queue is empty");
        break;
case 4:
    empty();
    break;
case 5:
    exit(0);
case 6:
    display();
    break;
case 7:
    queuesize();
    break;
default:
    printf("Wrong choice, Please enter correct choice ");
    break;
}
}
}

```

/* Create an empty queue */

```

void create()
{
    front = rear = NULL;
}

```

/* Returns queue size */

```

void queuesize()
{
    printf("\n Queue size : %d", count);
}

```

/* Enqueing the queue */

```

void enq(int data)
{
    if (rear == NULL)
    {
        rear = (struct node *)malloc(1*sizeof(struct node));
        rear->ptr = NULL;
        rear->info = data;
        front = rear;
    }
    else
    {
        temp=(struct node *)malloc(1*sizeof(struct node));
        rear->ptr = temp;
        temp->info = data;
        temp->ptr = NULL;

        rear = temp;
    }
}

```



```

    }
    count++;
}

/* Displaying the queue elements */
void display()
{
    front1 = front;

    if ((front1 == NULL) && (rear == NULL))
    {
        printf("Queue is empty");
        return;
    }
    while (front1 != rear)
    {
        printf("%d ", front1->info);
        front1 = front1->ptr;
    }
    if (front1 == rear)
        printf("%d", front1->info);
}

/* Dequeing the queue */
void deq()
{
    front1 = front;

    if (front1 == NULL)
    {
        printf("\n Error: Trying to display elements from empty queue");
        return;
    }
    else
        if (front1->ptr != NULL)
        {
            front1 = front1->ptr;
            printf("\n Dequed value : %d", front->info);
            free(front);
            front = front1;
        }
        else
        {
            printf("\n Dequed value : %d", front->info);
            free(front);
            front = NULL;
            rear = NULL;
        }
        count--;
}

/* Returns the front element of queue */
int frontelement()
{
    if ((front != NULL) && (rear != NULL))
        return(front->info);
    else

```

```
        return 0;
    }

    /* Display if queue is empty or not */
    void empty()
    {
        if ((front == NULL) && (rear == NULL))
            printf("\n Queue empty");
        else
            printf("Queue not empty");
    }
}
```

OUTPUT:

RESULT:

Thus a C program for Queue using Linked list was implemented successfully

Ex no: 6	INFIX TO POSTFIX CONVERSION
Date:	

AIM:

To write a C program to convert any INFIX expression to POSTFIX expression

ALGORITHM:

STEP 1: First Start scanning the expression from left to right

STEP 2: If the scanned character is an operand, output it, i.e. print

STEP 3: Else

- If the precedence of the scanned operator is higher than the precedence of the operator in the stack(or stack is empty or has '('), then push operator in the stack
- Else, pop all the operators that have greater or equal precedence than the scanned operator. Once you pop them push this scanned operator. (If we see a parenthesis while popping then stop and push scanned operator in the stack.

STEP 4: If the scanned character is an '(', push it to the stack.

STEP 5: If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.

STEP 6: Now, we should repeat the steps 2 – 6 until the whole infix i.e. whole characters are scanned.

STEP 7: Print output

STEP 8: Do the pop and output (print) until stack is not empty

PROGRAM:

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX 20
```

```
char stk[20];
int top = -1;
```

```
int isEmpty(){
    return top == -1;
}
```

```

int isFull(){
    return top == MAX - 1;
}

char peek(){
    return stk[top];
}

char pop(){
    if(isEmpty())
        return -1;

    char ch = stk[top];
    top--;
    return(ch);
}

void push(char oper){
    if(isFull())
        printf("Stack Full!!!!");

    else{
        top++;
        stk[top] = oper;
    }
}

// A utility function to check if the given character is operand
int checkIfOperand(char ch)
{
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}

// Fucntion to compare precedence
// If we return larger value means higher precedence
int precedence(char ch)
{
    switch (ch)
    {
        case '+':
        case '-':
            return 1;

        case '*':
        case '/':
            return 2;

        case '^':
            return 3;
    }
    return -1;
}

// The driver function for infix to postfix conversion
int covertInfixToPostfix(char* expression)
{
    int i, j;

```

```

for (i = 0, j = -1; expression[i]; ++i)
{
    // Here we are checking is the character we scanned is operand or not
    // and this adding to to output
    if (checkIfOperand(expression[i]))
        expression[++j] = expression[i];

    // Here, if we scan character '(', we need push it to the stack.
    else if (expression[i] == '(')
        push(expression[i]);

    // Here, if we scan character is an ')', we need to pop and print from the stack
    // do this until an '(' is encountered in the stack.
    else if (expression[i] == ')')
    {
        while (!isEmpty() && peek() != '(')
            expression[++j] = pop();
        if (!isEmpty() && peek() != '(')
            return -1; // invalid expression
        else
            pop();
    }
    else // if an opertor
    {
        while (!isEmpty() && precedence(expression[i]) <= precedence(peek()))
            expression[++j] = pop();
        push(expression[i]);
    }
}

// Once all inital expression characters are traversed
// adding all left elements from stack to exp
while (!isEmpty())
    expression[++j] = pop();

expression[++j] = '\0';
printf( "%s", expression);
}

int main()
{
    char expression[] = "((a+(b*c))-d)";
    covertInfixToPostfix(expression);
    return 0;
}

```

OUTPUT:

RESULT:

Thus a C program to convert any INFIX expression to POSTFIX expression was implemented successfully.

Ex no: 7	CIRCULAR QUEUE
Date:	

AIM:

To write a C program to implement CIRCULAR QUEUE

OPERATIONS ON CIRCULAR QUEUE:

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

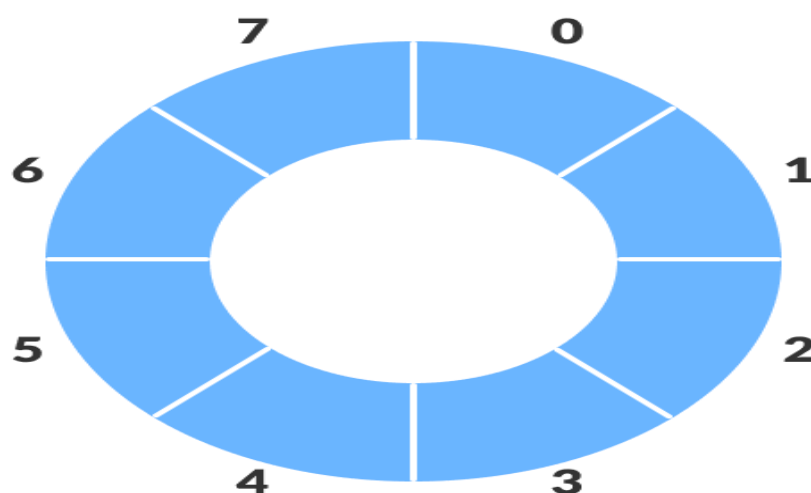
Steps:

1. Check whether queue is Full – Check $((\text{rear} == \text{SIZE}-1 \ \&\& \ \text{front} == 0) \parallel (\text{rear} == \text{front}-1))$.
2. If it is full then display Queue is full. If queue is not full then, check if $(\text{rear} == \text{SIZE} - 1 \ \&\& \ \text{front} != 0)$ if it is true then set $\text{rear}=0$ and insert element.

- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.

Steps:

1. Check whether queue is Empty means check $(\text{front} == -1)$.
2. If it is empty then display Queue is empty. If queue is not empty then step 3
3. Check if $(\text{front} == \text{rear})$ if it is true then set $\text{front} = \text{rear} = -1$ else check if $(\text{front} == \text{size}-1)$, if it is true then set $\text{front}=0$ and return the element.



PROGRAM:

```
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
int cqueue[6];
int front = -1, rear = -1, n=6;
void enqueue(int val){
    if ((front == 0 && rear == n-1) || (front == rear+1)) {
        printf("Queue Overflow \n");
        return;
    }
    if (front == -1) {
        front = 0;
        rear = 0;
    }
    else {
        if (rear == n - 1)
            rear = 0;
        else
            rear = rear + 1;
    }
    cqueue[rear] = val ;
}
void dequeue(){
    if (front == -1) {
        printf("Queue Underflow\n");
        return ;
    }
    printf("Element deleted from queue is : %d ", cqueue[front]);
    if (front == rear) {
        front = -1;
        rear = -1;
    }
    else {
        if (front == n - 1)
            front = 0;
        else
            front = front + 1;
    }
}
void display(){
    int f = front, r = rear;
    if (front == -1) {
        printf("Queue is empty");
        return;
    }
    printf("Queue elements are :\n");
    if (f <= r) {
        while (f <= r){
            printf("%d", cqueue[f]);
            f++;
        }
    }
}
```

```

else {
    while (f <= n - 1) {
        printf("%d",cqueue[f]);
        f++;
    }
    f = 0;
    while (f <= r) {
        printf("%d",cqueue[f]);
        f++;
    }
}

}

int menu()

{
    int choice;
    printf("\n 1.Add value to the list");
    printf("\n 2. Delete value to the list");
    printf("\n 3. Travesre/View List");
    printf("\n 4. exit");
    printf("\n Please enter your choice: \t");
    scanf("%d",&choice);
    return(choice);
}

void main()
{
    int value;
    while(1){
        switch(menu()){
            case 1:
                printf("Input for insertion: ");
                scanf("%d",&value);
                enqueue(value);
                break;

            case 2:
                dequeue();
                break;

            case 3:
                display();
                break;

            case 4:
                exit(0);
            default:
                printf("invalid choice");
        }
    }
    getch();
}

```

OUTPUT:

RESULT:

Thus a C program to implement Circular Queue was executed successfully.

Ex no: 8	IMPLEMENTATION OF BINARY SEARCH TREES AND TREE TRAVERSAL
Date:	

AIM:

To write a C program to implementation of binary search tree.

DESCRIPTION:

A Binary Search Tree (BST) is a tree in which all the nodes follow the below- mentioned properties

The left sub-tree of a node has a key less than or equal to its parent node's key. The right sub-tree of a node has a key greater than to its parent node's key. Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as

$$\text{left_subtree (keys)} \leq \text{node (key)} \leq \text{right_subtree (keys)}$$

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties.

Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved. Following is a pictorial representation of BST

Basic Operations

Following are the basic operations of a tree

- Search – Searches an element in a tree.
- Insert – Inserts an element in a tree.
- Pre-order Traversal – Traverses a tree in a pre-order manner.
- In-order Traversal – Traverses a tree in an in-order manner.
- Post-order Traversal – Traverses a tree in a post-order manner.

ALGORITHM:

1. Declare function create (), search (), delete (), Display ().
2. Create a structure for a tree contains left pointer and right pointer.
3. Insert an element is by checking the top node and the leaf node and the operation will be performed.
4. Deleting an element contains searching the tree and deleting the item.
5. Display the Tree elements.

PROGRAM:

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

typedef struct BST {
    int data;
    struct BST *lchild, *rchild;
} node;

void insert(node *, node *);
void inorder(node *);
void preorder(node *);
void postorder(node *);
node *search(node *, int, node **);

void main() {
    int choice;
    char ans = 'N';
    int key;
    node *new_node, *root, *tmp, *parent;
    node *get_node();
    root = NULL;
    clrscr();

    printf("\nProgram For Binary Search Tree ");
    do {
        printf("\n1.Create");
        printf("\n2.Search");
        printf("\n3.Recursive Traversals");
        printf("\n4.Exit");
        printf("\nEnter your choice :");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                do {
                    new_node = get_node();
                    printf("\nEnter The Element ");
                    scanf("%d", &new_node->data);

                    if (root == NULL) /* Tree is not Created */
                        root = new_node;
                    else
                        insert(root, new_node);

                    printf("\nWant To enter More Elements?(y/n)");
                    ans = getch();
                } while (ans == 'y');
```

```

        break;

case 2:
    printf("\nEnter Element to be searched :");
    scanf("%d", &key);

    tmp = search(root, key, &parent);
    printf("\nParent of node %d is %d", tmp->data, parent->data);
    break;

case 3:
    if (root == NULL)
        printf("Tree Is Not Created");
    else {
        printf("\nThe Inorder display : ");
        inorder(root);
        printf("\nThe Preorder display : ");
        preorder(root);
        printf("\nThe Postorder display : ");
        postorder(root);
    }
    break;
}
} while (choice != 4);
}
/*
Get new Node
*/
node *get_node() {
    node *temp;
    temp = (node *) malloc(sizeof(node));
    temp->lchild = NULL;
    temp->rchild = NULL;
    return temp;
}
/*
This function is for creating a binary search tree
*/
void insert(node *root, node *new_node) {
    if (new_node->data < root->data) {
        if (root->lchild == NULL)
            root->lchild = new_node;
        else
            insert(root->lchild, new_node);
    }

    if (new_node->data > root->data) {
        if (root->rchild == NULL)
            root->rchild = new_node;
        else
            insert(root->rchild, new_node);
    }
}
/*
This function is for searching the node from

```

binary Search Tree

```
*/  
node *search(node *root, int key, node **parent) {  
    node *temp;  
    temp = root;  
    while (temp != NULL) {  
        if (temp->data == key) {  
            printf("\nThe %d Element is Present", temp->data);  
            return temp;  
        }  
        *parent = temp;  
  
        if (temp->data > key)  
            temp = temp->lchild;  
        else  
            temp = temp->rchild;  
    }  
    return NULL;  
}
```

```
/*  
This function displays the tree in inorder fashion  
*/
```

```
void inorder(node *temp) {  
    if (temp != NULL) {  
        inorder(temp->lchild);  
        printf("%d", temp->data);  
        inorder(temp->rchild);  
    }  
}
```

```
/*  
This function displays the tree in preorder fashion  
*/
```

```
void preorder(node *temp) {  
    if (temp != NULL) {  
        printf("%d", temp->data);  
        preorder(temp->lchild);  
        preorder(temp->rchild);  
    }  
}
```

```
/*  
This function displays the tree in postorder fashion  
*/
```

```
void postorder(node *temp) {  
    if (temp != NULL) {  
        postorder(temp->lchild);  
        postorder(temp->rchild);  
        printf("%d", temp->data);  
    }  
}
```

```
}  
}
```

OUTPUT:

RESULT:

Thus a C program to implement binary search tree was executed successfully

Ex no: 9	IMPLEMENTATION OF AVL TREES
Date:	

AIM:-

To write a C program to implement insertion in AVL trees.

ALGORITHM:-

AVL Rotations

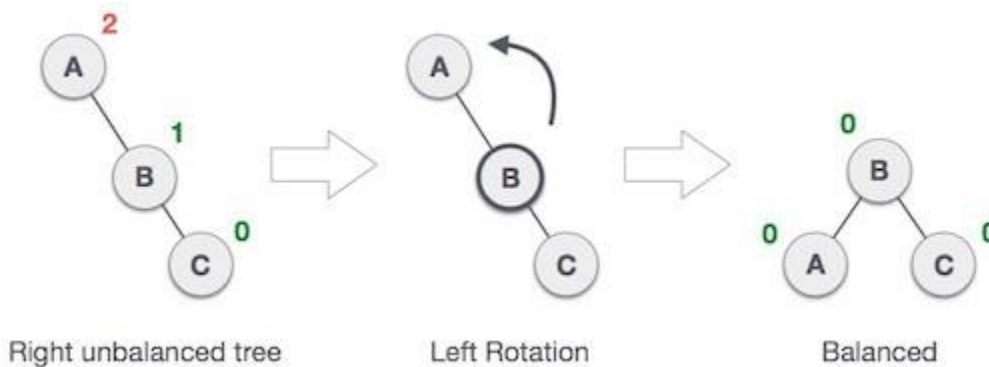
To balance itself, an AVL tree may perform the following four kinds of rotations –

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation:

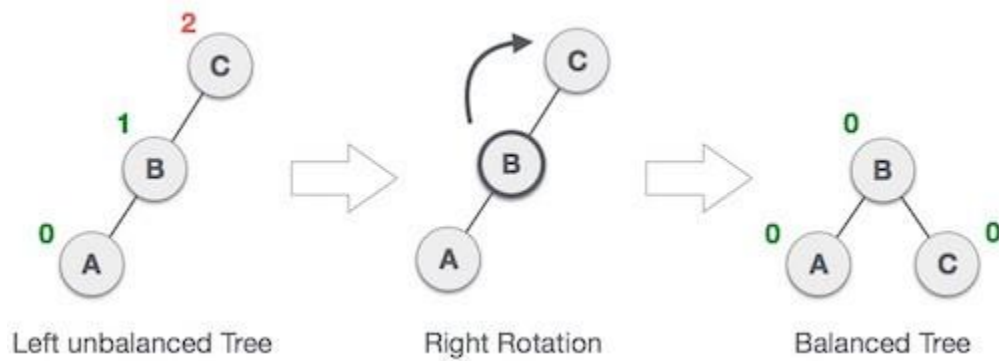
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left- subtree of B.

Right Rotation:

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

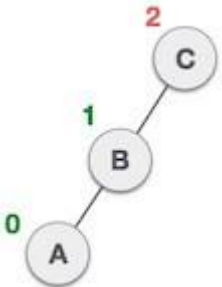
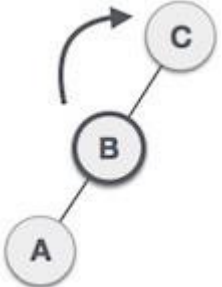
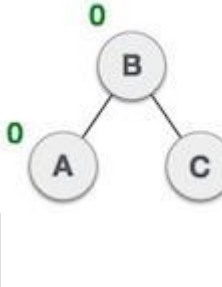


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation:

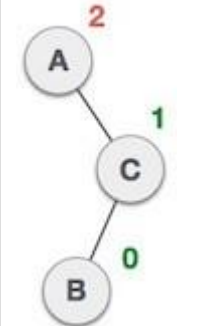
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

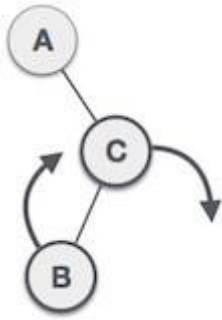
State	Action
	<p>A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.</p>
	<p>We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.</p>

	<p>Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.</p>
	<p>We shall now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.</p>
	<p>The tree is now balanced.</p>

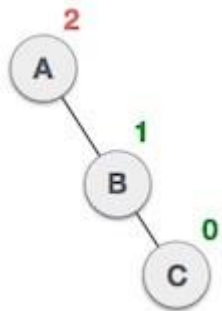
Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

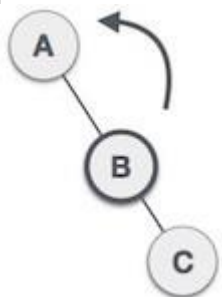
State	Action
	<p>A node has been inserted into the left subtree of the right subtree. This makes A, an unbalanced node with balance factor 2.</p>



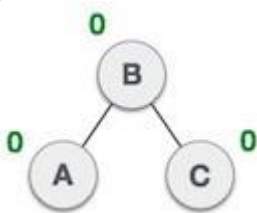
First, we perform the right rotation along C node, making C the right subtree of its own left subtree B. Now, B becomes the right subtree of A.



Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.



A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B.



The tree is now balanced.

PROGRAM:

// AVL tree implementation in C

```
#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:

RESULT:

Thus a C program to implement insertion in AVL trees was executed successfully.

Ex no: 10	BFS and DFS Of a GRAPH
Date:	

AIM:

To write a C program implement DFS and BFS graph traversal.

DESCRIPTION:

Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. BFS and its application in finding connected components of graphs Breadth-first search can be generalized to graphs.

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

ALGORITHM:**DFS**

1. Define a Stack of size total number of vertices in the graph.
2. Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.
3. Visit any one of the adjacent vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.
4. Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
5. When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.
6. Repeat steps 3, 4 and 5 until stack becomes Empty.
7. When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

BFS

1. Define a Queue of size total number of vertices in the graph.
2. Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
3. Visit all the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.
4. When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

5. Repeat step 3 and 4 until queue becomes empty.
6. When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

PROGRAM:

```
#include<stdio.h>

int q[20],top=-1,front=-1,rear=-1,a[20][20],vis[20],stack[20];

int delete();

void add(int item);

void bfs(int s,int n);

void dfs(int s,int n);

void push(int item);

int pop();


void main()
{
    int n,i,s,ch,j;

    char c,dummy;

    printf("ENTER THE NUMBER VERTICES ");

    scanf("%d",&n);

    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            printf("ENTER 1 IF %d HAS A NODE WITH %d ELSE 0 ",i,j);

            scanf("%d",&a[i][j]);

        }
    }
}
```

```
}

printf("THE ADJACENCY MATRIX IS\n");

for(i=1;i<=n;i++)

{

for(j=1;j<=n;j++)

{

printf(" %d",a[i][j]);

}

printf("\n");

}


do

{

for(i=1;i<=n;i++)

vis[i]=0;

printf("\nMENU");

printf("\n1.B.F.S");

printf("\n2.D.F.S");

printf("\nENTER YOUR CHOICE");

scanf("%d",&ch);

printf("ENTER THE SOURCE VERTEX :");

scanf("%d",&s);


switch(ch)

{
```

```
case 1:bfs(s,n);

break;

case 2:

dfs(s,n);

break;

}

printf("DO U WANT TO CONTINUE(Y/N) ? ");

scanf("%c",&dummy);

scanf("%c",&c);

}while((c=='y')||(c=='Y'));

}

//*****BFS(breadth-first search) code*****//

void bfs(int s,int n)

{

int p,i;

add(s);

vis[s]=1;

p=delete();

if(p!=0)

printf(" %d",p);

while(p!=0)

{

for(i=1;i<=n;i++)

if((a[p][i]!=0)&&(vis[i]==0))
```

```
{  
    add(i);  
    vis[i]=1;  
}  
p=delete();  
if(p!=0)  
    printf(" %d ",p);  
}  
for(i=1;i<=n;i++)  
    if(vis[i]==0)  
        bfs(i,n);  
}
```

```
void add(int item)  
{  
    if(rear==19)  
        printf("QUEUE FULL");  
    else  
    {  
        if(rear==-1)  
        {  
            q[++rear]=item;  
            front++;  
        }  
    }  
    else
```

```
q[++rear]=item;
}
}

int delete()
{
int k;
if((front>rear)|| (front==-1))
return(0);
else
{
k=q[front++];
return(k);
}
}

//*****DFS(depth-first search) code*****//

void dfs(int s,int n)
{
int i,k;
push(s);
vis[s]=1;
k=pop();
if(k!=0)
printf(" %d ",k);
while(k!=0)
{
```

```
for(i=1;i<=n;i++)
if((a[k][i]!=0)&&(vis[i]==0))
{
push(i);
vis[i]=1;
}
k=pop();
if(k!=0)
printf(" %d ",k);
}
for(i=1;i<=n;i++)
if(vis[i]==0)
dfs(i,n);
}
void push(int item)
{
if(top==19)
printf("Stack overflow ");
else
stack[++top]=item;
}
int pop()
{
int k;
if(top==1)
```



```
return(0);  
else  
{  
k=stack[top--];  
return(k);  
}  
}
```

OUTPUT:

RESULT:

Thus the C program implemented DFS and BFS graph traversal.

Ex no: 11A	IMPLEMENT BUBBLE SORT
Date:	

AIM

To write a C program to implement the concept of bubble sort

DESCRIPTION:

- Bubble sort is one of the simplest internal sorting algorithms.
- Bubble sort works by comparing two consecutive elements and the largest element among these two bubbles towards right at the end of the first pass the largest element gets sorted and placed at the end of the sorted list.
- This process is repeated for all pairs of elements until it moves the largest element to the end of the list in that iteration.
- Bubble sort consists of (n-1) passes, where n is the number of elements to be sorted.
- In 1st pass the largest element will be placed in the nth position.
- In 2nd pass the second largest element will be placed in the (n-1)th position.
- In (n-1)th pass only the first two elements are compared.

ALGORITHM:

STEP 1: Start with BubbleSort(arr)

STEP 2: For all array elements Check

```
    if arr[i] > arr[i+1]
        swap(arr[i], arr[i+1])
```

STEP 3: End if

STEP 4: End for and return the array

STEP 5: End BubbleSort

PROGRAM:

```
#include <stdio.h>

int main()
{
    int arr[50], num, x, y, temp;

    printf("Please Enter the Number of Elements you want in the array: ");
    scanf("%d", &num);
    printf("Please Enter the Value of Elements: ");
    for(x = 0; x < num; x++)
        scanf("%d", &arr[x]);
    for(x = 0; x < num - 1; x++){
        for(y = 0; y < num - x - 1; y++){
            if(arr[y] > arr[y + 1]){
                temp = arr[y];
                arr[y] = arr[y + 1];
                arr[y + 1] = temp;
            }
        }
    }
    printf("Array after implementing bubble sort: ");
    for(x = 0; x < num; x++){
        printf("%d ", arr[x]);
    }
    return 0;
}
```

OUTPUT:**RESULT:**

Thus the C program for the concept of bubble sort was implemented successfully

Ex no: 11B	IMPLEMENT SELECTION SORT
Date:	

AIM:

To write a C program to implement the concept of SELECTION sort

ALGORITHM:

Step 1 – Set MIN to location 0

Step 2 – Search the minimum element in the list

Step 3 – Swap with value at location MIN

Step 4 – Increment MIN to point to next element

Step 5 – Repeat until list is sorted

PROGRAM:

```
#include<stdio.h>

int main()
{
    j, count, temp, number[25];

    printf("How many numbers u are going to enter?: ");
    scanf("%d",&count);

    printf("Enter %d elements: ", count);
    // Loop to get the elements stored in array
    for(i=0;i<count;i++)
        scanf("%d",&number[i]);
```

```
// Logic of selection sort algorithm
for(i=0;i<count;i++){
    for(j=i+1;j<count;j++){
        if(number[i]>number[j]){
            temp=number[i];
            number[i]=number[j];
            number[j]=temp;
        }
    }
}
printf("Sorted elements: ");
for(i=0;i<count;i++)
    printf(" %d",number[i]);

return 0;
}
```

OUTPUT:

RESULT:

Thus the C program to implement the concept of selection sort was executed Successfully.

Ex no: 11C	IMPLEMENT INSERTION SORT
Date:	

AIM:

To write a C program to implement the concept of insertion sort

ALGORITHM:

STEP 1: Start insertionSort(array)

STEP 2: Mark first element as sorted, for each unsorted element X,'extract' the element X.

STEP 3: Using for loop check as

forj <- lastSortedIndex down to 0

STEP 4: if current element j > X,then move sorted element to the right by 1 or break loop and insert X here

STEP 5: End insertionSort

PROGRAM:

// Insertion sort in C

#include <stdio.h>

// Function to print an array

void printArray(int array[], int size) {

for (int i = 0; i < size; i++) {

printf("%d ", array[i]);

}

printf("\n");

}

void insertionSort(int array[], int size) {

for (int step = 1; step < size; step++) {

int key = array[step];

int j = step - 1;


```
// Compare key with each element on the left of it until an element smaller than
// it is found.
// For descending order, change key<array[j] to key>array[j].
while (key < array[j] && j >= 0) {
    array[j + 1] = array[j];
    --j;
}
array[j + 1] = key;
}
}

// Driver code
int main() {
    int data[] = {9, 5, 1, 4, 3};
    int size = sizeof(data) / sizeof(data[0]);
    insertionSort(data, size);
    printf("Sorted array in ascending order:\n");
    printArray(data, size);
}
```

OUTPUT:

RESULT:

Thus the C program to implement the concept of insertion sort was executed successfully

Ex no: 12	IMPLEMENT LINEAR SEARCH
Date:	

AIM:

To write a C Program to implement the searching techniques – Linear search and Binary Search.

DESCRIPTION:

Binary search however, cut down your search to half as soon as you find middle of a sorted list. The middle element is looked to check if it is greater than or less than the value to be searched.

Accordingly, search is done to either half of the given list

- Linear Search to find the element “J” in a given sorted list from A-X.
- Binary Search to find the element “J” in a given sorted list from A-X

ALGORITHM:**Linear Search:**

1. Read the search element from the user
2. Compare, the search element with the first element in the list.
3. If both are matching, then display "Given element found!!!" and terminate the function
4. If both are not matching, then compare search element with the next element in the

list.

5. Repeat steps 3 and 4 until the search element is compared with the last element in the list.

6. If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.

Binary search

1. Read the search element from the user

2. Find the middle element in the sorted list

3. Compare, the search element with the middle element in the sorted list.

4. If both are matching, then display "Given element found!!!" and terminate the function

5. If both are not matching, then check whether the search element is smaller or larger than middleelement.

6. If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.

7. If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.

8. Repeat the same process until we find the search element in the list or until sublist contains only one element.

9. If that element also doesn't match with the search element, then display "Element

not found in the list!!!" and terminate the function.

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>

main()
{

    printf("ENTER THE SIZE OF THE ARRAY:");

    scanf("%d",&n);

    printf("ENTER THE ELEMENTS OF THE ARRAY:\n");

    for(i=1;i<=n;i++)
    {

        scanf("%d",&array[i]);

    }

    /* Get the Search Key element for Linear Search */

    printf("ENTER THE SEARCH KEY:");

    scanf("%d",&search_key);

    /* Choice of Search Algorithm */

    printf("_____ \n");

    printf("1.LINEAR SEARCH\n");

    printf("2.BINARY SEARCH\n");

    printf("_____ \n");

    printf("ENTER YOUR CHOICE:");

    scanf("%d",&choice);

    switch(choice)
    {
```

```
case 1:
    linear_search(search_key,array,n);
    break;
case 2:
    binary_search(search_key,array,n);
    break;
default:
    exit(0);
}

getch();
return 0;
}

/* LINEAR SEARCH */

void linear_search(int search_key,int array[100],int n)
{
/*Declare Variable */

    int i,location;

    for(i=1;i<=n;i++)
    {
        if(search_key == array[i])
        {
            location = i;

printf("_____\\n");

printf("The location of Search Key = %d is %d\\n",search_key,location);

printf("_____\\n");

        }
    }
}
```

```
}

/* Binary Search to find Search Key */

void binary_search(int search_key,int array[100],int n)
{
    int mid,i,low,high;

    low = 1;

    high = n;

    mid = (low + high)/2;

    i=1;

    while(search_key != array[mid])
    {
        if(search_key <= array[mid])
        {
            low = 1;

            high = mid+1;

            mid = (low+high)/2;

        }
        else
        {
            low = mid+1;

            high = n;

            mid = (low+high)/2;

        }
    }

    printf("_____\\n");

    printf("location=%d\\t",mid);

    printf("Search_Key=%d Found!\\n",search_key);

    printf("_____\\n");
```

```
}
```

OUTPUT:**RESULT:**

Thus the C program to implement different searching techniques was executed successfully