**Eperiment-2A**

**Aim:-** Search the Song from yours mobile music playlist which are already arranged in the

alphabetical order. If the song is found in the play list, return the position of the song

in the play list.

**Algorithm:-**

Step 1: Set *L* to 0 and *R* to *n* − 1.

Step 2: If *L* > *R*, the search terminates as unsuccessful.

Step 3: Set *m* (the position of the middle element) to the [floor](https://en.wikipedia.org/wiki/Floor_and_ceiling_functions) of (*L* + *R*) / 2.

Step 4: If A*m* < *T*, set L to *m* + 1 and go to step 2.

Step 5: If A*m* > T, set R to *m* – 1 and go to step 2.

Step 6:Now A*m* = *T*, the search is done; return *m*.

**Code-**

#include<stdio.h>

#include<string.h>

int main()

{ int min=0,mid,max=5,pos,i;

char a[5][50],t[50];

for(i=0;i<5;i++)

{

gets(a[i]);

}

printf("enter the string");

gets(t);

while(min<=max)

{

mid=(min+max)/2;

if(strcmp(a[mid],t)==0)

{pos=mid+1;

break;

}

else if(strcmp(t,a[mid])<0)

{

min=mid+1;

}

else if(strcmp(t,a[mid])<0)

{

min=mid+1;

}

}

if(pos>-1)

{

printf("\n %s found at %d",t,pos);

}

else

{

printf("element not found");

}

return 0;

}

**Output:-**

abc

def

ghi

jkl

mno

enter the string ghi

ghi found at 3

**Experiment-:2B**

**Aim:-**To find the element for the array. Find the position of that element for the array and

return its value of its similar value.

**Algorithm:-**

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

**Code:-**

#include<stdio.h>

int main()

{

int a[50],i,t,pos=-1,n;

printf("enter the limit");

scanf("%d",&n);

for(i=0;i<n;i++)

{

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

}

printf("enter no. to search for");

scanf("%d",&t);

for(i=0;i<n;i++)

{

if (a[i]==t)

{

pos=i+1;

break;

}

}

if(pos>-1)

{

printf("%d found at %d",t,pos);

}

else

{

printf("element not found");

}

return 0;

}

**Output:-**

enter the limit5

12

23

43

54

28

enter no. to search for54

54 found at 4

**Experiment:- 3**

**Aim:-**Use the appropriate Sorting Tech. to short the customer based on the type of account

they hold.

Saving Account-Start from the name of the costumer in alphabetical order

Current Account-Start from the date of opening of the account.

**Algorithm:-**

Step 1: Repeat Steps 2 and 3 **for** i=1 to 10

 Step 2: Set j=1

 Step 3: Repeat **while** j<=n

(A) **if**  a[i] < a[j]

  Then interchange a[i] and a[j]

[End of **if**]

(B) Set j = j+1

         [End of Inner Loop]

 [End of Step 1 Outer Loop]

 Step 4: Exit

**Code:-**

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

int main()

{int a[10],m,i,j,d,n,k,l;

char s[5][10],e[10];

printf("enter the no. of current account\n");

scanf("%d",&n);

printf("enter the dates for a month:- \n");

for(i=0;i<n;i++)

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

for(i=0;i<n;i++)

{

for(j=0;j<n-1-i;j++)

{

if(a[j]>a[j+1])

{

d=a[j];

a[j]=a[j+1];

a[j+1]=d;

}

}

}

printf("the arranged date order are:-\n");

for(i=0;i<n;i++)

{ printf("%d\n",a[i]);}

printf("enter the no. of saving accnt. \n");

scanf("%d",&m);

printf("enter the name of the saving accnt holders\n");

for(k=0;k<m;k++)

{scanf("%s",s[k]);}

for(k=0;k<m;k++)

{ for(l=0;l<m-1-k;l++)

{ if(strcmp(s[l],s[l+1])>0)

{ strcpy(e,s[l]);

strcpy(s[l],s[l+1]);

strcpy(s[l+1],e);

}

}

}

printf("the arranged name of saving accnt. holders are \n");

for(k=0;k<m;k++)

{puts(s[k]);}

return 0;

}

**Output:-**

enter the no. of current account

5

enter the dates for a month:-

23

12

4

9

10

the arranged date order are:-

4

9

10

12

23

enter the no. of saving accnt.

5

enter the name of the saving accnt holders

acd

udf

rgt

edf

sdf

the arranged name of saving accnt. holders are

acd

edf

rgt

sdf

udf

**Experiment:-4**

**Aim:-** To create the list of the integers using the singly link list and find the smallest value

from the list.

**Algorithm:-**

[Where ‘head’ pointer has been caught in pointer ‘T’ & the value in ‘key’]

Step 1: [Checking for insertion at beginning and inserting]

if(key <= 0)

Call GETNODE (I)

DATA (I) <-- ‘xyz’

LINK (I) <-- T

T <-- I

return (T).

Step 2: [Traversing to the destination node]

I <-- T

while (DATA (T) != key)

if (LINK(T) == 0)

return (0)

else

T <-- LINK(T))

Step 3: [Inserting the node in the list]

if (LINK(T) == 0)

GETNODE (P)

LINK (P) <-- 0

LINK (T) <-- P

DATA (P) <-- ‘xyz’

return (I)

else

GETNODE (P)

C <-- LINK (LINK(T))

LINK (P) <-- C

LINK (T) <-- P

DATA (P) <-- ‘xyz’

return (I)

Step 4: [FINISH]

return.

**Code:-**

#include<stdio.h>

#include<stdlib.h>

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

struct node

{

int data;

struct node \*next;

}\*start=NULL;

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

void create()

{

char ch;

do

{

struct node \*new\_node,\*current;

new\_node=(struct node \*)malloc(sizeof(struct node));

printf("nEnter the data : ");

scanf("%d",&new\_node->data);

new\_node->next=NULL;

if(start==NULL)

{

start=new\_node;

current=new\_node;

}

else

{

current->next=new\_node;

current=new\_node;

}

printf("nDo you want to creat another : ");

scanf("%s",&ch);

}while(ch!='n');

}

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

void display()

{

struct node \*new\_node;

printf("The Linked List : n");

new\_node=start;

while(new\_node!=NULL)

{

printf("%d--->",new\_node->data);

new\_node=new\_node->next;

}

printf("NULL");

}

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

void m()

{ struct node \*min,\*max;

// start at the root

struct node \*currentNode = start;

if (currentNode == NULL)

printf("list is empty\n");

else{ // initialize the max and min values to the first node

max = min = currentNode;

// loop through the list

for (currentNode = currentNode->next; currentNode != NULL; currentNode = currentNode->next)

{

if (currentNode->data > max->data)

max = currentNode;

else if (currentNode->data < min->data)

min= currentNode;

}

printf("%d is the minimum value and %d is the maximum value\n",min->data,max->data);

}}

int main()

{

create();

display();

m();

return 0;}

**Output:-**

Enter the data : 5

Do you want to creat another : y

Enter the data : 4

Do you want to creat another : y

Enter the data : 9

Do you want to creat another : y

Enter the data : 1

Do you want to creat another : y

Enter the data : 3

Do you want to creat another : y

Enter the data : 56

Do you want to creat another : n

The Linked List : n5--->4--->9--->1--->3--->56--->NULL

1 is the minimum value and 56 is the maximum value

**Experiment :- 5**

**Aim:-** To perform all the operations in the singly link list.

**Algorithm:-**

[Where ‘head’ pointer has been caught in pointer ‘T’ & the value in ‘key’]

Step 1: [Checking for insertion at beginning and inserting]

if(key <= 0)

Call GETNODE (I)

DATA (I) <-- ‘xyz’

LINK (I) <-- T

T <-- I

return (T).

Step 2: [Traversing to the destination node]

I <-- T

while (DATA (T) != key)

if (LINK(T) == 0)

return (0)

else

T <-- LINK(T))

Step 3: [Inserting the node in the list]

if (LINK(T) == 0)

GETNODE (P)

LINK (P) <-- 0

LINK (T) <-- P

DATA (P) <-- ‘xyz’

return (I)

else

GETNODE (P)

C <-- LINK (LINK(T))

LINK (P) <-- C

LINK (T) <-- P

DATA (P) <-- ‘xyz’

return (I)

Step 4: [FINISH]

return.

**Code:-**

include<stdio.h>

#include<stdlib.h>

**struct** node

{

**int** data;

**struct** node \*next;

}\*head;

**void** append(**int** num)

{

**struct** node \*temp,\*right;

    temp= (**struct** node \*)**malloc**(**sizeof**(**struct** node));

    temp->data=num;

    right=(**struct** node \*)head;

**while**(right->next != NULL)

    right=right->next;

    right->next =temp;

    right=temp;

    right->next=NULL;

}

**void** add( **int** num )

{

**struct** node \*temp;

    temp=(**struct** node \*)**malloc**(**sizeof**(**struct** node));

    temp->data=num;

**if** (head== NULL)

    {

    head=temp;

    head->next=NULL;

    }

**else**

    {

    temp->next=head;

    head=temp;

    }

}

**void** addafter(**int** num, **int** loc)

{

**int** i;

**struct** node \*temp,\*left,\*right;

    right=head;

**for**(i=1;i<loc;i++)

    {

    left=right;

    right=right->next;

    }

    temp=(**struct** node \*)**malloc**(**sizeof**(**struct** node));

    temp->data=num;

    left->next=temp;

    left=temp;

    left->next=right;

**return**;

}

**void** insert(**int** num)

{

**int** c=0;

**struct** node \*temp;

    temp=head;

**if**(temp==NULL)

    {

    add(num);

    }

**else**

    {

**while**(temp!=NULL)

    {

**if**(temp->data<num)

        c++;

        temp=temp->next;

    }

**if**(c==0)

        add(num);

**else** **if**(c<count())

        addafter(num,++c);

**else**

        append(num);

    }

}

**int** **delete**(**int** num)

{

**struct** node \*temp, \*prev;

    temp=head;

**while**(temp!=NULL)

    {

**if**(temp->data==num)

    {

**if**(temp==head)

        {

        head=temp->next;

**free**(temp);

**return** 1;

        }

**else**

        {

        prev->next=temp->next;

**free**(temp);

**return** 1;

        }

    }

**else**

    {

        prev=temp;

        temp= temp->next;

    }

    }

**return** 0;

}

**void**  display(**struct** node \*r)

{

    r=head;

**if**(r==NULL)

    {

**return**;

    }

**while**(r!=NULL)

    {

**printf**("%d ",r->data);

    r=r->next;

    }

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

}

**int** count()

{

**struct** node \*n;

**int** c=0;

    n=head;

**while**(n!=NULL)

    {

    n=n->next;

    c++;

    }

**return** c;

}

**int**  main()

{

**int** i,num;

**struct** node \*n;

    head=NULL;

**while**(1)

    {

**printf**("\nList Operations\n");

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

**printf**("1.Insert\n");

**printf**("2.Display\n");

**printf**("3.Size\n");

**printf**("4.Delete\n");

**printf**("5.Exit\n");

**printf**("Enter your choice : ");

**if**(**scanf**("%d",&i)<=0){

**printf**("Enter only an Integer\n");

**exit**(0);

    } **else** {

**switch**(i)

        {

**case** 1:      **printf**("Enter the number to insert : ");

**scanf**("%d",&num);

                 insert(num);

**break**;

**case** 2:     **if**(head==NULL)

                {

**printf**("List is Empty\n");

                }

**else**

                {

**printf**("Element(s) in the list are : ");

                }

                display(n);

**break**;

**case** 3:     **printf**("Size of the list is %d\n",count());

**break**;

**case** 4:     **if**(head==NULL)

**printf**("List is Empty\n");

**else**{

**printf**("Enter the number to delete : ");

**scanf**("%d",&num);

**if**(**delete**(num))

**printf**("%d deleted successfully\n",num);

**else**

**printf**("%d not found in the list\n",num);

                }

**break**;

**case** 5:     **return** 0;

**default**:    **printf**("Invalid option\n");

        }

    }

    }

**return** 0;

}

**Output:-**

List Operations

===============

1.Insert

2.Display

3.Size

4.Delete

5.Exit

Enter your choice : 1

Enter the number to insert : 12

List Operations

===============

1.Insert

2.Display

3.Size

4.Delete

5.Exit

Enter your choice : 1

Enter the number to insert : 23

List Operations

===============

1.Insert

2.Display

3.Size

4.Delete

5.Exit

Enter your choice : 1

Enter the number to insert : 34

List Operations

===============

1.Insert

2.Display

3.Size

4.Delete

5.Exit

Enter your choice : 1

Enter the number to insert : 45

List Operations

===============

1.Insert

2.Display

3.Size

4.Delete

5.Exit

Enter your choice : 1

Enter the number to insert : 56

List Operations

===============

1.Insert

2.Display

3.Size

4.Delete

5.Exit

Enter your choice : 2

Element(s) in the list are : 12 23 34 45 56

List Operations

===============

1.Insert

2.Display

3.Size

4.Delete

5.Exit

Enter your choice : 4

Enter the number to delete : 12

12 deleted successfully

List Operations

===============

1.Insert

2.Display

3.Size

4.Delete

5.Exit

Enter your choice : 4

Enter the number to delete : 23

23 deleted successfully

List Operations

===============

1.Insert

2.Display

3.Size

4.Delete

5.Exit

Enter your choice : 2

Element(s) in the list are : 34 45 56

List Operations

===============

1.Insert

2.Display

3.Size

4.Delete

5.Exit

Enter your choice : 5

**Experiment:-6**

**Aim:-** To create the doubly link list and find the data in the node having the smallest value.

**Algorithm:-**

InsertAtBegDll(info,prev,next,start,end)

Step 1: create a new node and address in assigned to ptr.

Step 2: check[overflow] if(ptr=NULL)

write:overflow and exit

Step 3: set Info[ptr]=item;

Step 4: if(start=NULL)

set prev[ptr] = next[ptr] = NULL

set start = end = ptr

else

set prev[ptr] = NULL

next[ptr] = start

set prev[start] = ptr

set start = ptr

[end if]

Step 5: Exit.

**Code:-**

#include <stdio.h>

#include <stdlib.h>

struct node

{

struct node \*prev;

int n;

struct node \*next;

}\*h,\*temp,\*temp1,\*temp2,\*temp4;

void small();

void insert1();

void insert2();

void insert3();

void traversebeg();

void traverseend(int);

void sort();

void search();

void update();

void delete();

int count = 0;

int main()

{

int ch;

h = NULL;

temp = temp1 = NULL;

printf("\n 1 - Insert at beginning");

printf("\n 2 - Insert at end");

printf("\n 3 - Insert at position i");

printf("\n 4 - Delete at i");

printf("\n 5 - Display from end");

printf("\n 6 - Display from begnning");

printf("\n 7 - Search for element");

printf("\n 8 - Sort the list");

printf("\n 9 - Update an element\n 10 - Find the smallest number ");

printf("\n 11 - Exit");

while (1)

{

printf("\n Enter choice : ");

scanf("%d", &ch);

switch (ch)

{

case 1:

insert1();

break;

case 2:

insert2();

break;

case 3:

insert3();

break;

case 4:

delete();

break;

case 5:

traversebeg();

break;

case 6:

temp2 = h;

if (temp2 == NULL)

printf("\n Error : List empty to display ");

else

{

printf("\n Reverse order of linked list is : ");

traverseend(temp2->n);

}

break;

case 7:

search();

break;

case 8:

sort();

break;

case 9:

update();

break;

case 10:

small();

break;

case 11:

exit(0);

default:

printf("\n Wrong choice menu");

}

}

return 0;

}

/\* TO create an empty node \*/

void create()

{

int data;

temp =(struct node \*)malloc(1\*sizeof(struct node));

temp->prev = NULL;

temp->next = NULL;

printf("\n Enter value to node : ");

scanf("%d", &data);

temp->n = data;

count++;

}

/\* TO insert at beginning \*/

void insert1()

{

if (h == NULL)

{

create();

h = temp;

temp1 = h;

}

else

{

create();

temp->next = h;

h->prev = temp;

h = temp;

}

}

/\* To insert at end \*/

void insert2()

{

if (h == NULL)

{

create();

h = temp;

temp1 = h;

}

else

{

create();

temp1->next = temp;

temp->prev = temp1;

temp1 = temp;

}

}

/\* To insert at any position \*/

void insert3()

{

int pos, i = 2;

printf("\n Enter position to be inserted : ");

scanf("%d", &pos);

temp2 = h;

if ((pos < 1) || (pos >= count + 1))

{

printf("\n Position out of range to insert");

return;

}

if ((h == NULL) && (pos != 1))

{

printf("\n Empty list cannot insert other than 1st position");

return;

}

if ((h == NULL) && (pos == 1))

{

create();

h = temp;

temp1 = h;

return;

}

else

{

while (i < pos)

{

temp2 = temp2->next;

i++;

}

create();

temp->prev = temp2;

temp->next = temp2->next;

temp2->next->prev = temp;

temp2->next = temp;

}

}

/\* To delete an element \*/

void delete()

{

int i = 1, pos;

printf("\n Enter position to be deleted : ");

scanf("%d", &pos);

temp2 = h;

if ((pos < 1) || (pos >= count + 1))

{

printf("\n Error : Position out of range to delete");

return;

}

if (h == NULL)

{

printf("\n Error : Empty list no elements to delete");

return;

}

else

{

while (i < pos)

{

temp2 = temp2->next;

i++;

}

if (i == 1)

{

if (temp2->next == NULL)

{

printf("Node deleted from list");

free(temp2);

temp2 = h = NULL;

return;

}

}

if (temp2->next == NULL)

{

temp2->prev->next = NULL;

free(temp2);

printf("Node deleted from list");

return;

}

temp2->next->prev = temp2->prev;

if (i != 1)

temp2->prev->next = temp2->next; /\* Might not need this statement if i == 1 check \*/

if (i == 1)

h = temp2->next;

printf("\n Node deleted");

free(temp2);

}

count--;

}

/\* Traverse from beginning \*/

void traversebeg()

{

temp2 = h;

if (temp2 == NULL)

{

printf("List empty to display \n");

return;

}

printf("\n Linked list elements from begining : ");

while (temp2->next != NULL)

{

printf(" %d ", temp2->n);

temp2 = temp2->next;

}

printf(" %d ", temp2->n);

}

/\* To traverse from end recursively \*/

void traverseend(int i)

{

if (temp2 != NULL)

{

i = temp2->n;

temp2 = temp2->next;

traverseend(i);

printf(" %d ", i);

}

}

/\* To search for an element in the list \*/

void search()

{

int data, count = 0;

temp2 = h;

if (temp2 == NULL)

{

printf("\n Error : List empty to search for data");

return;

}

printf("\n Enter value to search : ");

scanf("%d", &data);

while (temp2 != NULL)

{

if (temp2->n == data)

{

printf("\n Data found in %d position",count + 1);

return;

}

else

temp2 = temp2->next;

count++;

}

printf("\n Error : %d not found in list", data);

}

/\* To update a node value in the list \*/

void update()

{

int data, data1;

printf("\n Enter node data to be updated : ");

scanf("%d", &data);

printf("\n Enter new data : ");

scanf("%d", &data1);

temp2 = h;

if (temp2 == NULL)

{

printf("\n Error : List empty no node to update");

return;

}

while (temp2 != NULL)

{

if (temp2->n == data)

{

temp2->n = data1;

traversebeg();

return;

}

else

temp2 = temp2->next;

}

printf("\n Error : %d not found in list to update", data);

}

/\* To sort the linked list \*/

void sort()

{

int i, j, x;

temp2 = h;

temp4 = h;

if (temp2 == NULL)

{

printf("\n List empty to sort");

return;

}

for (temp2 = h; temp2 != NULL; temp2 = temp2->next)

for (temp4 = temp2->next; temp4 != NULL; temp4 = temp4->next)

{

if (temp2->n > temp4->n)

{

x = temp2->n;

temp2->n = temp4->n;

temp4->n = x;

}

}

}

traversebeg();

}

// for finding smallest number.

void small()

{

int min;

struct node \*r,\*p,\*ne;

r=h;

min = r->n;

while(r!=NULL)

{

if(min>r->n)

{

min=r->n;

}

r=r->next;

}

printf("\n smallest no. is %d",min);

/\* p = r->prev;

printf("\nprev->info %d,p->next %p",p->n,p->next);

ne=r->next;

printf("\nne->next %p",ne->next);

p->next=ne;

printf("\np->next %p",p->next);

if(ne!=NULL)

{

ne->prev=p;

}\*/

}

**Output**:-

1 - Insert at beginning

2 - Insert at end

3 - Insert at position i

4 - Delete at i

5 - Display from end

6 - Display from begnning

7 - Search for element

8 - Sort the list

9 - Update an element

10 - Find the smallest number

11 - Exit

Enter choice : 1

Enter value to node : 12

Enter choice : 1

Enter value to node : 23

Enter choice : 1

Enter value to node : 34

Enter choice : 1

Enter value to node : 45

Enter choice : 1

Enter value to node : 67

Enter choice : 6

Reverse order of linked list is : 12 23 34 45 67

Enter choice : 10

smallest no. is 12

Enter choice : 11

**Experiment:-7[ Stacks using array]**

**Aim:-** To check the parenthesis balanced or not in the given expression using the stack

concept.

**Algorithm:-**

Step 1: Whenever we see a opening parenthesis, we put it on stack.

Step 2: For closing parenthesis, check what is at the top of the stack, if it corresponding

opening parenthesis, remove it from the top of the stack.

Step 3: If parenthesis at the top of the stack is not corresponding opening parenthesis, return

false, as there is no point check the string further.

Step 5: After processing entire string, check if stack is empty or not.

Step 4A: If the stack is empty, return true.

Step 4B: If stack is not empty, parenthesis do not match.

**Code:-**

#include<stdio.h>

#define MAX 30

int top=-1;

int stack[MAX];

void push(char);

char pop();

int match(char a,char b);

int main()

{

char exp[MAX];

int valid;

printf("Enter an algebraic expression : ");

gets(exp);

valid=check(exp);

if(valid==1)

printf("Valid expression\n");

else

printf("Invalid expression\n");

return 0;

}

int check(char exp[] )

{

int i;

char temp;

for(i=0;i<strlen(exp);i++)

{

if(exp[i]=='(' || exp[i]=='{' || exp[i]=='[')

push(exp[i]);

if(exp[i]==')' || exp[i]=='}' || exp[i]==']')

if(top==-1) /\*stack empty\*/

{

printf("Right parentheses are more than left parentheses\n");

return 0;

}

else

{

temp=pop();

if(!match(temp, exp[i]))

{

printf("Mismatched parentheses are : ");

printf("%c and %c\n",temp,exp[i]);

return 0;

}

}

}

if(top==-1) /\*stack empty\*/

{

printf("Balanced Parentheses\n");

return 1;

}

else

{

printf("Left parentheses more than right parentheses\n");

return 0;

}

}/\*End of main()\*/

int match(char a,char b)

{

if(a=='[' && b==']')

return 1;

if(a=='{' && b=='}')

return 1;

if(a=='(' && b==')')

return 1;

return 0;

}/\*End of match()\*/

void push(char item)

{

if(top==(MAX-1))

{

printf("Stack Overflow\n");

return;

}

top=top+1;

stack[top]=item;

}/\*End of push()\*/

char pop()

{

if(top==-1)

{

printf("Stack Underflow\n");

exit(1);

}

return(stack[top--]);

}/\*End of pop()\*/

**Output:-**

Enter an algebraic expression : [{a+b}\*(d-c)]

Balanced Parentheses

Valid expression

Enter an algebraic expression : {(a+b)\*[f-g]]

Mismatched parentheses are : { and ]

Invalid expression

**Experiment:-8 [Stacks using link list]**

**Aim:-** To check the parenthesis balanced or not in the given expression using the stack

concept.

**Algorithm:-**

Step 1: Whenever we see a opening parenthesis, we put it on stack.

Step 2: For closing parenthesis, check what is at the top of the stack, if it corresponding

opening parenthesis, remove it from the top of the stack.

Step 3: If parenthesis at the top of the stack is not corresponding opening parenthesis, return

false, as there is no point check the string further.

Step 5: After processing entire string, check if stack is empty or not.

Step 4A: If the stack is empty, return true.

Step 4B: If stack is not empty, parenthesis do not match.

**Code:-**

#include <stdio.h>

#include <stdlib.h>

#define TRUE 1

#define FALSE 0

#define MAX 128

typedef struct \_stack

{

char \*s;

int top;

} stack;

void stack\_push (stack \* s, char c);

char stack\_pop (stack \* s);

int stack\_is\_empty (stack \* s);

stack \*stack\_create (int n);

void stack\_delete (stack \*stk);

int check\_balanced\_paranthesis (char \*exp, int \*error\_at);

int

main (void)

{

char exp[MAX] = "";

int retval, error\_at;

printf ("\n\nCheck for balanced parenthesized equation.");

printf ("\nEnter paranthesised string :\n");

scanf (" %[^\n]", exp);

retval = check\_balanced\_paranthesis (exp, &error\_at);

if (retval == TRUE)

printf ("\nParanthesis are UNBALANCED.\nFirst Error at location %d\n",

error\_at);

else

printf ("\nParanthesis are BALANCED\n");

return 0;

}

/\* Accepts an expression exp to check for balanced parenthesis,

\* other symbols are ignored. The error\_at will store the position

\* where the first error was encountered. If NULL is passed in

\* error\_at, it will not be considered. In case of an error a

\* nonzero values is returned, in case of a correct expression a 0

\* is returned.

\*/

int

check\_balanced\_paranthesis (char \*exp, int \*error\_at)

{

char token = '\0', temp;

int error = FALSE, i = 0;

stack \*stk;

stk = stack\_create (MAX);

while (exp[i] != '\0')

{

token = exp[i];

if (token == '(' || token == '{' || token == '[')

stack\_push (stk, token);

else if (token == ')' || token == '}' || token == ']')

{

if (stack\_is\_empty (stk))

{

error = TRUE;

break;

}

temp = stack\_pop (stk);

if (!((temp == '(' && token == ')') ||

(temp == '{' && token == '}') ||

(temp == '[' && token == ']')))

{

error = TRUE;

break;

}

}

i++;

}

if (!stack\_is\_empty (stk))

error = TRUE;

if ((error == TRUE) && (error\_at != NULL))

\*error\_at = i + 1;

stack\_delete (stk);

return error;

}

/\* STACK ROUTINES START \*/

void

stack\_push (stack \* stk, char token)

{

if (stk->top >= MAX)

{

printf ("\nStack FULL Terminating");

exit (1);

}

stk->s[++(stk->top)] = token;

}

char

stack\_pop (stack \* stk)

{

if (stk->top <= -1)

return '\0';

return stk->s[(stk->top)--];

}

int

stack\_is\_empty (stack \* stk)

{

if (stk->top <= -1)

return TRUE;

return FALSE;

}

stack \*

stack\_create (int n)

{

stack \*temp;

temp = malloc (sizeof (stack));

if (temp == NULL)

{

perror ("Terminating\n");

exit (1);

}

temp->s = malloc (sizeof (char) \* n);

if (temp->s == NULL)

{

perror ("Terminating\n");

exit (1);

}

temp->top = -1;

return temp;

}

void

stack\_delete (stack \*stk)

{

if (stk != NULL)

{

if (stk->s)

free (stk->s);

free (stk);

}

}

**Output:-**

Check for balanced parenthesized equation.

Enter paranthesised string :

[(a+b)-{c\*g}]

Paranthesis are BALANCED

Check for balanced parenthesized equation.

Enter paranthesised string :

[{a-b)\*(d+c)]

Paranthesis are UNBALANCED.

First Error at location 6

**Experiment:- 9 [Queue]**

**Aim:-** Simulate the phone answering system with a help of an appropriate data structure.

**Algorithm:-**

Step 1 − Check if queue is full.

Step 2 − If queue is full, produce overflow error

and exit.

Step 3 − If queue is not full, increment rear pointer to point next empty space.

Step 4 − Add data element to the queue location, where rear is pointing.

Step 5 − return success.

**Code:-**

#include <stdio.h>

#define MAX 50

struct no

{

int ph\_no;

int time;

int wait;

};

void insert(struct no s[]);

void calculate(struct no s[]);

int front=-1;

int rear=-1;

int queue\_array[MAX];

int main()

{

int ch;

struct no s[10];

while(1)

{

printf("Enter your's choice:\n\n");

printf("<1> Recent answered calling no.\n<2> Display\n<3> Exit\n\n");

scanf("%d",&ch);

switch(ch)

{

case 1:

insert(s);

printf("\n%d\n",s[0].time);

break;

case 2:

calculate(s);

break;

case 3:

return 0;

default:

printf("\n\nEntered choice is wrong\n\n");

}

}

return 0;

}

void insert(struct no s[])

{ int no,time;

if(rear==MAX-1)

{

printf("\nList is Full\n\n");

}

else

{

if(front==-1)

front=0;

printf("\n\nEnter the phone no. with total time of calling\n\n");

scanf("%d",&no);

scanf("%d",&time);

rear=rear+1;

s[rear].ph\_no=no;

s[rear].time=time;

}

}

void calculate(struct no s[])

{

// printf("\nInside calculate");

int i=0;

s[0].wait=0;

for(i=front+1;i<=rear;i++)

{

//printf("\ninside loop1");

s[i].wait=s[i].time-s[i-1].time;

}

for(i=front;i<=rear;i++)

{

printf("\n%d\t",s[i].ph\_no);

printf("\t%d\t",s[i].time);

printf("\t%d\n",s[i].wait);

}

}

**Output:-**

Enter your's choice:

<1> Recent answered calling no.

<2> Display

<3> Exit

1

Enter the phone no. with total time of calling

123

5

5

Enter your's choice:

<1> Recent answered calling no.

<2> Display

<3> Exit

1

Enter the phone no. with total time of calling

234

10

5

Enter your's choice:

<1> Recent answered calling no.

<2> Display

<3> Exit

1

Enter the phone no. with total time of calling

345

15

5

Enter your's choice:

<1> Recent answered calling no.

<2> Display

<3> Exit

1

Enter the phone no. with total time of calling

456

20

5

Enter your's choice:

<1> Recent answered calling no.

<2> Display

<3> Exit

1

Enter the phone no. with total time of calling

567

25

5

Enter your's choice:

<1> Recent answered calling no.

<2> Display

<3> Exit

2

123 5 0

234 10 5

345 15 5

456 20 5

567 25 5

**Experiment:- 10A & 10B**

**Aim:-** To implement the tree concept and transverse of the tree.

**Algorithm:-**

**Pre-order**

1. Check if the current node is empty / null
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order function.
4. Traverse the right subtree by recursively calling the pre-order function.

**In-order**

1. Check if the current node is empty / null
2. Traverse the left subtree by recursively calling the in-order function.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order function.

In a [search tree](https://en.wikipedia.org/wiki/Search_tree), in-order traversal retrieves data in sorted order.[4]

**Post-order**

1. Check if the current node is empty / null
2. Traverse the left subtree by recursively calling the post-order function.
3. Traverse the right subtree by recursively calling the post-order function.

Display the data part of the root (or current node).

**Code:-**

#include <stdio.h>

#include <stdlib.h>

struct node

{

int value;

struct node\* left;

struct node\* right;

};

struct node\* root;

struct node\* insert(struct node\* r, int data);

void inOrder(struct node\* r);

void preOrder(struct node\* r);

void postOrder(struct node\* r);

int main()

{

root = NULL;

int n, v;

printf("How many data's do you want to insert ?\n");

scanf("%d", &n);

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

printf("Data %d: ", i+1);

scanf("%d", &v);

root = insert(root, v);

}

printf("Inorder Traversal: ");

inOrder(root);

printf("\n");

printf("Preorder Traversal: ");

preOrder(root);

printf("\n");

printf("Postorder Traversal: ");

postOrder(root);

printf("\n");

return 0;

}

struct node\* insert(struct node\* r, int data)

{

if(r==NULL)

{

r = (struct node\*) malloc(sizeof(struct node));

r->value = data;

r->left = NULL;

r->right = NULL;

}

else if(data < r->value){

r->left = insert(r->left, data);

}

else {

r->right = insert(r->right, data);

}

return r;

}

void inOrder(struct node\* r)

{

if(r!=NULL){

inOrder(r->left);

printf("%d ", r->value);

inOrder(r->right);

}

}

void preOrder(struct node\* r)

{

if(r!=NULL){

printf("%d ", r->value);

preOrder(r->left);

preOrder(r->right);

}

}

void postOrder(struct node\* r)

{

if(r!=NULL){

postOrder(r->left);

postOrder(r->right);

printf("%d ", r->value);

}

}

**Output:-**

How many data's do you want to insert ?

5

Data 1: 4

Data 2: 1

Data 3: 2

Data 4: 7

Data 5: 0

Inorder Traversal: 0 1 2 4 7

Preorder Traversal: 4 1 0 2 7

Postorder Traversal: 0 2 1 7 4

**Experiment:- 11**

**Aim:-**  To Implement the prim’s algorithm.

**Algorithm:-**

Step 1: Associate with each vertex *v* of the graph a number *C*[*v*] (the cheapest cost of a connection to *v*) and an edge *E*[*v*] (the edge providing that cheapest connection). To initialize these values, set all values of *C*[*v*] to +∞ (or to any number larger than the maximum edge weight) and set each *E*[*v*] to a special [flag value](https://en.wikipedia.org/wiki/Flag_value) indicating that there is no edge connecting *v* to earlier vertices.

Step 2: Initialize an empty forest *F* and a set *Q* of vertices that have not yet been included in *F* (initially, all vertices).

Step 3: Repeat the following steps until *Q* is empty:

* 1. Find and remove a vertex *v* from *Q* having the minimum possible value of *C*[*v*]
  2. Add *v* to *F* and, if *E*[*v*] is not the special flag value, also add *E*[*v*] to *F*
  3. Loop over the edges *vw* connecting *v* to other vertices *w*. For each such edge, if *w* still belongs to *Q* and *vw* has smaller weight than *C*[*w*], perform the following steps:
     1. Set *C*[*w*] to the cost of edge *vw*
     2. Set *E*[*w*] to point to edge *vw*.

Step 4: Return *F*

**Code:-**

#include<stdio.h>

int a,b,u,v,n,i,j,ne=1;

int visited[10]={0},min,mincost=0,cost[10][10];

int main()

{

printf("\nEnter the number of nodes:");

scanf("%d",&n);

printf("\nEnter the adjacency matrix:\n");

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

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

{

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

if(cost[i][j]==0)

cost[i][j]=999;

}

visited[1]=1;

printf("\n");

while(ne < n)

{

for(i=1,min=999;i<=n;i++)

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

if(cost[i][j]< min)

if(visited[i]!=0)

{

min=cost[i][j];

a=u=i;

b=v=j;

}

if(visited[u]==0 || visited[v]==0)

{

printf("\n Edge %d:(%d %d) cost:%d",ne++,a,b,min);

mincost+=min;

visited[b]=1;

}

cost[a][b]=cost[b][a]=999;

}

printf("\n Minimun cost=%d",mincost);

return 0;

}

**Output:-**

Enter the number of nodes:3

Enter the adjacency matrix:

1 2 3

4 5 6

7 8 9

Edge 1:(1 2) cost:2

Edge 2:(1 3) cost:3

Minimun cost=5