**INFANT JESUS COLLEGE OF ENGINEERING**

**KAMARAJAR NAGAR**

**KEELAVALLANADU-628 851**

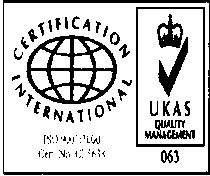
.



**CS8381 –DATA STRUCTURES LABORATORY**

NAME : ------------------------------------------------------------------------------------------

REGISTER NUMBER : -------------------------------------------------------------------------------------------

**INFANT JESUS COLLEGE OF ENGINEERING**

**KAMARAJAR NAGAR**

**KEELAVALLANADU-628 851**

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NAME : ------------------------------------------------------------------------------------------

REGISTER NUMBER : ---------------------------------------------------------------ROLL NO:--------------

BRANCH : ------------------------------------------------------------------------------------------

YEAR : ------------------------------------------------------------------------------------------

Bonafede Record of work done in the **CS8381 – Data Structures Laboratory** at Infant Jesus College of Engineering, Keelavallanadu during the year

**STAFF INCHARGE HEAD OF THE DEPARTMENT**

Submitted for the Practical Examination held on ----------------------------------------------------------

at Infant Jesus College of Engineering, Keelavallanadu.

**INTERNAL EXAMINER EXTERNAL EXAMINER**

**Ex. No.: 1a Stack Array**

**Date :**

**Aim:**

To implement stack operations using array.

**Algorithm:**

1. Star

2. Define a array stack of size max = 5

3. Initialize top = -1

4. Display a menu listing stack operation

5. Accept choice

6. If choice = 1 then

If top < max -1

Increment top

Store element at current position of top

Else

Print Stack overflow

Else If choice = 2 then

If top < 0 then

Print Stack underflow

Else

Display current top element

Decrement top

Else If choice = 3 then

Display stack elements starting from top

7. Stop

**Program:**

/\* Stack Operation using Arrays \*/

#include <stdio.h>

#include <stdlib.h>

#define max 5

static int stack[max];

int top = -1;

void push(int x)

{

stack[++top] = x;

}

int pop()

{

return (stack[top--]);

}

void view()

{

int i;

if (top < 0)

printf("\n Stack Empty \n");

else

{

printf("\n Top-->");

for(i=top; i>=0; i--)

{

printf("%4d", stack[i]);

}

printf("\n");

}

}

void main()

{

int ch=0, val;

system("clear");

while(ch != 4)

{

printf("\n STACK OPERATION \n");

printf("1.PUSH ");

printf("2.POP ");

printf("3.VIEW ");

printf("4.QUIT \n");

printf("Enter Choice : ");

scanf("%d", &ch);

switch(ch)

{

case 1:

if(top < max-1)

{

printf("\nEnter Stack element : ");

scanf("%d", &val);

push(val);

}

else

printf("\n Stack Overflow \n");

break;

case 2:

if(top < 0)

printf("\n Stack Underflow \n");

else

{

val = pop();

printf("\n Popped element is %d\n", val);

}

break;

case 3:

view();

break;

case 4:

exit(0);

default:

printf("\n Invalid Choice \n");

}

}

}

**Output:**

**Result:**

Thus, push and pop operations of a stack was demonstrated using arrays.

**Ex. No. 1b Queue Array**

**Date:**

**Aim:**

To implement queue operations using array.

**Algorithm:**

1. Start

2. Define a array queue of size max = 5

3. Initialize front = rear = –1

4. Display a menu listing queue operations

5. Accept choice

6. If choice = 1 then

If rear < max -1

Increment rear Store element at current position of rear

Else

Print Queue Full

Else If choice = 2 then

If front = –1 then

Print Queue empty

Else

Display current front element

Increment front

Else If choice = 3 then

Display queue elements starting from front to rear.

7. Stop

**Program**

#include<stdio.h>

#include<stdlib.h>

#define SIZE 5

int front = - 1;

int rear = - 1;

int q[SIZE];

void insert( );

void del( );

void display( );

void main( )

{

int choice;

do

{

printf("\n\t Menu");

printf("\n 1. Insert");

printf("\n 2. Delete");

printf("\n 3. Display ");

printf("\n 4. Exit");

printf("\n Enter Your Choice:");

scanf("%d", &choice);

switch(choice)

{

case 1:

insert( );

display( );

break;

case 2:

del( );

display( );

break;

case 3:

display();

break;

case 4:

printf("End of Program....!!!!\n");

exit(0);

}

}while(choice != 4);

}

void insert( )

{

int no;

printf("\n Enter No.:");

scanf("%d", &no);

if(rear < SIZE - 1)

{

q[++rear]=no;

if(front == -1)

front=0;// front=front+1;

}

else

{

printf("\n Queue overflow");

}

}

void del( )

{

if(front == - 1)

{

printf("\n Queue Underflow");

return;

}

else

{

printf("\n Deleted Item:-->%d\n", q[front]);

}

if(front == rear)

{

front = - 1;

rear = - 1;

}

else

{

front = front + 1;

}

}

void display( )

{

int i;

if( front == - 1)

{

printf("\nQueue is empty....");

return;

}

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

printf("\t%d",q[i]);

}

**Output:**

**Result**

Thus, insert and delete operations of a queue was demonstrated using arrays.

**Ex. No. 2 List using Array**

**Date:**

**Aim**

To perform various operations on List ADT using array implementation.

**Algorithm**

1. Start

2. Create a list of n elements

3. Display list operations as a menu

4. Accept user choice

5. If choice = 1 then

Get position of element to be deleted

Move elements one position upwards thereon.

Decrement length of the list

Else if choice = 2

Get position of element to be inserted.

Increment length of the list

Move elements one position downwards thereon

Store the new element in corresponding position

Else if choice = 3

Traverse the list and inspect each element

Report position if it exists.

6. Stop

**Program**

|  |
| --- |
|  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

/\* List operation using Arrays \*/

#include <stdio.h>

#include <stdlib.h>

void create();

void insert();

void search();

void deletion();

void display();

int i, e, n, pos;

static int b[50];

void main()

{

int ch;

char g = 'y';

create();

do

{

printf("\n List Operations");

printf("\n 1.Deletion\n 2.Insert\n 3.Search\n4.Exit\n");

printf("Enter your choice: ");

scanf("%d", &ch);

switch(ch)

{

case 1:

printf("\n");

deletion();

break;

case 2:

printf("\n");

insert();

break;

case 3:

printf("\n");

//printf("\nEnter the element to be searched: ");

search();

break;

case 4:

exit(0);

default:

printf("\nEnter the correct choice:");

}

printf("\n");

printf("Do you want to continue: ");

fflush(stdin);

scanf("\n %c",&g);

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

}

void create()

{

printf("\n Enter the number of elements:");

scanf("%d",&n);

printf("\n Enter list elements: ");

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

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

}

void deletion()

{

printf("\n enter the position you 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("List elements after deletion");

display();

}

}

void search()

{

int flag = 0;

printf("\nEnter the element to be searched: ");

scanf("%d", &e);

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

{

if(b[i] == e)

{

flag = 1;

printf("Element is in the %d position", i);

break;

}

}

if(flag == 0)

printf("Value %d is not in the list", e);

}

void insert()

{

printf("\n Enter the position you need to insert: ");

scanf("%d", &pos);

if(pos >= n)

printf("\n Invalid location");

else

{

++n;

for(i=n; i>pos; i--)

b[i] = b[i-1];

printf("\n Enter the element to insert: ");

scanf("%d", &e);

b[pos] = e;

}

printf("\n List after insertion:");

display();

}

void display()

{

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

printf("\n %d", b[i]);

}

**Output**

**Result**

Thus various operations was successfully executed on list using array implementation.

**Ex. No. 3a Singly Linked List**

**Date:**

**Aim:**

To define a singly linked list node and perform operations such as insertions and deletions dynamically

**Algorithm:**

1. Start

2. Define single linked list node as self referential structure

3. Create Head node with label = -1 and next = NULL using

4. Display menu on list operation

5. Accept user choice

6. If choice = 1 then

Locate node after which insertion is to be done

Create a new node and get data part

Insert new node at appropriate position by manipulating address

Else if choice = 2

Get node's data to be deleted.

Locate the node and delink the node

Rearrange the links

Else

Traverse the list from Head node to node which points to null

7. Stop

**Program:**

|  |
| --- |
|  |
|  |  |
|  |  |
|  |  |
| **/\* single linked list\*/**  **#include<stdio.h>**  **#include<stdlib.h>**  **struct node**  **{**  **int data;**  **struct node \*next;**  **}\*top,\*new1,\*first;**  **void main()**  **{**  **int wish,opt;**  **void create(),push(),pop(),view();**  **do**  **{**  **system("clear");**  **printf("Stack using linked list menu");**  **printf("\n1.Create\n2.Push\n3.Pop\n4.View\n5.Exit\n");**  **printf("\nEnter your option(1,2,3,4,5):");**  **scanf("%d",&wish);**  **switch(wish)**  **{**  **case 1:**  **create();**  **break;**  **case 2:**  **push();**  **break;**  **case 3:**  **pop();**  **break;**  **case 4:**  **view();**  **break;**  **case 5:**  **exit(0);**  **}**  **printf("\nDo you wnat to continue(0/1):");**  **scanf("%d",&opt);**  **}while(opt==1);**  **}**  **void create()**  **{**  **int ch;**  **top=(struct node\*)malloc(sizeof(struct node));**  **top->next=NULL;**  **do**  **{**  **system("clear");**  **printf("Enter the data:\n");**  **scanf("%d",&top->data);**  **printf("Do you want to insert another(1/0)\n");**  **scanf("%d",&ch);**  **if(ch==1)**  **{**  **new1=(struct node\*)malloc(sizeof(struct node));**  **new1->next=top;**  **top=new1;**  **first=top;**  **}**  **else**  **break;**  **}while(ch==1);**  **}**  **void push()**  **{**  **top=first;**  **new1=(struct node\*)malloc(sizeof(struct node));**  **printf("Enter the element to be pushed:");**  **scanf("%d",&new1->data);**  **new1->next=top;**  **top=new1;**  **first=top;**  **}**  **void pop()**  **{**  **system("clear");**  **top=first;**  **if(top==NULL)**  **printf("\n Stack is empty");**  **else**  **{**  **printf("\nThe element popped out from stack is %d",top->data);**  **top=top->next;**  **first=top;**  **}**  **}**  **void view()**  **{**  **printf("\nStack contents\n");**  **while(top->next!=NULL)**  **{**  **printf("%d->",top->data);**  **top=top->next;**  **}**  **printf("%d\n",top->data);**    **}** |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Result:**

Thus, operation on single linked list is performed

**Ex. No : 3b Stack Using Linked List**

**Date :**

**Aim:**

To implement stack operations using linked list.

**Algorithm:**

1. Start

2. Define a singly linked list node for stack

3. Create Head node

4. Display a menu listing stack operations

5. Accept choice

6. If choice = 1 then

Create a new node with data

Make new node point to first node

Make head node point to new node

Else If choice = 2 then

Make temp node point to first node

Make head node point to next of temp node

Release memory

Else If choice = 3 then

Display stack elements starting from head node till null

7. Stop

**Program:**

/\* Stack using Single Linked List \*/

#include <stdio.h>

#include <stdlib.h>

struct node

{

int label;

struct node \*next;

};

void main()

{

int ch = 0;

int k;

struct node \*h, \*temp, \*head;

/\* Head node construction \*/

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

head->next = NULL;

while(1)

{

printf("\n Stack using Linked List \n");

printf("1->Push ");

printf("2->Pop ");

printf("3->View ");

printf("4->Exit \n");

printf("Enter your choice : ");

scanf("%d", &ch);

switch(ch)

{

case 1:

/\* Create a new node \*/

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

printf("Enter label for new node : ");

scanf("%d", &temp->label);

h = head;

temp->next = h->next;

h->next = temp;

break;

case 2:

/\* Delink the first node \*/

h = head->next;

head->next = h->next;

printf("Node %s deleted\n", h->label);

free(h);

break;

case 3:

printf("\n HEAD -> ");

h = head;

/\* Loop till last node \*/

while(h->next != NULL)

{

h = h->next;

printf("%d -> ",h->label);

}

printf("NULL \n");

break;

case 4:

exit(0);

}

}

}

**Result:**

Thus, push and pop operations of a stack was demonstrated using linked list.

**Ex. No:3c Queue Using Linked List**

**Date :**

**Aim:**

To implement queue operations using linked list

**Algorithm:**

1. Start

2. Define a singly linked list node for stack

3. Create Head node

4. Display a menu listing stack operation

5. Accept choice

6. If choice = 1 then

Create a new node with data

Make new node point to first node

Make head node point to new node

Else If choice = 2 then

Make temp node point to first node

Make head node point to next of temp node

Release memory

Else If choice = 3 then

Display stack elements starting from head node till null

7. Stop

**Program:**

/\* Queue using Single Linked List \*/

#include <stdio.h>

#include <stdlib.h>

struct node

{

int label;

struct node \*next;

};

void main()

{

int ch=0;

int k;

struct node \*h, \*temp, \*head;

/\* Head node construction \*/

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

head->next = NULL;

while(1)

{

printf("\n Queue using Linked List \n");

printf("1->Insert ");

printf("2->Delete ");

printf("3->View ");

printf("4->Exit \n");

printf("Enter your choice : ");

scanf("%d", &ch);

switch(ch)

{

case 1:

/\* Create a new node \*/

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

printf("Enter label for new node : ");

scanf("%d", &temp->label);

/\* Reorganize the links \*/

h = head;

while (h->next != NULL)

h = h->next;

h->next = temp;

temp->next = NULL;

break;

case 2:

/\* Delink the first node \*/

h = head->next;

head->next = h->next;

printf("Node deleted \n");

free(h);

break;

case 3:

printf("\n\nHEAD -> ");

h=head;

while (h->next!=NULL)

{

h = h->next;

printf("%d -> ",h->label);

}

printf("NULL \n");

break;

case 4:

exit(0);

}

}

}

**Result:**

Thus, insert and delete operations of a queue was demonstrated using linked list.

**Ex. No. 4a Polynomial Addition**

**Date:**

**Aim:**

To add any two given polynomial using linked lists.

**Algorithm :**

1. Create a structure for polynomial with exp and coeff terms.

2. Read the coefficient and exponent of given two polynomials p and q.

3. While p and q are not null, repeat step 4.

If powers of the two terms are equal then

Insert the sum of the terms into the sum Polynomial

Advance p and q

Else if the power of the first polynomial> power of second then

Insert the term from first polynomial into sum polynomial

Advance p

Else

Insert the term from second polynomial into sum polynomial

Advance q

4. Copy the remaining terms from the non empty polynomial into the sum polynomial

5. Stop

**Program**

/\* Polynomial Addition \*/

/\* Add two polynomials \*/

#include<stdio.h>

#include<stdlib.h>

void main()

{

int a[10], b[10], c[10],m,n,k,k1,i,j,x;

system("clear");

printf("\n\tPolynomial Addition\n");

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

printf("\n\tEnter the no. of terms of the polynomial:");

scanf("%d", &m);

printf("\n\tEnter the degrees and coefficients:");

for (i=0;i<2\*m;i++)

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

printf("\n\tFirst polynomial is:");

k1=0;

if(a[k1+1]==1)

printf("x^%d", a[k1]);

else

printf("%dx^%d", a[k1+1],a[k1]);

k1+=2;

while (k1<i)

{

printf("+%dx^%d", a[k1+1],a[k1]);

k1+=2;

}

printf("\n\n\n\tEnter the no. of terms of 2nd polynomial:");

scanf("%d", &n);

printf("\n\tEnter the degrees and co-efficients:");

for(j=0;j<2\*n;j++)

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

printf("\n\tSecond polynomial is:");

k1=0;

if(b[k1+1]==1)

printf("x^%d", b[k1]);

else

printf("%dx^%d",b[k1+1],b[k1]);

k1+=2;

while (k1<2\*n)

{

printf("+%dx^%d", b[k1+1],b[k1]);

k1+=2;

}

i=0;

j=0;

k=0;

while (m>0 && n>0)

{

if (a[i]==b[j])

{

c[k+1]=a[i+1]+b[j+1];

c[k]=a[i];

m--;

n--;

i+=2;

j+=2;

}

else if (a[i]>b[j])

{

c[k+1]=a[i+1];

c[k]=a[i];

m--;

i+=2;

}

else

{

c[k+1]=b[j+1];

c[k]=b[j];

n--;

j+=2;

}

k+=2;

}

while (m>0)

{

c[k+1]=a[i+1];

c[k]=a[i];

k+=2;

i+=2;

m--;

}

while (n>0)

{

c[k+1]=b[j+1];

c[k]=b[j];

k+=2;

j+=2;

n--;

}

printf("\n\n\n\n\tSum of the two polynomials is:");

k1=0;

if (c[k1+1]==1)

printf("x^%d", c[k1]);

else

printf("%dx^%d", c[k1+1],c[k1]);

k1+=2;

while (k1<k)

{

if (c[k1+1]==1)

printf("+x^%d", c[k1]);

else

printf("+%dx^%d", c[k1+1], c[k1]);

k1+=2;

}

}

**Output**

**Result:**

Thus, the two given polynomials were added using lists.

**Ex. No. 4b Infix To Postfix Conversion**

**Date:**

**Aim**

To convert infix expression to its postfix form using stack operations.

**Algorithm**

1. Start

2. Define a array stack of size max = 20

3. Initialize top = -1

4. Read the infix expression character-by-character

If character is an operand print it

If character is an operator

Compare the operator’s priority with the stack[top] operator.

If the stack [top] has higher/equal priority than the input operator,

Pop it from the stack and print it.

Else

Push the input operator onto the stack

If character is a left parenthesis, then push it onto the stack.

If character is a right parenthesis, pop all operators from stack and print

it until a left parenthesis is encountered. Do not print the parenthesis.

If character = $ then Pop out all operators, Print them and Stop

**Program**

/\* Conversion of infix to postfix expression \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX 20

int top = -1;

char stack[MAX];

char pop();

void push(char item);

int prcd(char symbol)

{

switch(symbol)

{

case '+':

case '-':

return 2;

break;

case '\*':

case '/':

return 4;

break;

case '^':

case '$':

return 6;

break;

case '(':

case ')':

case '#':

return 1;

break;

}

}

int isoperator(char symbol)

{

switch(symbol)

{

case '+':

case '-':

case '\*':

case '/':

case '^':

case '$':

case '(':

case ')':

return 1;

break;

default:

return 0;

}

}

void convertip(char infix[],char postfix[])

{

int i,symbol,j = 0;

stack[++top] = '#';

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

{

symbol = infix[i];

if(isoperator(symbol) == 0)

{

postfix[j] = symbol;

j++;

}

else

{

if(symbol == '(')

push(symbol);

else if(symbol == ')')

{

while(stack[top] != '(')

{

postfix[j] = pop();

j++;

}

pop(); //pop out (.

}

else

{

if(prcd(symbol) > prcd(stack[top]))

push(symbol);

else

{

while(prcd(symbol) <= prcd(stack[top]))

{

postfix[j] = pop();

j++;

}

push(symbol);

}

}

}

}

while(stack[top] != '#')

{

postfix[j] = pop();

j++;

}

postfix[j] = '\0';

}

main()

{

char infix[20],postfix[20];

system("clear");

printf("Enter the valid infix string: ");

fgets(infix,20,stdin);

convertip(infix, postfix);

printf("The corresponding postfix string is: ");

puts(postfix);

}

void push(char item)

{

top++;

stack[top] = item;

}

char pop()

{

char a;

a = stack[top];

top--;

return a;

}

**Output**

**Result**

Thus the given infix expression was converted into postfix form using stack.

**Ex. No. 4c Postfix Expression Evaluation**

**Date:**

**Aim**

To evaluate the given postfix expression using stack operations.

**Algorithm**

1. Start

2. Define a array stack of size max = 20

3. Initialize top = -1

4. Read the postfix expression character-by-character If character is an operand push it onto the stack If character is an operator Pop topmost two elements from stack. Apply operator on the elements and push the result onto the stack,

5. Eventually only result will be in the stack at end of the expression.

6. Pop the result and print it.

7. Stop

**Program**

/\* Evaluation of Postfix expression using stack \*/

#include <stdio.h>

#include <stdlib.h>

struct stack

{

int top;

float a[50];

}s;

void main()

{

char pf[50];

float d1,d2,d3;

int i;

system("clear");

s.top = -1;

printf("\n\n Enter the postfix expression: ");

fgets(pf,20,stdin);

for(i=0; pf[i]!='\0'; i++)

{

switch(pf[i])

{

case '0':

case '1':

case '2':

case '3':

case '4':

case '5':

case '6':

case '7':

case '8':

case '9':

s.a[++s.top] = pf[i]-'0';

break;

case '+':

d1 = s.a[s.top--];

d2 = s.a[s.top--];

s.a[++s.top] = d1 + d2;

break;

case '-':

d2 = s.a[s.top--];

d1 = s.a[s.top--];

s.a[++s.top] = d1 - d2;

break;

case '\*':

d2 = s.a[s.top--];

d1 = s.a[s.top--];

s.a[++s.top] = d1\*d2;

break;

case '/':

d2 = s.a[s.top--];

d1 = s.a[s.top--];

s.a[++s.top] = d1 / d2;

break;

}

}

printf("\n Expression value is %5.2f", s.a[s.top]);

}

**Output**

**Result**

Thus, the given postfix expression was evaluated using stack.

**Ex. No. 5 Binary Tree Traversal**

**Date:**

**Aim:**

To implement different types of traversal for the given binary tree

**Algorithm:**

1. Create a structure with key and 2 pointer variables left and right
2. 2. Read the node to be inserted.

If (root==NULL)

root=node

else if (root->key < node->key)

root->right=NULL

else

Root->left=node

1. For Inorder Traversal

Traverse Left subtree

Visit root

Traverse Right subtree

1. For Preorder Traversal

Visit root

Traverse Left subtree

Traverse Right subtree

1. For Postorder Traversal

Traverse Left subtree

Traverse Right subtree

Visit root

1. Stop

**Program:**

/\* Tree Traversal \*/

#include <stdio.h>

#include <stdlib.h>

typedef struct node

{

int data;

struct node \*left;

struct node \*right;

}node;

int count=1;

node \*insert(node \*tree,int digit)

{

if(tree == NULL)

{

tree = (node \*)malloc(sizeof(node));

tree->left = tree->right=NULL;

tree->data = digit;

count++;

}

else if(count%2 == 0)

tree->left = insert(tree->left, digit);

else

tree->right = insert(tree->right, digit);

return tree;

}

void preorder(node \*t)

{

if(t != NULL)

{

printf(" %d", t->data);

preorder(t->left);

preorder(t->right);

}

}

void postorder(node \*t)

{

if(t != NULL)

{

postorder(t->left);

postorder(t->right);

printf(" %d", t->data);

}

}

void inorder(node \*t)

{

if(t != NULL)

{

inorder(t->left);

printf(" %d", t->data);

inorder(t->right);

}

}

void main()

{

node \*root = NULL;

int digit;

puts("Enter integer:To quit enter 0");

scanf("%d", &digit);

while(digit != 0)

{

root=insert(root,digit);

scanf("%d",&digit);

}

printf("\nThe preorder traversal of tree is:\n");

preorder(root);

printf("\nThe inorder traversal of tree is:\n");

inorder(root);

printf("\nThe postorder traversal of tree is:\n");

postorder(root);

}

**Output:**

**Result:**

Thus, three types of tree traversal were performed on the given binary tree

**Ex. No. 6 Binary Search Tree**

**Date:**

**Aim**

To insert and delete nodes in a binary search tree.

**Algorithm**

1. Create a structure with key and 2 pointer variable left and right.

2. Read the node to be inserted.

If (root==NULL)

root=node

else if (root->keykey)

root->right=NULL

else

Root->left=node

3. For Deletion

if it is a leaf node

Remove immediately

Remove pointer between del node & child

if it is having one child

Remove link between del node&child

Link delnode is child with delnodes parent

If it is a node with a children

Find min value in right subtree

Copy min value to delnode place

Delete the duplicate

4. Stop

**Program**

/\* Binary Search Tree \*/

#include <stdio.h>

#include <stdlib.h>

struct node

{

int key;

struct node \*left;

struct node \*right;

};

struct node \*newNode(int item)

{

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

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

void inorder(struct node \*root)

{

if (root != NULL)

{

inorder(root->left);

printf("%d ", root->key);

inorder(root->right);

}

}

struct node\* insert(struct node\* node, int key)

{

if (node == NULL)

return newNode(key);

if (key < node->key)

node->left = insert(node->left, key);

else

node->right = insert(node->right, key);

return node;

}

struct node \* minValueNode(struct node\* node)

{

struct node\* current = node;

while (current->left != NULL)

current = current->left;

return current;

}

struct node\* deleteNode(struct node\* root, int key)

{

struct node \*temp;

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)

{

temp = root->right;

free(root);

return temp;

}

else if (root->right == NULL)

{

temp = root->left;

free(root);

return temp;

}

temp = minValueNode(root->right);

root->key = temp->key;

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

}

return root;

}

void main()

{

struct node \*root = NULL;

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 70);

root = insert(root, 60);

root = insert(root, 80);

printf("Inorder traversal of the given tree \n");

inorder(root);

printf("\nDelete 20\n");

root = deleteNode(root, 20);

printf("Inorder traversal of the modified tree \n");

inorder(root);

printf("\nDelete 30\n");

root = deleteNode(root, 30);

printf("Inorder traversal of the modified tree \n");

inorder(root);

printf("\nDelete 50\n");

root = deleteNode(root, 50);

printf("Inorder traversal of the modified tree \n");

inorder(root);

}

**Output:**

**Result**

Thus, nodes were inserted and deleted from a binary search tree.

Ex. No. 7 AVL Trees

Date:

**Aim:**

To perform insertion operation on an AVL tree and to maintain balance factor.

**Algorithm:**

1. Start

2. Perform standard BST insert for w

3. Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.

4. Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways.

a) y is left child of z and x is left child of y (Left Left Case)

b) y is left child of z and x is right child of y (Left Right Case)

c) y is right child of z and x is right child of y (Right Right Case) d) y is right child of z and x is left child of y (Right Left Case)

5. Stop

**Program:**

/\* AVL Tree \*/

#include <stdio.h>

#include <stdlib.h>

#define CHANGED 0

#define BALANCED 1

typedef struct bnode

{

int data,bfactor;

struct bnode \*left;

struct bnode \*right;

}node;

int height;

void displaymenu()

{

printf("\nBasic Operations in AVL tree");

printf("\n0.Display menu list");

printf("\n1.Insert a node in AVL tree");

printf("\n2.View AVL tree");

printf("\n3.Exit");

}

node\* getnode()

{

int size;

node \*newnode;

size = sizeof(node);

newnode = (node\*)malloc(size);

return(newnode);

}

void copynode(node \*r, int data)

{

r->data = data;

r->left = NULL;

r->right = NULL;

r->bfactor = 0;

}

void releasenode(node \*p)

{

free(p);

}

node\* searchnode(node \*root, int data)

{

if(root!=NULL)

if(data < root->data)

root = searchnode(root->left, data);

else if(data > root->data)

root = searchnode(root->right, data);

return(root);

}

void lefttoleft(node \*\*pptr, node \*\*aptr)

{

node \*p = \*pptr, \*a = \*aptr;

printf("\nLeft to Left AVL rotation");

p->left = a->right;

a->right = p;

if(a->bfactor == 0)

{

p->bfactor = 1;

a->bfactor = -1;

height = BALANCED;

}

else

{

p->bfactor = 0;

a->bfactor = 0;

}

p = a;

\*pptr = p;

\*aptr = a;

}

void lefttoright(node \*\*pptr, node \*\*aptr, node \*\*bptr)

{

node \*p = \*pptr, \*a = \*aptr, \*b = \*bptr;

printf("\nLeft to Right AVL rotation");

b = a->right;

b->right = p;

if(b->bfactor == 1)

p->bfactor = -1;

else

p->bfactor = 0;

if(b->bfactor == -1)

a->bfactor = 1;

else

a->bfactor = 1;

b->bfactor = 0;

p = b;

\*pptr = p;

\*aptr = a;

\*bptr = b;

}

void righttoright(node \*\*pptr, node \*\*aptr)

{

node \*p = \*pptr, \*a = \*aptr;

printf("\nRight to Right AVL rotation");

p->right = a->left;

a->left = p;

if(a->bfactor == 0)

{

p->bfactor = -1;

a->bfactor = 1;

height = BALANCED;

}

else

{

p->bfactor = 0;

a->bfactor = 0;

}

p = a;

\*pptr = p;

\*aptr = a;

}

void righttoleft(node \*\*pptr, node \*\*aptr, node \*\*bptr)

{

node \*p = \*pptr, \*a = \*aptr, \*b = \*bptr;

printf("\nRight to Left AVL rotation");

b = a->left;

a->left = b->right;

b->right = a;

p->right = b->left;

b->left = p;

if(b->bfactor == -1)

p->bfactor = 1;

else

p->bfactor = 0;

if(b->bfactor == -1)

a->bfactor = 0;

b->bfactor = 0;

p = b;

\*pptr = p;

\*aptr = a;

\*bptr = b;

}

void inorder(node \*root)

{

if(root == NULL)

return;

inorder(root->left);

printf("\n%4d", root->data);

inorder(root->right);

}

void view(node \*root, int level)

{

int k;

if(root == NULL)

return;

view(root->right, level+1);

printf("\n");

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

printf(" ");

printf("%d", root->data);

view(root->left, level+1);

}

node\* insertnode(int data, node \*p)

{

node \*a,\*b;

if(p == NULL)

{

p=getnode();

copynode(p, data);

height = CHANGED;

return(p);

}

if(data < p->data)

{

p->left = insertnode(data, p->left);

if(height == CHANGED)

{

switch(p->bfactor)

{

case -1:

p->bfactor = 0;

height = BALANCED;

break;

case 0:

p->bfactor = 1;

break;

case 1:

a = p->left;

if(a->bfactor == 1)

lefttoleft(&p, &a);

else

lefttoright(&p, &a, &b);

height = BALANCED;

break;

}

}

}

if(data > p->data)

{

p->right = insertnode(data, p->right);

if(height == CHANGED)

{

switch(p->bfactor)

{

case 1:

p->bfactor = 0;

height = BALANCED;

break;

case 0:

p->bfactor = -1;

break;

case -1:

a=p->right;

if(a->bfactor == -1)

righttoright(&p, &a);

else

righttoleft(&p, &a, &b);

height=BALANCED;

break;

}

}

}

return(p);

}

void main()

{

int data, ch;

char choice = 'y';

node \*root = NULL;

system("clear");

displaymenu();

while((choice == 'y') || (choice == 'Y'))

{

printf("\nEnter your choice: ");

fflush(stdin);

scanf("%d",&ch);

switch(ch)

{

case 0:

displaymenu();

break;

case 1:

printf("Enter the value to be inserted ");

scanf("%d", &data);

if(searchnode(root, data) == NULL)

root = insertnode(data, root);

else

printf("\nData already exists");

break;

case 2:

if(root == NULL)

{

printf("\nAVL tree is empty");

continue;

}

printf("\nInorder traversal of AVL tree");

inorder(root);

printf("\nAVL tree is");

view(root, 1);

break;

case 3:

releasenode(root);

exit(0);

}

}

}

**Output:**

**Result :**

Thus rotations were performed as a result of insertions to AVL Tree..

**Ex. No. 8 Binary Heap**

**Date:**

**Aim:**

To build a binary heap from an array of input elements.

**Algorithm:**

1. Start

2. In a heap, for every node x with parent p, the key in p is smaller than or equal to the key in x.

3. For insertion operation

a. Add the element to the bottom level of the heap.

b. Compare the added element with its parent; if they are in the correct order, stop.

c. If not, swap the element with its parent and return to the previous step

4. For deleteMin operation

a. Replace the root of the heap with the last element on the last level.

b. Compare the new root with its children; if they are in the correct order, stop.

c. If not, Swap with its smaller child in a min-heap

5. Stop

**Program:**

/\* Binary Heap \*/

#include <stdio.h>

#include <limits.h>

int heap[1000000], heapSize;

void Init()

{

heapSize = 0;

heap[0] = -INT\_MAX;

}

void Insert(int element)

{

heapSize++;

heap[heapSize] = element;

int now = heapSize;

while (heap[now / 2] > element)

{

heap[now] = heap[now / 2];

now /= 2;

}

heap[now] = element;

}

int DeleteMin()

{

int minElement, lastElement, child, now;

minElement = heap[1];

lastElement = heap[heapSize--];

for (now = 1; now \* 2 <= heapSize; now = child)

{

child = now \* 2;

if (child != heapSize && heap[child + 1] < heap[child])

child++;

if (lastElement > heap[child])

heap[now] = heap[child];

else

break;

}

heap[now] = lastElement;

return minElement;

}

void main()

{

int number\_of\_elements;

printf("Program to demonstrate Heap:\nEnter the number of number\_of\_elements:");

scanf("%d", &number\_of\_elements);

int iter, element;

Init();

printf("Enter the elements: ");

for (iter = 0; iter < number\_of\_elements; iter++)

{

scanf("%d", &element);

Insert(element);

}

for (iter = 0; iter < number\_of\_elements; iter++)

printf("%d ", DeleteMin());

printf("\n");

}

**Result:**

Thus a binary heap is constructed for the given elements

Ex. No. 9a **Breadth First Search**

Date:

**Aim**

To create adjacency matrix of the given graph and to perform breadth first search traversal.

**Algorithm**

1. Start

2. Obtain Adjacency matrix for the given graph

3. Define a Queue of size total number of vertices in the graph

4. Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

5. Visit all the adjacent vertices of the verex which is at front of the Queue which is not visited and insert them into the Queue.

6. When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

7. Repeat step 5 and 6 until queue becomes empty.

8. When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph.

9. Stop

**Program**

/\* Graph Traversal – BFS \*/

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

#define initial 1

#define waiting 2

#define visited 3

int n;

int adj[MAX][MAX];

int state[MAX];

void create\_graph();

void BF\_Traversal();

void BFS(int v);

int queue[MAX], front = -1,rear = -1;

void insert\_queue(int vertex);

int delete\_queue();

int isEmpty\_queue();

int main()

{

create\_graph();

BF\_Traversal();

return 0;

}

void BF\_Traversal()

{

int v;

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

state[v] = initial;

printf("Enter Start Vertex for BFS: ");

scanf("%d", &v);

BFS(v);

}

void BFS(int v)

{

int i;

insert\_queue(v);

state[v] = waiting;

printf("BFS Traversal : ");

while(!isEmpty\_queue())

{

v = delete\_queue( );

printf("%d ", v);

state[v] = visited;

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

{

if(adj[v][i] == 1 && state[i] == initial)

{

insert\_queue(i);

state[i] = waiting;

}

}

}

printf("\n");

}

void insert\_queue(int vertex)

{

if(rear == MAX-1)

printf("Queue Overflow\n");

else

{

if(front == -1)

front = 0;

rear = rear+1;

queue[rear] = vertex ;

}

}

int isEmpty\_queue()

{

if(front == -1 || front > rear)

return 1;

else

return 0;

}

int delete\_queue()

{

int delete\_item;

if(front == -1 || front > rear)

{

printf("Queue Underflow\n");

exit(1);

}

delete\_item = queue[front];

front = front+1;

return delete\_item;

}

void create\_graph()

{

int count,max\_edge,origin,destin;

printf("Enter number of vertices : ");

scanf("%d", &n);

max\_edge = n \* (n-1);

for(count=1; count<=max\_edge; count++)

{

printf("Enter edge %d( -1 -1 to quit ) : ",count);

scanf("%d %d", &origin, &destin);

if((origin == -1) && (destin == -1))

break;

if(origin>=n || destin>=n || origin<0 || destin<0)

{

printf("Invalid edge!\n");

count--;

}

else

adj[origin][destin] = 1;

}

}

**Output**

**Result**

Thus Breadth First Traversal is executed on the given graph.

Ex. No. 9b **Depth First Search**

Date:

**Aim**

To create adjacency matrix of the given graph and to perform depth first search traversal.

**Algorithm**

1. Start

2. Obtain Adjacency matrix for the given graph

3. Define a Stack of size total number of vertices in the graph.

4. Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

5. Visit any one of the adjacent vertex of the verex which is at top of the stack which is not visited and push it on to the stack.

6. Repeat step 5 until there are no new vertex to be visit from the vertex on top of the stack.

7. When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.

8. Repeat steps 5, 6 and 7 until stack becomes Empty.

9. When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph.

10. Stop

**Program**

/\* DFS on undirected graph \*/

#include <stdio.h>

#include <stdlib.h>

#define true 1

#define false 0

#define MAX 5

struct Vertex

{

char label;

int visited;

};

int stack[MAX];

int top = -1;

struct Vertex\* lstVertices[MAX];

static int adjMatrix[MAX][MAX];

int vertexCount = 0;

void push(int item)

{

stack[++top] = item;

}

int pop()

{

return stack[top--];

}

int peek()

{

return stack[top];

}

int isStackEmpty()

{

return top == -1;

}

void addVertex(char label)

{

struct Vertex\* vertex = (struct Vertex\*)

malloc(sizeof(struct Vertex));

vertex->label = label;

vertex->visited = false;

lstVertices[vertexCount++] = vertex;

}

void addEdge(int start, int end)

{

adjMatrix[start][end] = 1;

adjMatrix[end][start] = 1;

}

void displayVertex(int vertexIndex)

{

printf("%c ", lstVertices[vertexIndex]->label);

}

int getAdjUnvisitedVertex(int vertexIndex)

{

int i;

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

{

if(adjMatrix[vertexIndex][i] == 1 &&

lstVertices[i]->visited == false)

return i;

}

return -1;

}

void depthFirstSearch()

{

int i;

lstVertices[0]->visited = true;

displayVertex(0);

push(0);

while(!isStackEmpty())

{

int unvisitedVertex = getAdjUnvisitedVertex(peek());

if(unvisitedVertex == -1)

pop();

else

{

lstVertices[unvisitedVertex]->visited = true;

displayVertex(unvisitedVertex);

push(unvisitedVertex);

}

}

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

lstVertices[i]->visited = false;

}

void main()

{

int i, j, n, edges, orgn, destn;

char ch;

printf("Enter no. of vertices : ");

scanf("%d", &n);

edges = n \* (n - 1);

printf("Enter Vertex Labels : \n");

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

{

fflush(stdin);

scanf("%c", &ch);

addVertex(ch);

}

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

{

printf("Enter edge ( -1 -1 to quit ) : ");

scanf("%d %d", &orgn, &destn);

if((orgn == -1) && (destn == -1))

break;

if(orgn>=n || destn>=n || orgn<0 || destn<0)

printf("Invalid edge!\n");

else

addEdge(orgn, destn);

}

printf("\nDepth First Search: ");

depthFirstSearch();

}

**Output**

**Result**

Thus depth first traversal is executed on the given undirected graph.

Ex. No. 9c Dijkstra’s Shortest Path

Date:

**Aim :**

To find the shortest path for the given graph from a specified source to all other vertices using Dijkstra’s algorithm.

**Algorithm**

1. Start

2. Obtain no. of vertices and adjacency matrix for the given graph

3. Create cost matrix from adjacency matrix. C[i][j] is the cost of going from vertex i to vertex j. If there is no edge between vertices i and j then C[i][j] is infinity

4. Initialize visited[] to zero

5. Read source vertex and mark it as visited

6. Create the distance matrix, by storing the cost of vertices from vertex no. 0 to n-1 from the source vertex

distance[i]=cost[0][i];

7. Choose a vertex w, such that distance[w] is minimum and visited[w] is 0. Mark visited[w] as 1.

8. Recalculate the shortest distance of remaining vertices from the source.

9. Only, the vertices not marked as 1 in array visited[ ] should be considered for recalculation of distance. i.e. for each vertex v

if(visited[v]==0) distance[v]=min(distance[v]

distance[w]+cost[w][v])

10. Stop

**Program:**

/\* Dijkstra’s Shortest Path \*/

#include <stdio.h>

#include <stdlib.h>

#define INFINITY 9999

#define MAX 10

void dijkstra(int G[MAX][MAX], int n, int startnode);

void main()

{

int G[MAX][MAX], i, j, n, u;

printf("Enter no. of vertices: ");

scanf("%d", &n);

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

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

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

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

printf("Enter the starting node: ");

scanf("%d", &u);

dijkstra(G, n, u);

}

void dijkstra(int G[MAX][MAX], int n,int startnode)

{

int cost[MAX][MAX], distance[MAX], pred[MAX];

int visited[MAX],count, mindistance, nextnode, i, j;

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

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

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

cost[i][j] = INFINITY;

else

cost[i][j] = G[i][j];

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

{

distance[i] = cost[startnode][i];

pred[i] = startnode;

visited[i] = 0;

}

distance[startnode] = 0;

visited[startnode] = 1;

count = 1;

while(count < n-1)

{

mindistance = INFINITY;

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

if(distance[i] < mindistance && !visited[i])

{

mindistance = distance[i];

nextnode=i;

}

visited[nextnode] = 1;

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

if(!visited[i])

if(mindistance + cost[nextnode][i] < distance[i])

{

distance[i] = mindistance +

cost[nextnode][i];

pred[i] = nextnode;

}

count++;

}

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

if(i != startnode)

{

printf("\nDistance to node%d = %d", i,distance[i]);

printf("\nPath = %d", i);

j = i;

do

{

j = pred[j];

printf("<-%d", j);

} while(j != startnode);

}

}

**Result:**

Thus Dijkstra's algorithm is used to find shortest path from a given vertex.

**Ex. No. 10a Linear Search**

**Date:**

**Aim :**

To perform linear search of an element on the given array.

**Algorithm :**

1. Start

2. Read number of array elements n

3. Read array elements Ai, i = 0,1,2,…n–1

4. Read search value

5. Assign 0 to found

6. Check each array element against search

If Ai = search then

found = 1

Print "Element found" Print position i

Stop

7. If found = 0 then

print "Element not found"

8. Stop

**Program:**

/\* Linear search on a sorted array \*/

#include <stdio.h>

#include <stdlib.h>

void main()

{

int a[50],i, n, val, found;

system("clear");

printf("Enter number of elements : ");

scanf("%d", &n);

printf("Enter Array Elements : \n");

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

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

printf("Enter element to locate : ");

scanf("%d", &val);

found = 0;

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

{

if (a[i] == val)

{

printf("Element found at position %d", i);

found = 1;

break;

}

}

if (found == 0)

printf("\n Element not found");

}

**Output:**

**Result :**

Thus an array was linearly searched for an element's existence

**Ex. No. 10b Binary Search**

**Date:**

**Aim:**

To locate an element in a sorted array using Binary search method

**Algorithm:**

1. Start

2. Read number of array elements, say n

3. Create an array arr consisting n sorted elements

4. Get element, say key to be located

5. Assign 0 to lower and n to upper

6. While (lower < upper)

Determine middle element mid = (upper+lower)/2

If key = arr[mid] then

Print mid

Stop

Else if key > arr[mid] then

lower = mid + 1

else

upper = mid – 1

7. Print "Element not found"

8. Stop

**Program:**

/\* Binary Search on a sorted array \*/

#include <stdio.h>

#include <stdlib.h>

void main()

{

int a[50],i, n, upper, lower, mid, val, found;

system("clear");

printf("Enter array size : ");

scanf("%d", &n);

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

a[i] = 2 \* i;

printf("\n Elements in Sorted Order \n");

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

printf("%4d", a[i]);

printf("\n Enter element to locate : ");

scanf("%d", &val);

upper = n;

lower = 0;

found = -1;

while (lower <= upper)

{

mid = (upper + lower)/2;

if (a[mid] == val)

{

printf("Located at position %d", mid);

found = 1;

break;

}

else if(a[mid] > val)

upper = mid - 1;

else

lower = mid + 1;

}

if (found == -1)

printf("Element not found");

}

**Output:**

**Result:**

Thus an element is located quickly using binary search method.

Ex. No. 10c **Bubble Sort**

Date:

**Aim**

To sort an array of N numbers using Bubble sort

**Algorithm**

1. Start

2. Read number of array elements n

3. Read array elements Ai

4. Index i varies from 0 to n-2

5. Index j varies from i+1 to n-1

6. Traverse the array and compare each pair of elements If Ai > Aj then Swap Ai and A

7. Stop

**Program**

/\* Bubble Sort \*/

#include <stdio.h>

#include <stdlib.h>

void main()

{

int a[50],i, j, n, t;

system("clear");

printf("Enter number of elements : ");

scanf("%d", &n);

printf("Enter Array Elements \n");

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

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

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

{

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

{

if (a[i] > a[j])

{

t = a[i];

a[i] = a[j];

a[j] = t;

}

}

}

printf("\n Elements in Sorted order :");

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

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

}

**Output**

**Result**

Thus an array was sorted using bubble sort.

Ex. No. 10d **Quick Sort**

Date:

**Aim:**

To sort an array of N numbers using Quick sort.

**Algorithm:**

1. Start

2. Read number of array elements n

3. Read array elements Ai

4. Select an pivot element x from Ai

5. Divide the array into 3 sequences: elements < x, x, elements > x

6. Recursively quick sort both sets (Ai < x and Ai > x)

7. Stop

**Program:**

/\* quick sort \*/

/\* C implementation QuickSort \*/

#include<stdio.h>

// A utility function to swap two elements

void swap(int\* a, int\* b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

/\* This function takes last element as pivot, places

the pivot element at its correct position in sorted

array, and places all smaller (smaller than pivot)

to left of pivot and all greater elements to right

of pivot \*/

int partition (int arr[], int low, int high)

{

int pivot = arr[high]; // pivot

int i = (low - 1); // Index of smaller element

for (int j = low; j <= high- 1; j++)

{

// If current element is smaller than the pivot

if (arr[j] < pivot)

{

i++; // increment index of smaller element

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

/\* The main function that implements QuickSort

arr[] --> Array to be sorted,

low --> Starting index,

high --> Ending index \*/

void quickSort(int arr[], int low, int high)

{

if (low < high)

{

/\* pi is partitioning index, arr[p] is now

at right place \*/

int pi = partition(arr, low, high);

// Separately sort elements before

// partition and after partition

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

int i;

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

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

printf("n");

}

// Driver program to test above functions

int main()

{

int arr[] = {10, 7, 8, 9, 1, 5};

int n = sizeof(arr)/sizeof(arr[0]);

quickSort(arr, 0, n-1);

printf("Sorted array: n");

printArray(arr, n);

return 0;

}

**Output:**

**Result :**

Thus an array was sorted using quick sort's divide and conquer method**.**

Ex. No. 10e **Merge Sort**

Date:

**Aim**

To sort an array of N numbers using Merge sort.

**Algorithm**

1. Start

2. Read number of array elements n

3. Read array elements Ai

4. Divide the array into sub-arrays with a set of elements

5. Recursively sort the sub-arrays

6. Merge the sorted sub-arrays onto a single sorte

7. Stop

**Program**

/\* Merge sort \*/

#include <stdio.h>

#include <stdlib.h>

void merge(int [],int ,int ,int );

void part(int [],int ,int );

int size;

void main()

{

int i, arr[30];

printf("Enter total no. of elements : ");

scanf("%d", &size);

printf("Enter array elements : ");

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

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

part(arr, 0, size-1);

printf("\n Merge sorted list : ");

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

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

}

void part(int arr[], int min, int max)

{

int i, mid;

if(min < max)

{

mid = (min + max) / 2;

part(arr, min, mid);

part(arr, mid+1, max);

merge(arr, min, mid, max);

}

if (max-min == (size/2)-1)

{

printf("\n Half sorted list : ");

for(i=min; i<=max; i++)

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

}

}

void merge(int arr[],int min,int mid,int max)

{

int tmp[30];

int i, j, k, m;

j = min;

m = mid + 1;

for(i=min; j<=mid && m<=max; i++)

{

if(arr[j] <= arr[m])

{

tmp[i] = arr[j];

j++;

}

else

{

tmp[i] = arr[m];

m++;

}

}

if(j > mid)

{

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

{

tmp[i] = arr[k];

i++;

}

}

else

{

for(k=j; k<=mid; k++)

{

tmp[i] = arr[k];

i++;

}

}

for(k=min; k<=max; k++)

arr[k] = tmp[k];

}

**Output**

**Result**

Thus array elements was sorted using merge sort's divide and conquer method.

Ex. No. 12 **Open Addressing Hashing Technique**

Date:

**Aim:**

To implement hash table using a C program.

**Algorithm:**

1. Create a structure, data (hash table item) with key and value as data.

2. Now create an array of structure, data of some certain size (10, in this case). But, the size of array must be immediately updated to a prime number just greater than initial array capacity (i.e 10, in this case).

3. A menu is displayed on the screen.

4. User must choose one option from four choices given in the menu

5. Perform all the operations

6. Stop

**Program:**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <stdbool.h>

#define SIZE 20

struct DataItem {

int data;

int key;

};

struct DataItem\* hashArray[SIZE];

struct DataItem\* dummyItem;

struct DataItem\* item;

int hashCode(int key) {

return key % SIZE;

}

struct DataItem \*search(int key) {

//get the hash

int hashIndex = hashCode(key);

//move in array until an empty

while(hashArray[hashIndex] != NULL) {

if(hashArray[hashIndex]->key == key)

return hashArray[hashIndex];

//go to next cell

++hashIndex;

//wrap around the table

hashIndex %= SIZE;

}

return NULL;

}

void insert(int key,int data) {

struct DataItem \*item = (struct DataItem\*) malloc(sizeof(struct DataItem));

item->data = data;

item->key = key;

//get the hash

int hashIndex = hashCode(key);

//move in array until an empty or deleted cell

while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {

//go to next cell

++hashIndex;

//wrap around the table

hashIndex %= SIZE;

}

hashArray[hashIndex] = item;

}

struct DataItem\* delete(struct DataItem\* item) {

int key = item->key;

//get the hash

int hashIndex = hashCode(key);

//move in array until an empty

while(hashArray[hashIndex] != NULL) {

if(hashArray[hashIndex]->key == key) {

struct DataItem\* temp = hashArray[hashIndex];

//assign a dummy item at deleted position

hashArray[hashIndex] = dummyItem;

return temp;

}

//go to next cell

++hashIndex;

//wrap around the table

hashIndex %= SIZE;

}

return NULL;

}

void display() {

int i = 0;

for(i = 0; i<SIZE; i++) {

if(hashArray[i] != NULL)

printf(" (%d,%d)",hashArray[i]->key,hashArray[i]->data);

else

printf(" ~~ ");

}

printf("\n");

}

int main() {

dummyItem = (struct DataItem\*) malloc(sizeof(struct DataItem));

dummyItem->data = -1;

dummyItem->key = -1;

insert(1, 20);

insert(2, 70);

insert(42, 80);

insert(4, 25);

insert(12, 44);

insert(14, 32);

insert(17, 11);

insert(13, 78);

insert(37, 97);

display();

item = search(37);

if(item != NULL) {

printf("Element found: %d\n", item->data);

} else {

printf("Element not found\n");

}

delete(item);

item = search(37);

if(item != NULL) {

printf("Element found: %d\n", item->data);

} else {

printf("Element not found\n");

}

}

**Output:**

**Result:**

Thus hashing has been performed successfully.