TPCT'S
TERNA ENGINEERING COLLEGE,
NERUL, NAVI MUMBAI

Data Structure & Algorithm Laboratory

LAB MANUAL

SH- 2020

Gaurav Deshmukh	Prof. Vikram Vyavhare
Sub Incharge	HOD

Lab Objective:

1.	To design and implement various data structures and their operations.
2.	Apply the appropriate search method on a given problem
3.	To develop application using suitable data structure and algorithms.

Lab Outcome:

1.	Implement various operations using linear data structures.
2.	Apply concepts of Trees and Graphs to a given problem.
3.	Analyze time and space complexity of an algorithm.
4.	Apply divide and conquer strategy to solve problems.
5.	Apply the concept of Greedy and Dynamic Programming approach to solve problems.
6.	Apply the concept of backtracking, branch and bound strategy to solve problems.

Do’s and Don’ts in Laboratory:

1. Make entry in the Log Book as soon as you enter the Laboratory.

2. All the students should sit according to their roll numbers starting from their left to right.

3. All the students are supposed to enter the terminal number in the log book.

4. Do not change the terminal on which you are working.

5. All the students are expected to get at least the algorithm of the program/concept to be implemented.

6.Strictly observe the instructions given by the teacher/Lab Instructor.

Instructions for STUDENTS:

1. Submission related to whatever lab work has been completed should be done during the next lab session. The immediate arrangements for printouts related to submission on the day of practical assignments.

2. Students should be taught for taking the printouts under the observation of lab teacher.

3. The promptness of submission should be encouraged by way of marking and evaluation patterns that will benefit the sincere students.

The pre-requisite for Data structures and algorithm analysis is knowledge of C language. It includes-

- Arrays and Strings,
- Switch case statement
- Pointers,
- Structures, and
- Dynamic memory allocation

Terna Engineering College

Department of Mechatronics

Experiment List
SH-2020

Subject: Data Structure and Algorithms lab **Class/Sem: SE MTX/ III**

Sr. No.	**Experiments**	**LO**	**CO**	**PO**
	WAP using structures to read and display the information about all students in a class			
1	Implementation of any one application of stack / Queue/Circular Queue			
2	Implementation of operations on Linked Lists			
3	Implementation of stack and queue using Link list.			
4	Implementation and analysis of selection sort/insertion sort.			
5	Implementation of Binary search/ merge sort/quick sort			
6	Implementation of operations on Binary Tree/Binary Search Tree/ Heap			
7	Implementation Greedy method algorithms Prim's/ Kruskal's algorithm			
8	Implementation of Dynamic programming approach algorithms knapsack/Traveling sales persons problem			
9	Implementation of Backtracking & branch and bound technique : N queens problem/15 puzzle problem			
10	Implementation of any game based on uninformed/informed search algorithms BFS/DFS/A* algorithm Like Maze problems, 4 connect etc			

Gaurav Deshmukh
Subject In-charge

Prof. Vikram Vyavhare
HOD

EXPERIMENT NO:-1

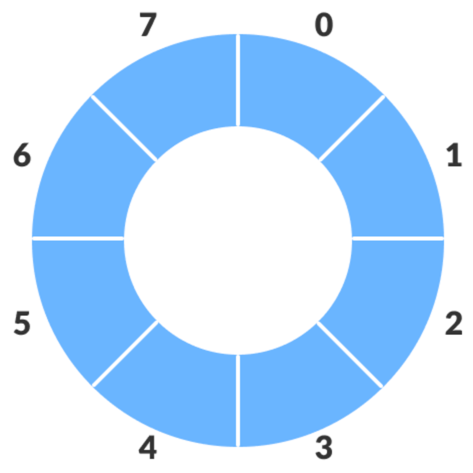
PART-A

AIM:-To implement of any one application of stack / Queue/Circular Queue

THEORY:-

- **How Circular Queue Works**

Circular Queue works by the process of circular increment i.e. when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.
Here, the circular increment is performed by modulo division with the queue size. That is,
if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0 (start of queue)



Circular queue representation

- **Circular Queue Operations**

The circular queue work as follows:

- two pointers FRONT and REAR
- FRONT track the first element of the queue
- REAR track the last elements of the queue
- initially, set value of FRONT and REAR to -1

1. Enqueue Operation

- check if the queue is full
- for the first element, set value of FRONT to 0
- circularly increase the REAR index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)
- add the new element in the position pointed to by REAR

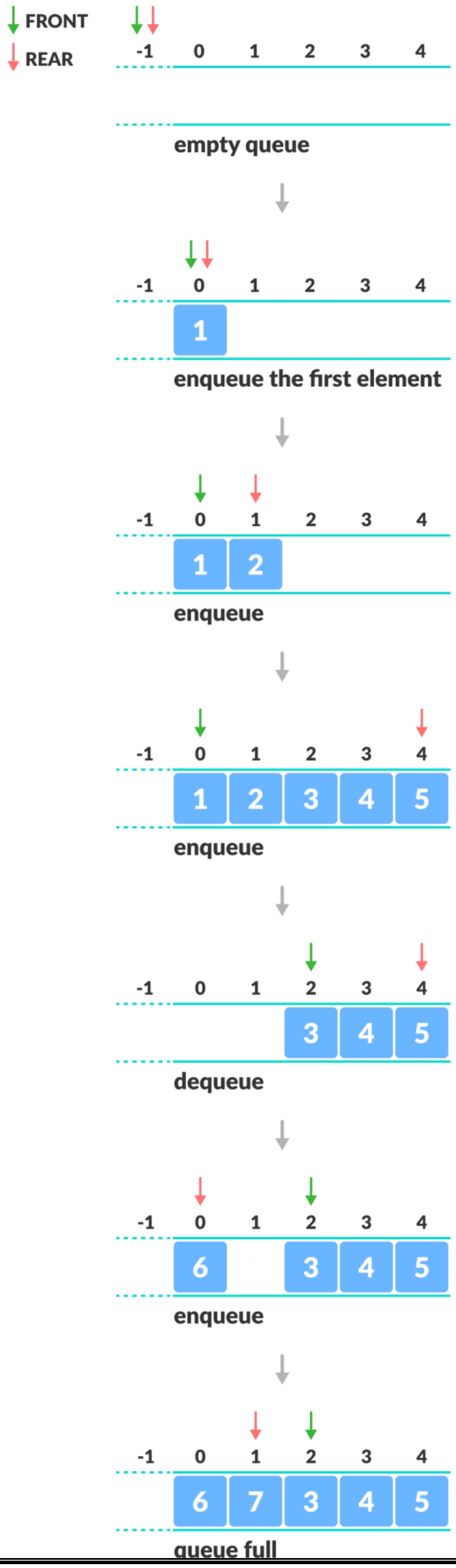
2. Dequeue Operation

- check if the queue is empty
- return the value pointed by FRONT
- circularly increase the FRONT index by 1
- for the last element, reset the values of FRONT and REAR to -1

However, the check for full queue has a new additional case:

- Case 1: FRONT = 0 && REAR == SIZE - 1
- Case 2: FRONT = REAR + 1

The second case happens when REAR starts from 0 due to circular increment and when its value is just 1 less than FRONT, the queue is full.

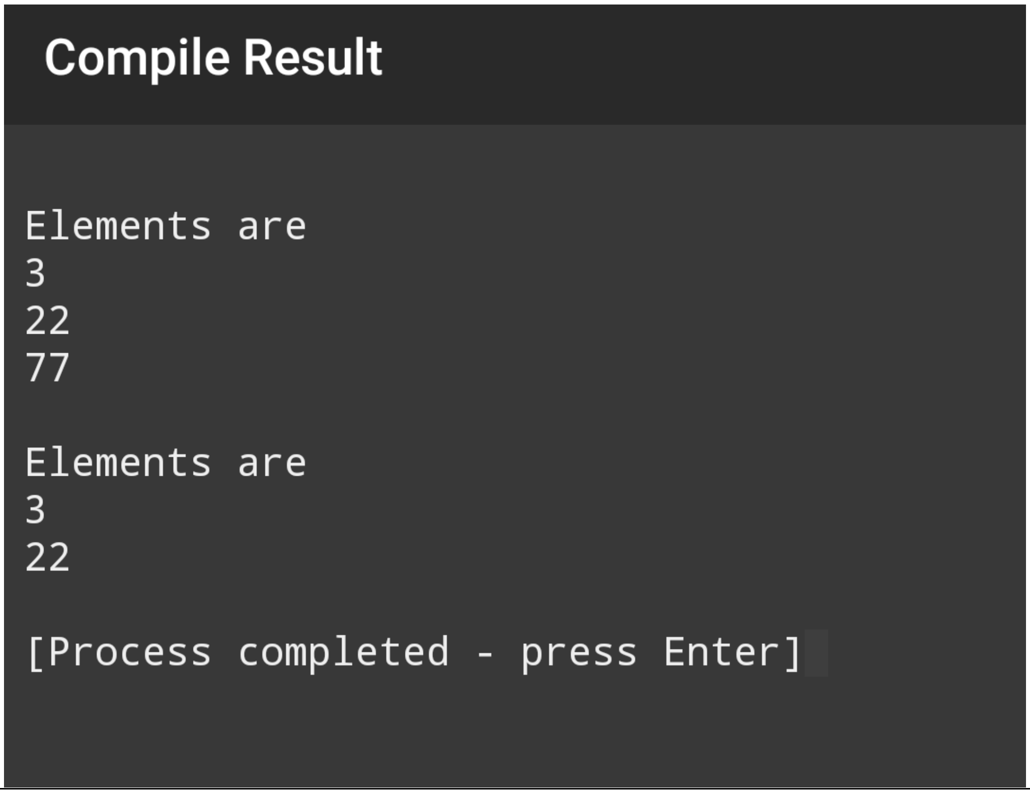

FRONT
REAR
-1 0 1 2 3 4
empty queue
1
enqueue the first element
1 2
enqueue
1 2 3 4 5
enqueue
3 4 5
dequeue
6 3 4 5
enqueue
6 7 3 4 5
queue full

Part-B

Program:-

```
//Umesh Mane
//Roll No 44
#include<stdio.h>
#define MAX_SIZE 5
int arr[MAX_SIZE],top=-1,i;
void push(int x){
if(top==(MAX_SIZE-1)){
printf("Stack is Overflow\n");
}else{
arr[++top] = x;
}
}
void pop(){
if(top<0){
printf("Stack is Underflow\n");
}else{
arr[top--];
}
}
void disp(){
printf("\nElements are\n");
for(i=0; i<=top; i++){
printf("%d\n",arr[i]);
}
}
int main(){
push(10);
push(20);
push(30);
disp();
pop();
disp();
return 0;
}
```

OUTPUT:


Compile Result

```
Elements are
3
22
77

Elements are
3
22

[Process completed - press Enter]
```

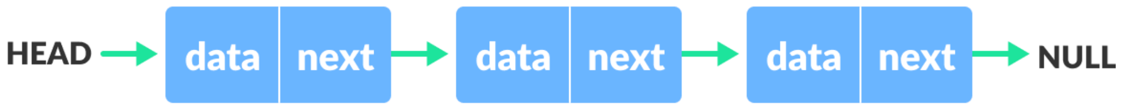
CONCLUSION:- We have understood the implementation of Queue using Array.

Experiment No: 02

Part-A

Title: To implement of operations on Linked Lists

Theory:
A linked list data structure includes a series of connected nodes. Here, each node store the data and the address of the next node. For example,



Linkedin

Data Structure
You have to start somewhere, so we give the address of the first node a special name called HEAD.
Also, the last node in the linked list can be identified because its next portion points to NULL.
You might have played the game Treasure Hunt, where each clue includes the information about the next clue. That is how the linked list operates.

- Representation of LinkedList

Let's see how each node of the LinkedList is represented. Each node consists:

- A data item
- An address of another node

We wrap both the data item and the next node reference in a struct as:

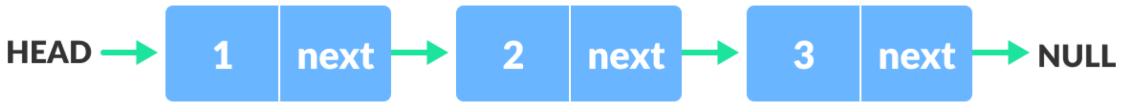
```
struct node
{
  int data;
  struct node *next;
};
```

Understanding the structure of a linked list node is the key to having a grasp on it.
Each struct node has a data item and a pointer to another struct node. Let us create a simple Linked List with three items to understand how this works.

```
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* Assign data values */
```

```
one->data = 1;
two->data = 2;
three->data=3;

/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;

/* Save address of first node in head */
head = one;
```

If you didn't understand any of the lines above, all you need is a refresher on pointers and structs.
In just a few steps, we have created a simple linked list with three nodes.



LinkedList Representation

The power of LinkedList comes from the ability to break the chain and rejoin it. E.g. if you wanted to put an element 4 between 1 and 2, the steps would be:

- Create a new struct node and allocate memory to it.
- Add its data value as 4
- Point its next pointer to the struct node containing 2 as the data value
- Change the next pointer of "1" to the node we just created.

Part-B

Program:-

```
//Umesh Mane
//Roll No 44
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
struct node
{
int data;
struct node *next;
};
struct node *start=NULL;
void create();
void insert_beg();
void insert_before();
void insert_end();
void insert_after();
void display();
void del_beg();
void del_end();
void del_pos();
void main()
{
int choice;
printf("\t\t**MAIN MENU**\n");
printf("\t---------------------------------\n");
printf("\t1.Create Linked List.\n\t2.Insert data at beginning.\n\t3.Insert data at end.\n\t4.Insert
new node before a given node.\n\t5.Insert new node after a given node.\n\t6.Delete 1st
node.\n\t7.Delete last node.\n\t8.Delete a given node.\n\t9.Display Linked List.\n\t10.Exit.\n");
do
{
printf("Enter choice:");
scanf("%d",&choice);
switch(choice)
{
case 1:
{
create();
break;
}
case 2:
{
insert_beg();
break;
}
case 3:
{
insert_end();
break;
}
case 4:
{
insert_before();
break;
}
case 5:
{
insert_after();
break;
}
case 6:
{
del_beg();
break;
}
case 7:
```

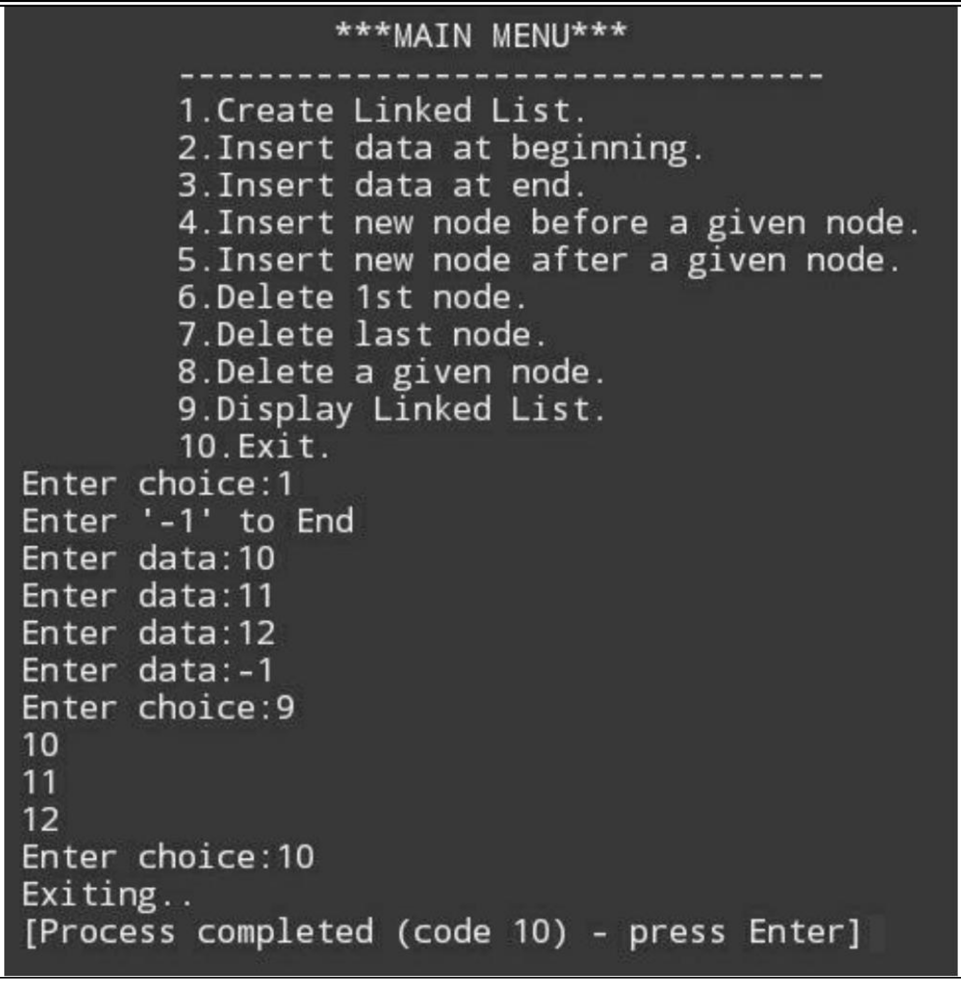
```
{
del_end();
break;
}
case 8:
{
del_pos();
break;
}
case 9:
{
display();
break;
}
case 10:
{
printf("Exiting..");
break;
}
default :
{
printf("Enter Valid Option!
\n");
}
}
}while(choice!=10);
getch();
}
void create()
{
struct node *new_node,*ptr;
int val;
printf("Enter '-1' to End");
printf("\nEnter data:");
scanf("%d",&val);
while(val!=-1)
{
new_node=(struct node *)malloc(sizeof(struct node));
new_node->data=val;
if(start==NULL)
{
start=new_node;
new_node->next=NULL;
}
else
{
ptr=start;
```

```
while(ptr->next!=NULL)
ptr=ptr->next;
ptr->next=new_node;
new_node->next=NULL;
}
printf("Enter data:");
scanf("%d",&val);
}
}
void insert_beg()
{
struct node *new_node;
new_node=(struct node *)malloc(sizeof(struct node));
printf("Enter data:");
scanf("%d",&new_node->data);
new_node->next=start;
start=new_node;
}
void display()
{
struct node *ptr;
ptr=start;
while(ptr!=NULL)
{
printf("%d\n",ptr->data);
ptr=ptr->next;
}
}
void insert_end()
{
struct node *new_node,*ptr;
new_node=(struct node*)malloc(sizeof(struct node));
printf("Enter data:");
scanf("%d",&new_node->data);
new_node->next=NULL;
ptr=start;
while(ptr->next!=NULL)
ptr=ptr->next;
ptr->next=new_node;
}
void insert_before()
{
struct node *ptr,*preptr,*new_node;
int num;
printf("Enter node before which you want to enter new node:");
scanf("%d",&num);
if(start->data==num)
```

```
{
insert_beg();
}
else
{
new_node=(struct node *)malloc(sizeof(struct node));
printf("Enter data:");
scanf("%d",&new_node->data);
ptr=start;
while(ptr->data!=num)
{
preptr=ptr;
ptr=ptr->next;
}
preptr->next=new_node;
new_node->next=ptr;
}
}
void insert_after()
{
struct node *ptr,*preptr,*new_node,*eptr;
int num;
printf("Enter node after which you want to enter new node:");
scanf("%d",&num);
eptr=start;
while(eptr->next!=NULL)
{
eptr=eptr->next;
}
if(eptr->data==num)
{
insert_end();
}
else
{
new_node=(struct node *)malloc(sizeof(struct node));
printf("Enter data:");
scanf("%d",&new_node->data);
ptr=start;
preptr=ptr;
if(start->data==num)
{
ptr=ptr->next;
}
else
{
while(preptr->data != num)
{
preptr=ptr;
ptr=ptr->next;
}
}
preptr->next=new_node;
new_node->next=ptr;
}
}
void del_beg()
{
struct node *ptr;
if(start==NULL)
{
printf("UNDERFLOW\n");
}
else
{
ptr=start;
start=start->next;
free(ptr);
printf("Deleted!\n");
}
}
void del_end()
{
struct node *ptr,*preptr;
if(start==NULL)
{
printf("UNDERFLOW\n");
}
else
{
ptr=start;
preptr=ptr;
while(ptr->next!=NULL)
{
preptr=ptr;
ptr=ptr->next;
}
preptr->next=NULL;
free(ptr);
printf("Deleted!\n");
}
}
void del_pos()
{
```

```
struct node *ptr,*preptr;
if(start==NULL)
{
printf("UNDERFLOW\n");
}
else
{
int val;
printf("Enter the node to be delete:");
scanf("%d",&val);
if(start->data==val)
{
del_beg();
}
else
{
ptr=start;
preptr=ptr;
while(ptr->data!=val)
{
preptr=ptr;
ptr=ptr->next;
}
preptr->next=ptr->next;
free(ptr);
printf("Deleted!\n");
}
}
}
```

OUTPUT:



```
                 ***MAIN MENU***
        ---------------------------------
        1.Create Linked List.
        2.Insert data at beginning.
        3.Insert data at end.
        4.Insert new node before a given node.
        5.Insert new node after a given node.
        6.Delete 1st node.
        7.Delete last node.
        8.Delete a given node.
        9.Display Linked List.
        10.Exit.
Enter choice:1
Enter '-1' to End
Enter data:10
Enter data:11
Enter data:12
Enter data:-1
Enter choice:9
10
11
12
Enter choice:10
Exiting..
[Process completed (code 10) - press Enter]
```

CONCLUSION:- We have understood the implementation of operations on a Linked List.

EXPERIMENT NO:-3

Part-A

AIM: -To implement of stack and queue using Link list.

THEORY: -

Stack is a type of queue that in practice is implemented as an area of memory that holds all local variables and parameters used by any function, and remembers the order in which functions are called so that function returns occur correctly. Each time a function is called, its local variables and parameters are "pushed onto" the stack. When the function returns,these locals and parameters are "popped." Because of this, the size of a program's stack fluctuates constantly as the program is running, but it has some maximum size. stack is as a last in, first out (LIFO) abstract data type and linear data structure. Linked list is a data structure consisting of a group of nodes which together represent a sequence. Here we need to apply the application of linked list to perform basic operations of stack.

In a Queue data structure, we maintain two pointers, *front* and *rear*. The *front* points the first item of queue and *rear* points to last item.
enQueue() This operation adds a new node after *rear* and moves *rear* to the next node.
deQueue() This operation removes the front node and moves *front* to the next node.

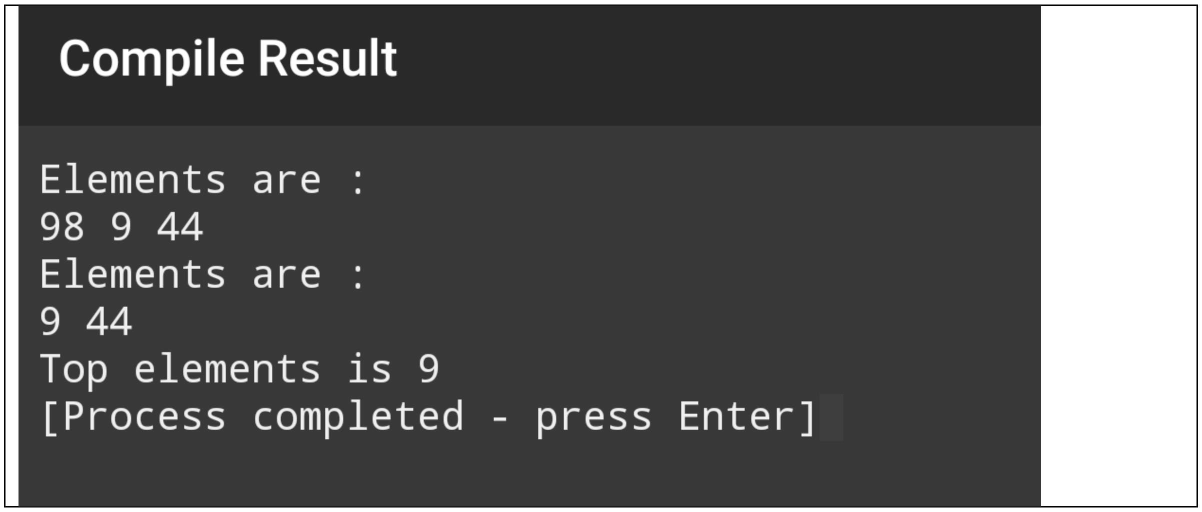
Part-B

Program:-

```
//Umesh Mane
//Roll No 44
[11:41 AM, 12/6/2020] Babu: #include<stdio.h>
#include<stdlib.h>
struct Node{
int data;
struct Node *link;
};
struct Node *top = NULL;
```

```
void push(int value){
struct Node temp = (struct Node)malloc(sizeof(struct Node));
temp->data = value;
temp->link = top;
top = temp;
}
void pop(){
struct Node *del = top;
if(top!=NULL){
top = top->link;
free(del);
}else{
printf("Stack Underflow\n");
}
}
void TOP(){
struct Node *t = top;
printf("Top elements is %d",t->data);
}
void disp(){
struct Node *head = top;
printf("Elements are : \n");
while(head!=NULL){
printf("%d ",head->data);
head = head->link;
}
printf("\n");
}
int main(){
push(10);
push(20);
push(30);
disp();
pop();
disp();
TOP();
return 0;
}
```

OUTPUT:



Compile Result

```
Elements are :
98 9 44
Elements are :
9 44
Top elements is 9
[Process completed - press Enter]
```

CONCLUSION:- We have understood the implementation of stack using Linked List.

EXPERIMENT NO:-4

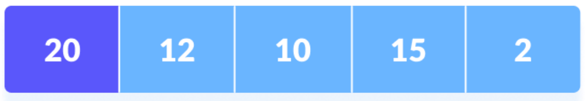
Part-A

AIM:-Toimplement and analyse of selection sort/insertion sort.

THEORY: -

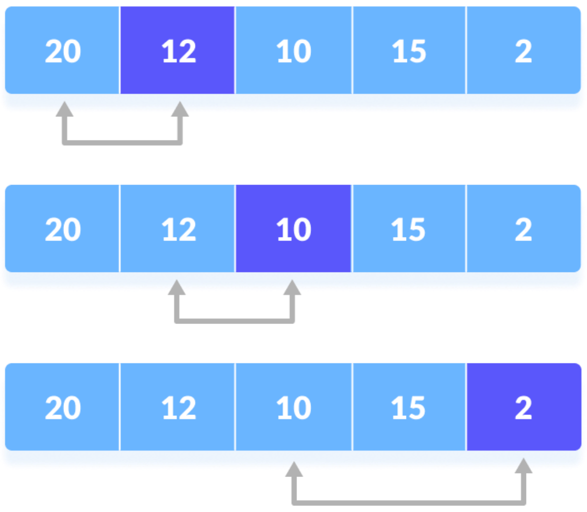
Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

1. How Selection Sort Works?



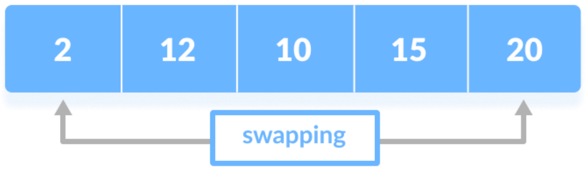
2. Setthe first element as minimum. Select first element as minimum
3. Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.

   Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes



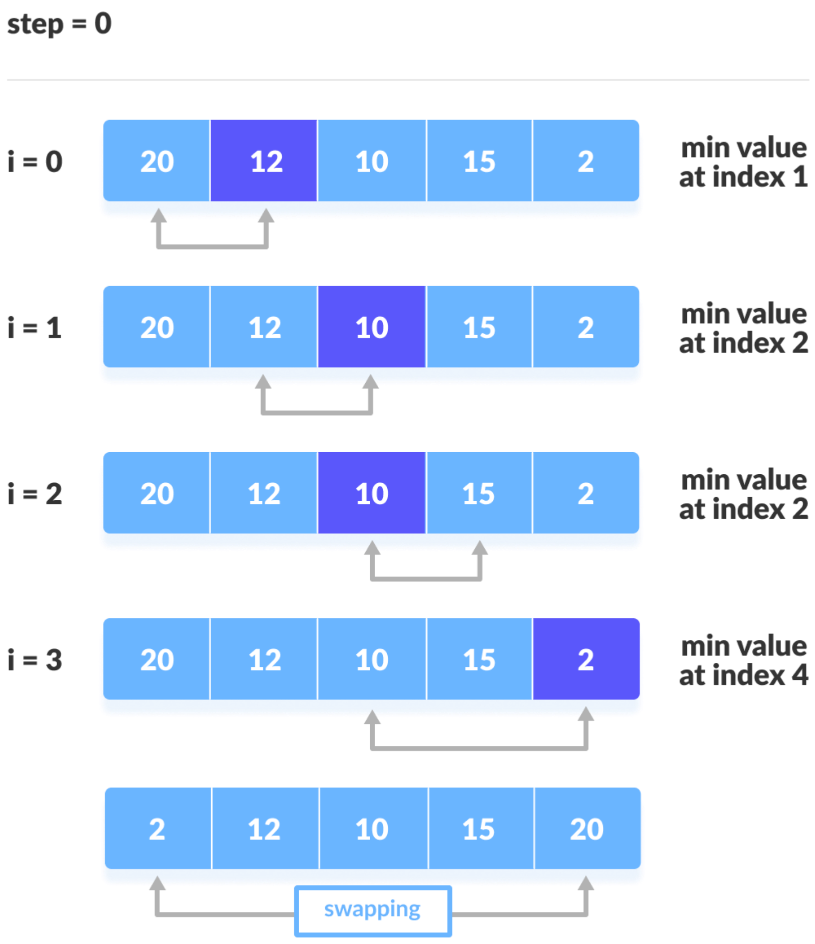
   on until the last element. Compare minimum with the remaining elements

4. After each iteration, minimum is placed in the front of the unsorted list.

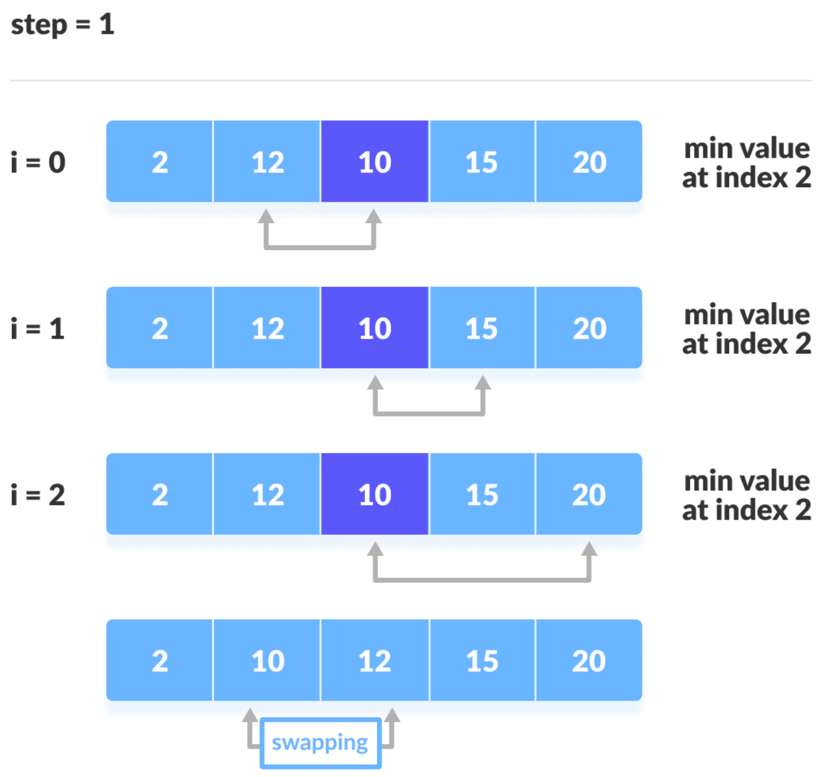


Swap the first with minimum

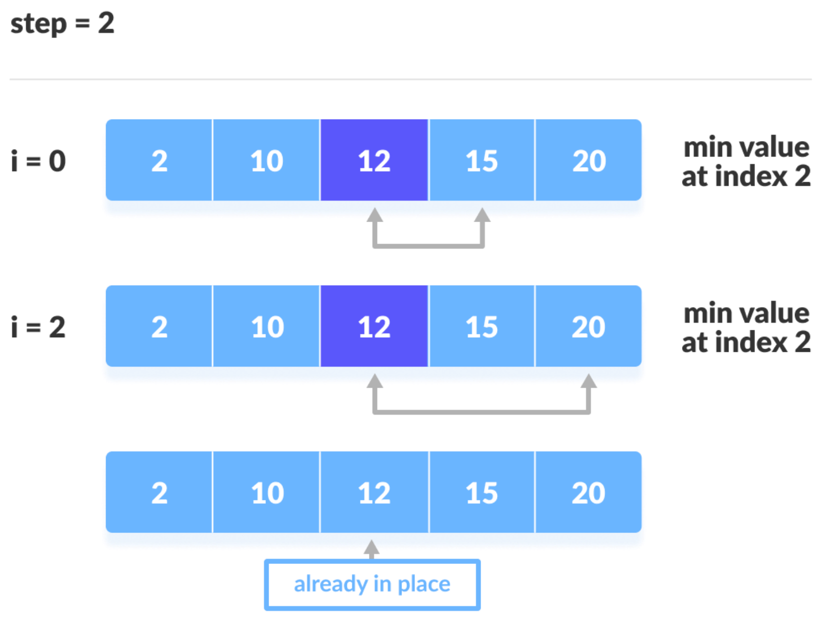
5. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.



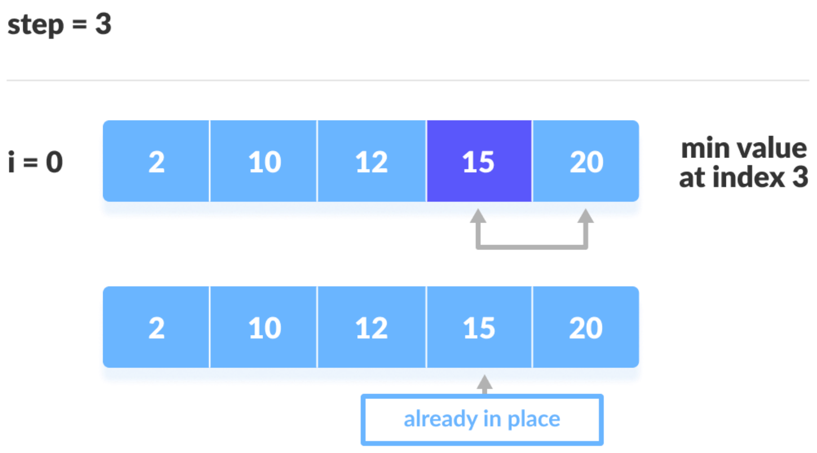
The first



iteration The second



iteration The third



iteration The fourth iteration

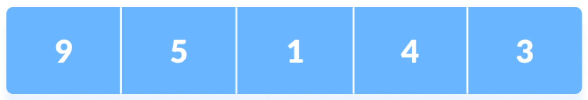
Selection Sort Algorithm

```
selectionSort(array, size)
  repeat (size - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
    if element <currentMinimum
      set element as new minimum
  swap minimum with first unsorted position
end selectionSort
```

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

How Insertion Sort Works?

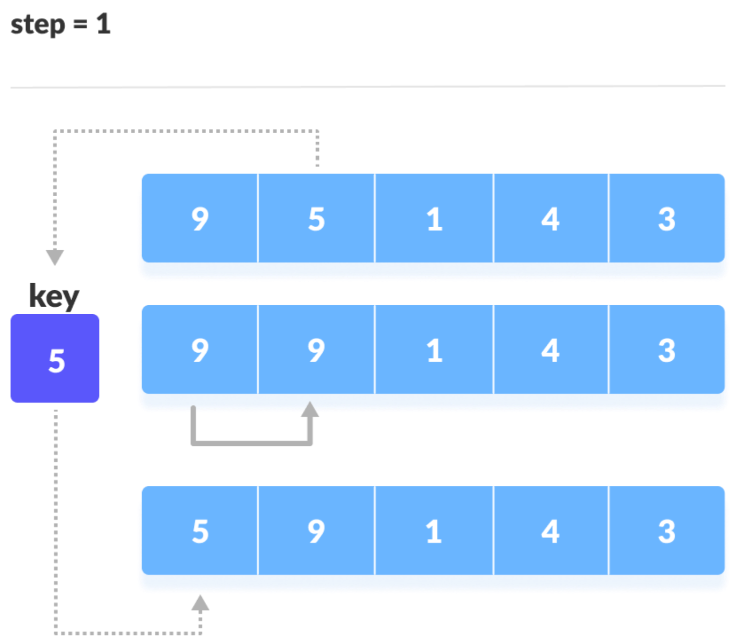
Suppose we need to sort the following array.



Initial array

1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

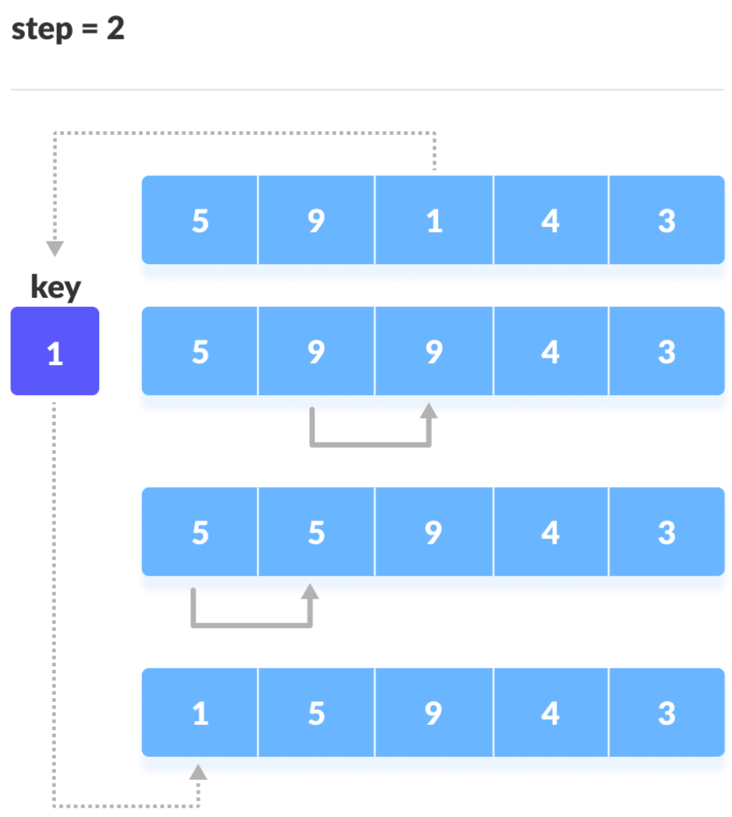
   Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.



If the first element is greater than key, then key is placed in front of the first element.

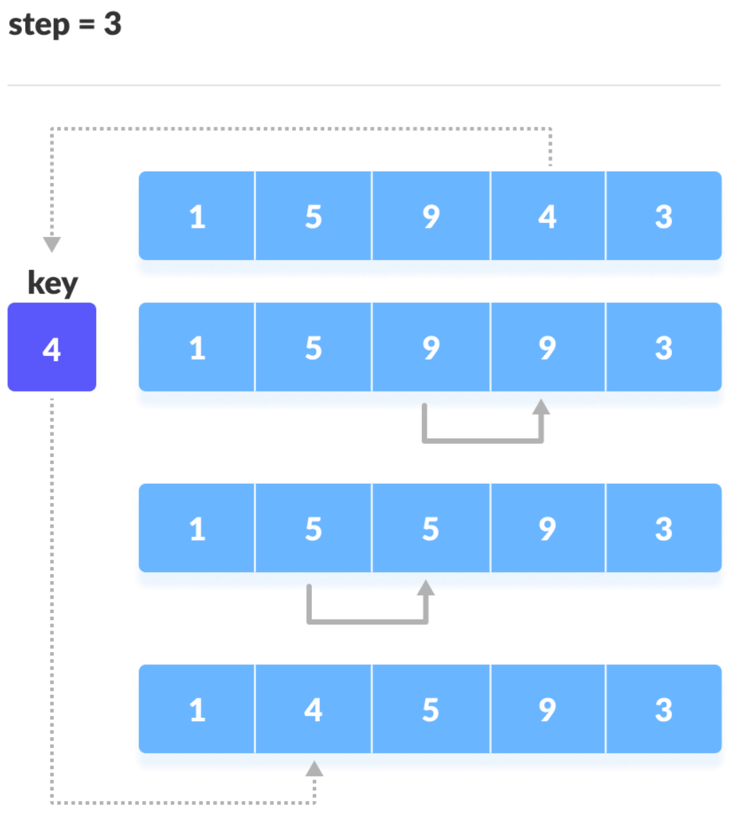
2. Now, the first two elements are sorted.

   Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.

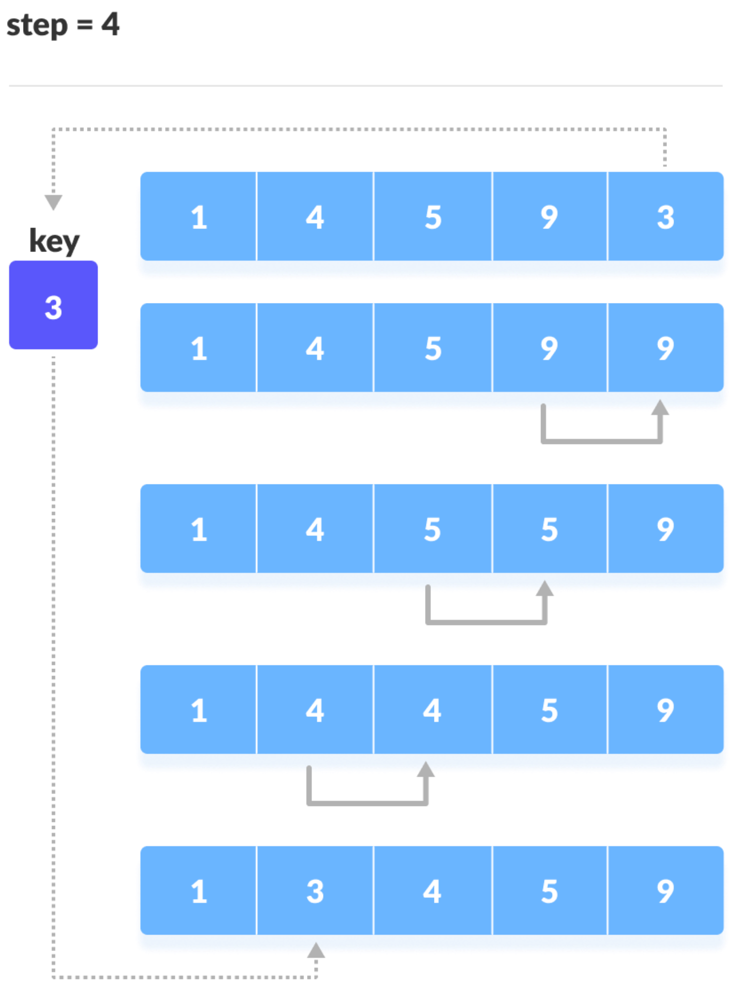


Place 1 at the beginning

3. Similarly, place every unsorted element at its correct position.



Place 4 behind



1 Place 3 behind 1 and the array is sorted

Insertion Sort Algorithm

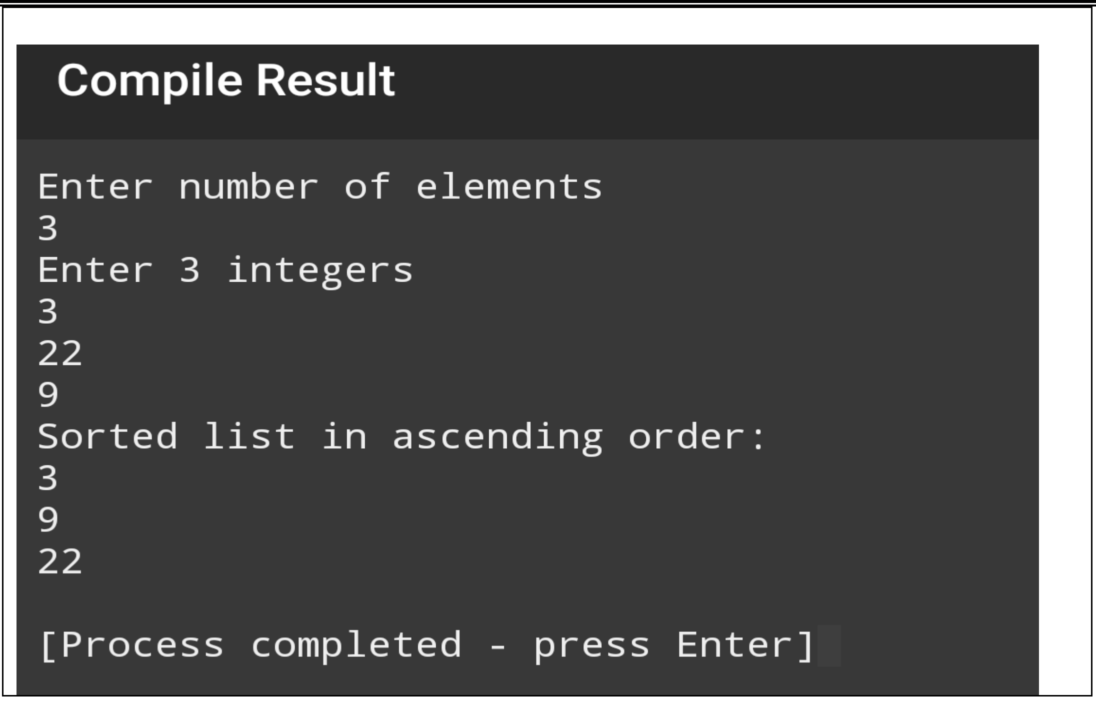
```
insertionSort(array)
  mark first element as sorted
  for each unsorted element X
    'extract' the element X
    for j <- lastSortedIndex down to 0
      if current element j > X
        move sorted element to the right by 1
    break loop and insert X here
end insertionSort
```

Part-B

Program of selection sort:-

```
//Umesh Mane
//Roll No 44
#include <stdio.h>
int main()
{
int array[100], n, c, d, position, swap;
printf("Enter number of elements\n");
scanf("%d", &n);
printf("Enter %d integers\n", n);
for (c = 0; c < n; c++)
scanf("%d", &array[c]);
for (c = 0; c < (n - 1); c++)
{
position = c;
for (d = c + 1; d < n; d++)
{
if (array[position] > array[d])
position = d;
}
if (position != c)
{
swap = array[c];
array[c] = array[position];
array[position] = swap;
}
}
printf("Sorted list in ascending order:\n");
for (c = 0; c < n; c++)
printf("%d\n", array[c]);
return 0;
}
```

OUTPUT:

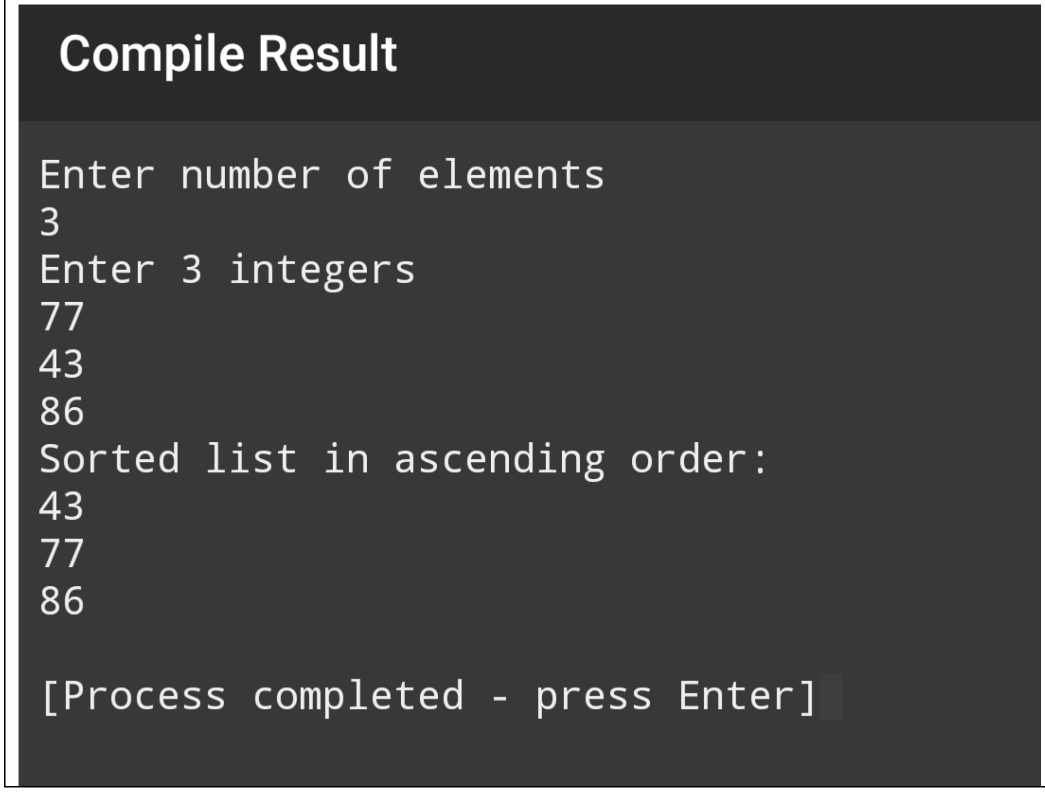


```
Compile Result

Enter number of elements
3
Enter 3 integers
3
22
9
Sorted list in ascending order:
3
9
22

[Process completed - press Enter]
```

Program of insertion sort:-

```
//Umesh Mane
//Roll No 44
#include <stdio.h>
int main()
{
int n, array[1000], c, d, t;
printf("Enter number of elements\n");
scanf("%d", &n);
printf("Enter %d integers\n", n);
for (c = 0; c < n; c++)
scanf("%d", &array[c]);
for (c = 1 ; c <= n - 1; c++) {
d = c;
while ( d > 0 && array[d-1] > array[d]) {
t = array[d];
array[d] = array[d-1];
array[d-1] = t;
d--;
}
}
printf("Sorted list in ascending order:\n");
for (c = 0; c <= n - 1; c++) {
printf("%d\n", array[c]);
}
return 0;
}
```

OUTPUT:



Compile Result

```
Enter number of elements
3
Enter 3 integers
77
43
86
Sorted list in ascending order:
43
77
86

[Process completed - press Enter]
```

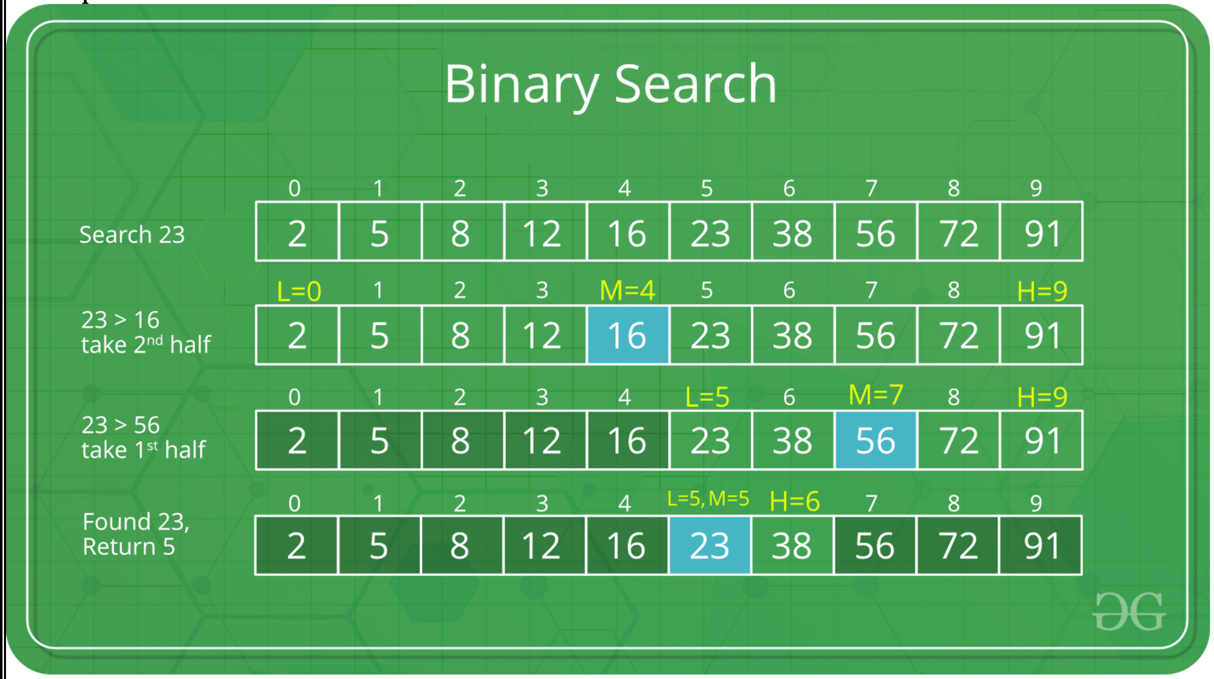
CONCLUSION:- We have understood the program of selection and insertion sort.

EXPERIMENT NO:-5

Part-A

AIM: - To implement Binary search/ merge sort/quick sort.

THEORY: -
Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.
Example :



The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See the following C implementation for details.

```
MergeSort(arr[], l,  r)
If r > l
    1. Find the middle point to divide the array into two halves:
           middle m = (l+r)/2
    2. Call mergeSort for first half:
           Call mergeSort(arr, l, m)
    3. Call mergeSort for second half:
           Call mergeSort(arr, m+1, r)
    4. Merge the two halves sorted in step 2 and 3:
           Call merge(arr, l, m, r)
```

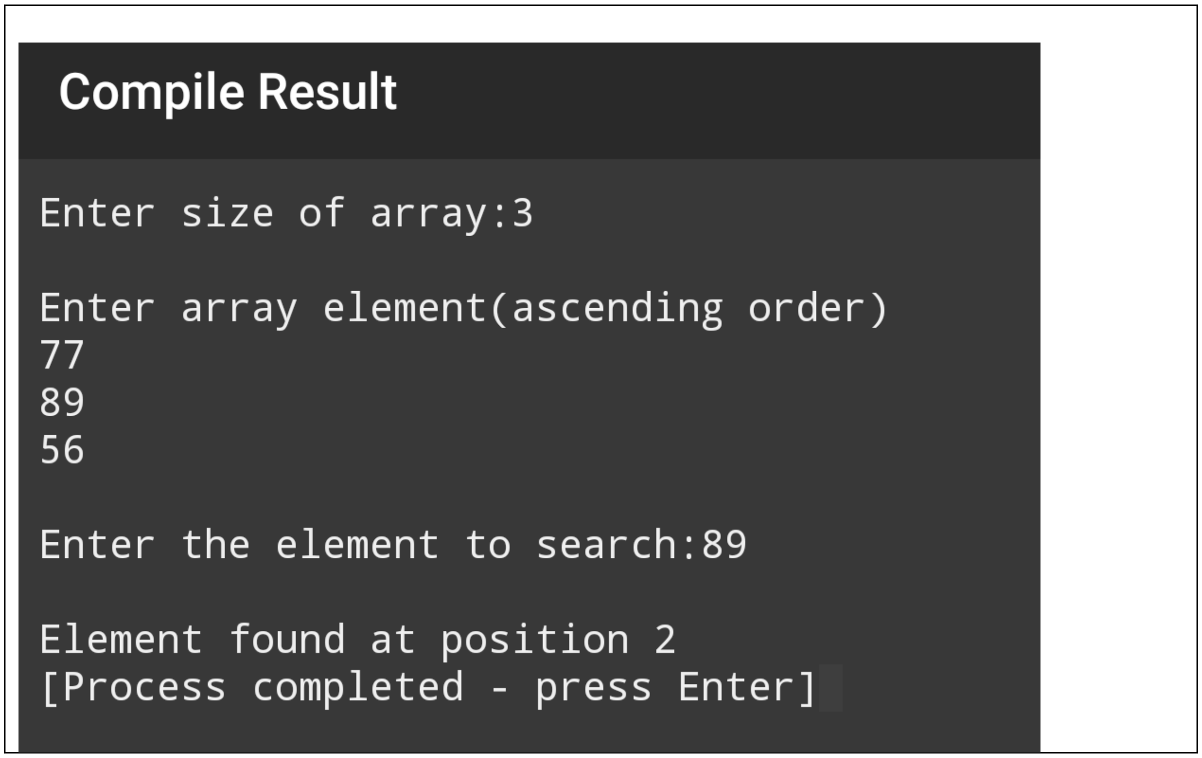
The following diagram from wikipedia shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.

Part-B

Program:-

```
//Umesh Mane
//Roll No 44
#include<stdio.h>
int main(){
int arr[50],i,n,x,flag=0,first,last,mid;
printf("Enter size of array:");
scanf("%d",&n);
printf("\nEnter array element(ascending order)\n");
for(i=0;i<n;++i)
scanf("%d",&arr[i]);
printf("\nEnter the element to search:");
scanf("%d",&x);
first=0;
last=n-1;
while(first<=last)
{
mid=(first+last)/2;
if(x==arr[mid]){
flag=1;
break;
}
else
if(x>arr[mid])
first=mid+1;
else
last=mid-1;
}
if(flag==1)
printf("\nElement found at position %d",mid+1);
else
printf("\nElement not found");
return 0;
}
```

OUTPUT:



Compile Result

```
Enter size of array:3

Enter array element(ascending order)
77
89
56

Enter the element to search:89

Element found at position 2
[Process completed - press Enter]
```

CONCLUSION:- We have understood the implementation of Binary Search.

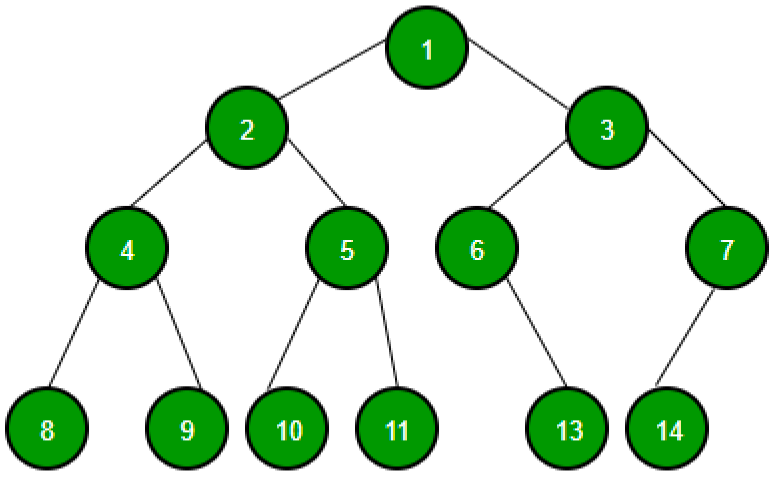
EXPERIMENT NO:-6

AIM: - To implementoperations on Binary Tree/Binary Search Tree/ Heap.

THEORY: -

Binary Tree Data Structure

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



A Binary Tree node contains following parts.

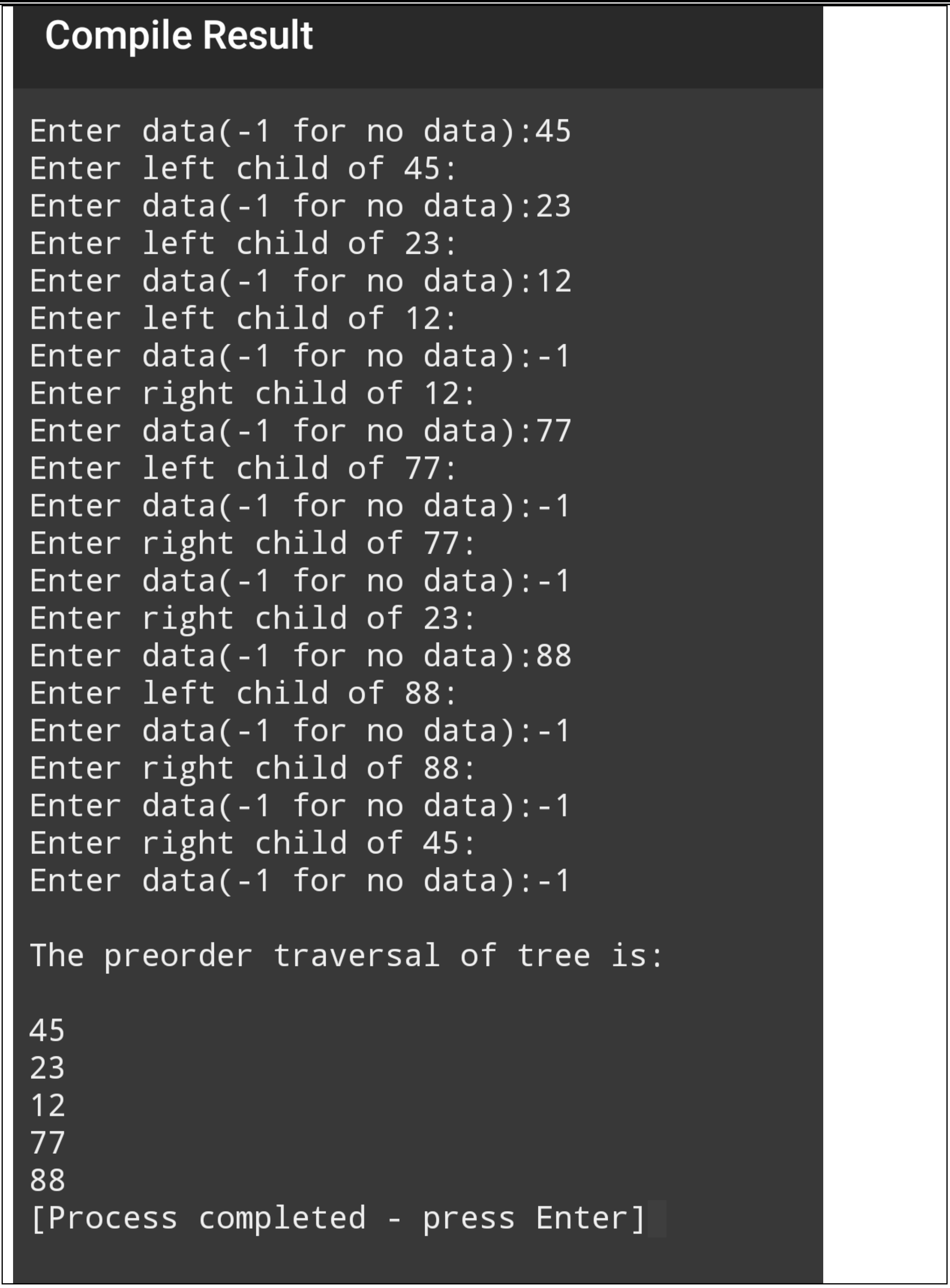
1. **Data**
2. **Pointer to left child**
3. **Pointer to right child**

Part-B

Program:-

```
//Umesh Mane
//Roll No 44
#include<stdio.h>
#include<stdlib.h>
typedef struct node
{
int data;
struct node *left;
struct node *right;
} node;
node *create()
{
node *p;
int x;
printf("Enter data(-1 for no data):");
scanf("%d",&x);
if(x==-1)
return NULL;
p=(node*)malloc(sizeof(node));
p->data=x;
printf("Enter left child of %d:\n",x);
p->left=create();
printf("Enter right child of %d:\n",x);
p->right=create();
return p;
}
void preorder(node *t)
{
if(t!=NULL)
{
printf("\n%d",t->data);
preorder(t->left);
preorder(t->right);
}
}
int main()
{
node *root;
root=create();
printf("\nThe preorder traversal of tree is:\n");
preorder(root);
return 0;
}
```

OUTPUT:



Compile Result

```
Enter data(-1 for no data):45
Enter left child of 45:
Enter data(-1 for no data):23
Enter left child of 23:
Enter data(-1 for no data):12
Enter left child of 12:
Enter data(-1 for no data):-1
Enter right child of 12:
Enter data(-1 for no data):77
Enter left child of 77:
Enter data(-1 for no data):-1
Enter right child of 77:
Enter data(-1 for no data):-1
Enter right child of 23:
Enter data(-1 for no data):88
Enter left child of 88:
Enter data(-1 for no data):-1
Enter right child of 88:
Enter data(-1 for no data):-1
Enter right child of 45:
Enter data(-1 for no data):-1

The preorder traversal of tree is:

45
23
12
77
88
[Process completed - press Enter]
```

CONCLUSION:- We have understood the implementation of Binary Tree.

EXPERIMENT NO:-7

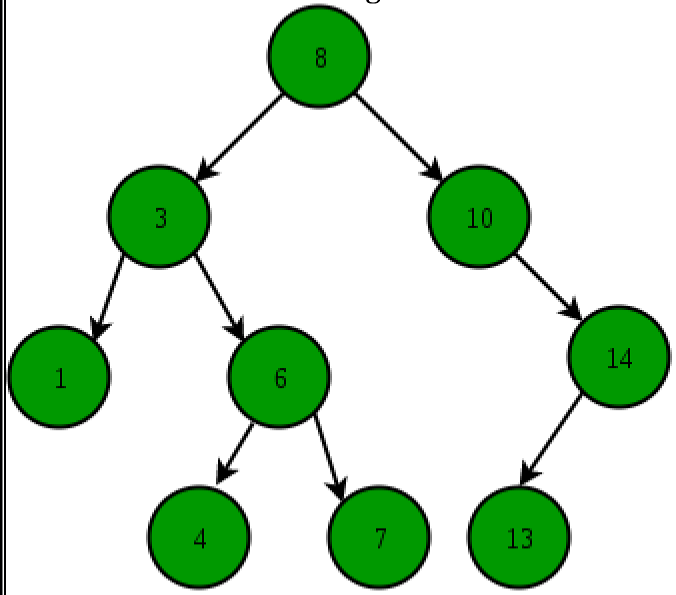
Part-A

AIM: - To implement Greedy method algorithms Prim's/ Kruskal's algorithm.

THEORY: -

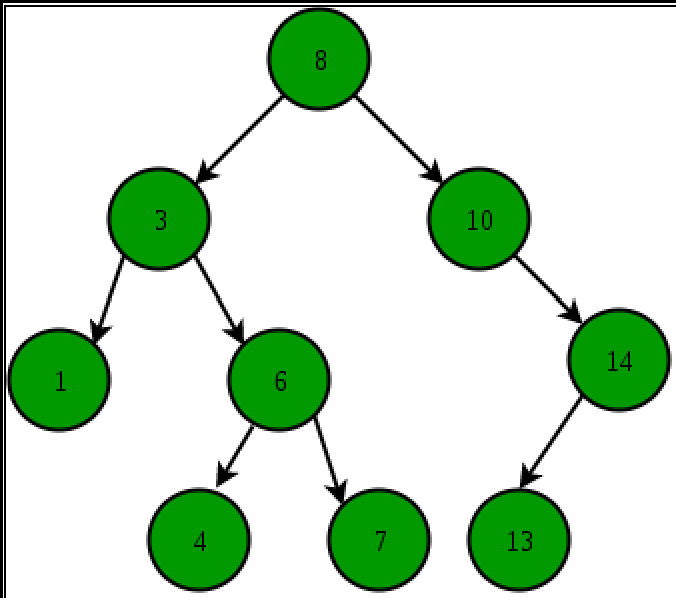
Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



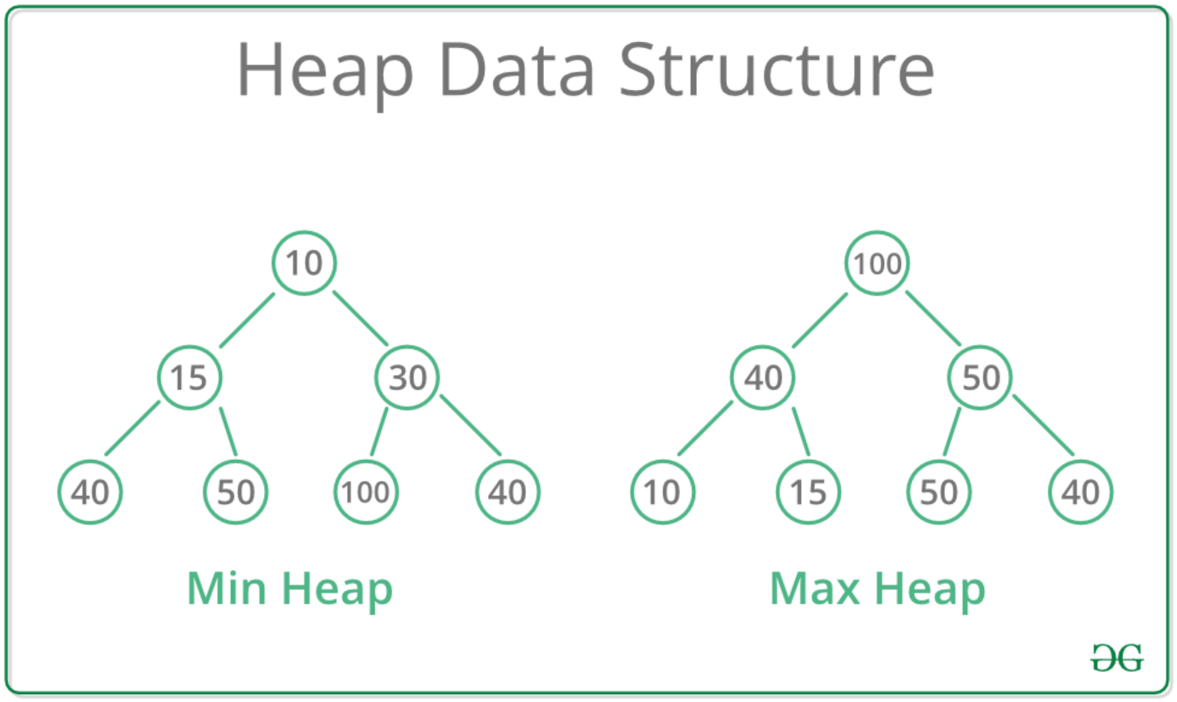
Binary Search Tree is a node-based binary tree data structure which has the following properties:

- **The left subtree of a node contains only nodes with keys lesser than the node's key.**
- **The right subtree of a node contains only nodes with keys greater than the node's key.**
- **The left and right subtree each must also be a binary search tree.**



A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:

1. **Max-Heap: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.**
2. **Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.**



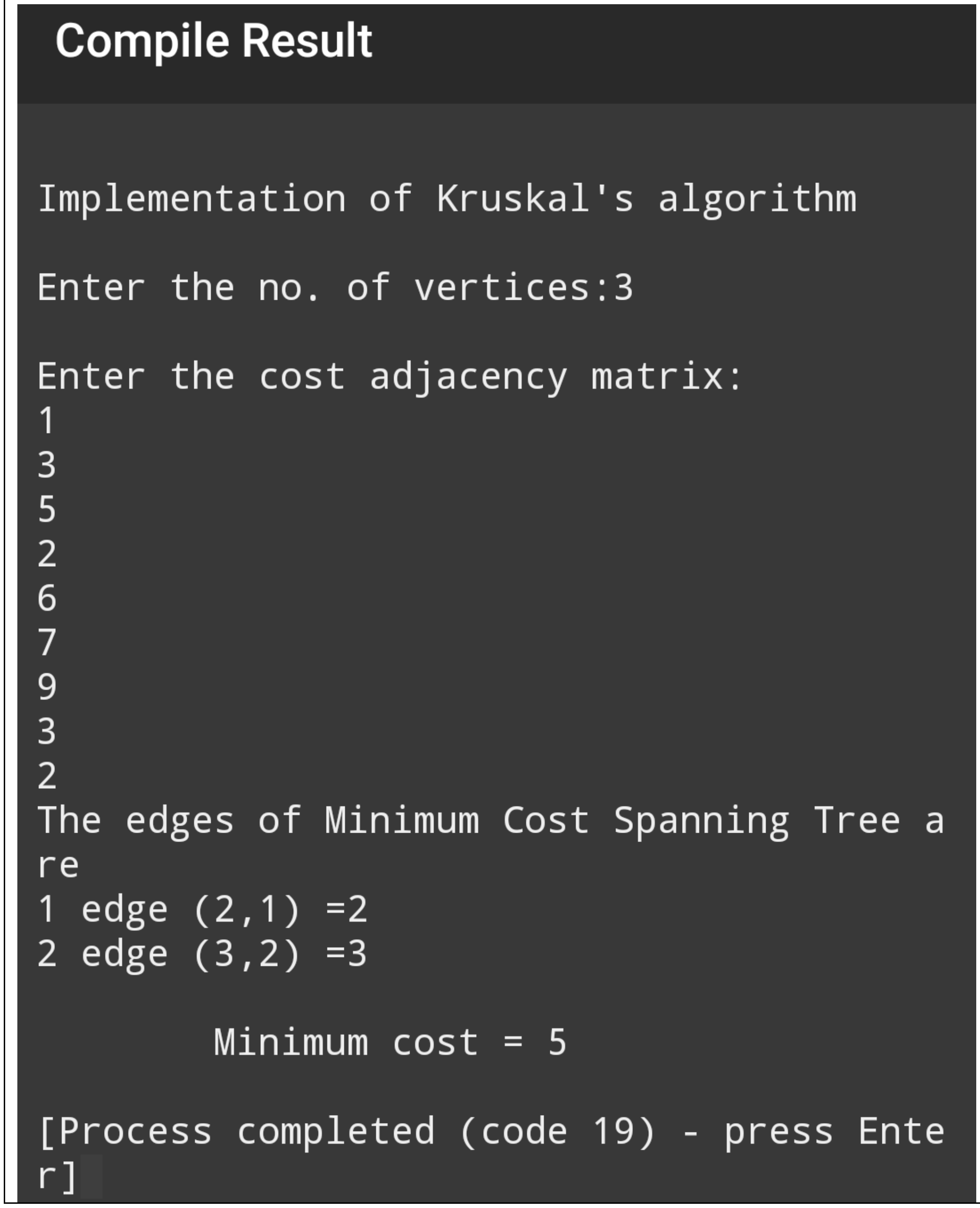
Part-B

Program:-

```
//Umesh Mane
//Roll No 44
#include<stdio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
void main()
{
printf("\nImplementation of Kruskal's algorithm\n");
printf("\nEnter the no. of vertices:");
scanf("%d",&n);
printf("\nEnter the cost 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;
}
}
printf("The edges of Minimum Cost Spanning Tree are\n");
while(ne < n)
{
for(i=1,min=999;i<=n;i++)
{
for(j=1;j <= n;j++)
{
if(cost[i][j] < min)
{
min=cost[i][j];
a=u=i;
b=v=j;
}
}
}
u=find(u);
v=find(v);
if(uni(u,v))
{
printf("%d edge (%d,%d) =%d\n",ne++,a,b,min);
mincost +=min;
}
cost[a][b]=cost[b][a]=999;
}
printf("\n\tMinimum cost = %d\n",mincost);
}
int find(int i)
{
while(parent[i])
i=parent[i];
return i;
}
int uni(int i,int j)
{
if(i!=j)
{
```

```
parent[j]=i;
return 1;
}
return 0;
}
```

OUTPUT:



```
Compile Result

Implementation of Kruskal's algorithm

Enter the no. of vertices:3

Enter the cost adjacency matrix:
1
3
5
2
6
7
9
3
2
The edges of Minimum Cost Spanning Tree a
re
1 edge (2,1) =2
2 edge (3,2) =3

        Minimum cost = 5

[Process completed (code 19) - press Ente
r]
```

CONCLUSION:- We have understood the kruskal's Algoritm for minimum spanning

EXPERIMENT NO:-8

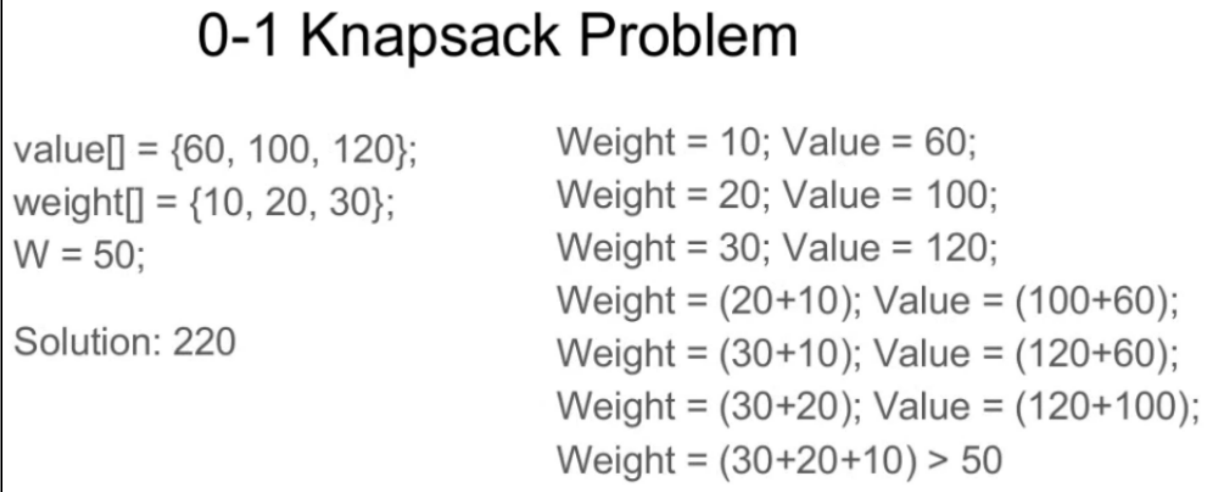
Part-A

AIM: - To implement Dynamic programming approach algorithms knapsack/Traveling salespersons problem.

THEORY: -

0-1 Knapsack problem:-

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don't pick it (0-1 property).



0-1 Knapsack Problem

value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50

Travelling salesman problem:-

A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

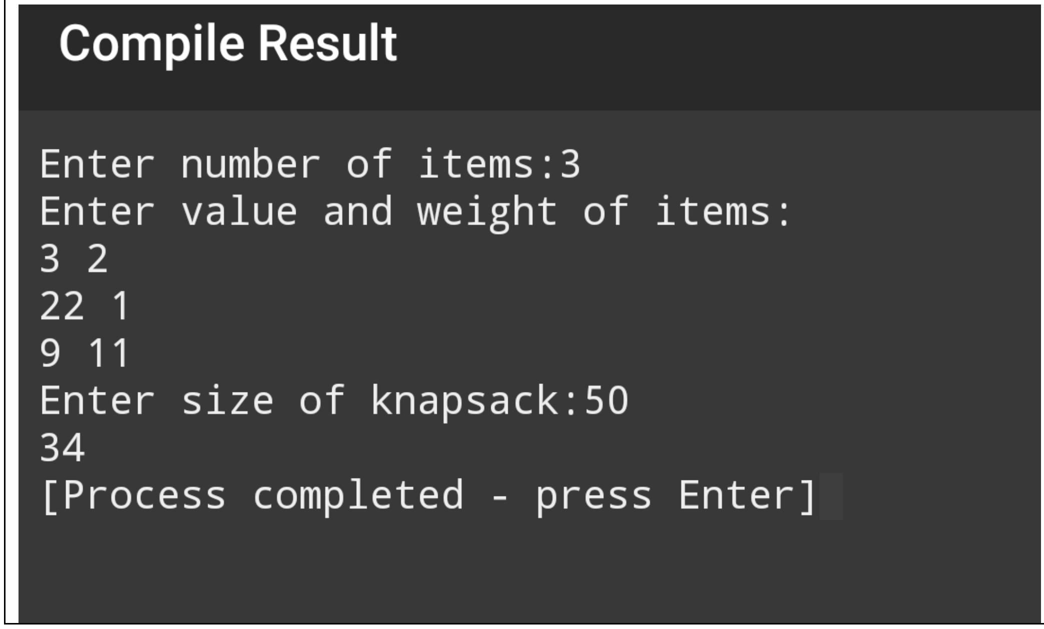
Part-B

Program:-

//Umesh Mane

```
//Roll No 44
#include<stdio.h>
int max(int a, int b) { return (a > b)? a : b; }
int knapSack(int W, int wt[], int val[], int n)
{
int i, w;
int K[n+1][W+1];
for (i = 0; i <= n; i++)
{
for (w = 0; w <= W; w++)
{
if (i==0 || w==0)
K[i][w] = 0;
else if (wt[i-1] <= w)
K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
else
K[i][w] = K[i-1][w];
}
}
return K[n][W];
}
int main()
{
int i, n, val[20], wt[20], W;
printf("Enter number of items:");
scanf("%d", &n);
printf("Enter value and weight of items:\n");
for(i = 0;i < n; ++i){
scanf("%d%d", &val[i], &wt[i]);
}
printf("Enter size of knapsack:");
scanf("%d", &W);
printf("%d", knapSack(W, wt, val, n));
return 0;
}
```

OUTPUT:



```
Enter number of items:3
Enter value and weight of items:
3 2
22 1
9 11
Enter size of knapsack:50
34
[Process completed - press Enter]
```

CONCLUSION:- We have understood the program of knapsack 0-1.

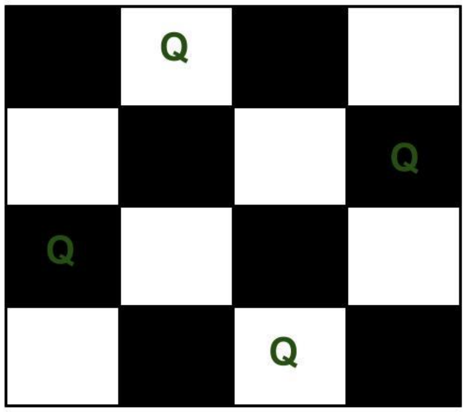
EXPERIMENT NO:-10

Part-A

AIM: - To implement Backtracking & branch and bound technique : N queens problem/15 puzzle problem

THEORY: -

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for above 4 queen solution.

{ 0, 1, 0, 0}
{ 0, 0, 0, 1}
{ 1, 0, 0, 0}
{ 0, 0, 1, 0}

15 puzzle problem:-

The 15 Puzzle is a sliding puzzle that consists of a 4 by 4 frame of numbered square tiles in an arbitrary ordering with one space. The objective of the puzzle is to place the tiles in order, as shown in the figure below, by making sliding moves that use the empty space.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

The 15 Puzzle consists of 15 squares numbered from 1 to 15 that are placed in a 4 by 4 box with one empty position. The objective of the puzzle is to reposition the squares by sliding them one at a time into a configuration with the numbers in order.

Part-B

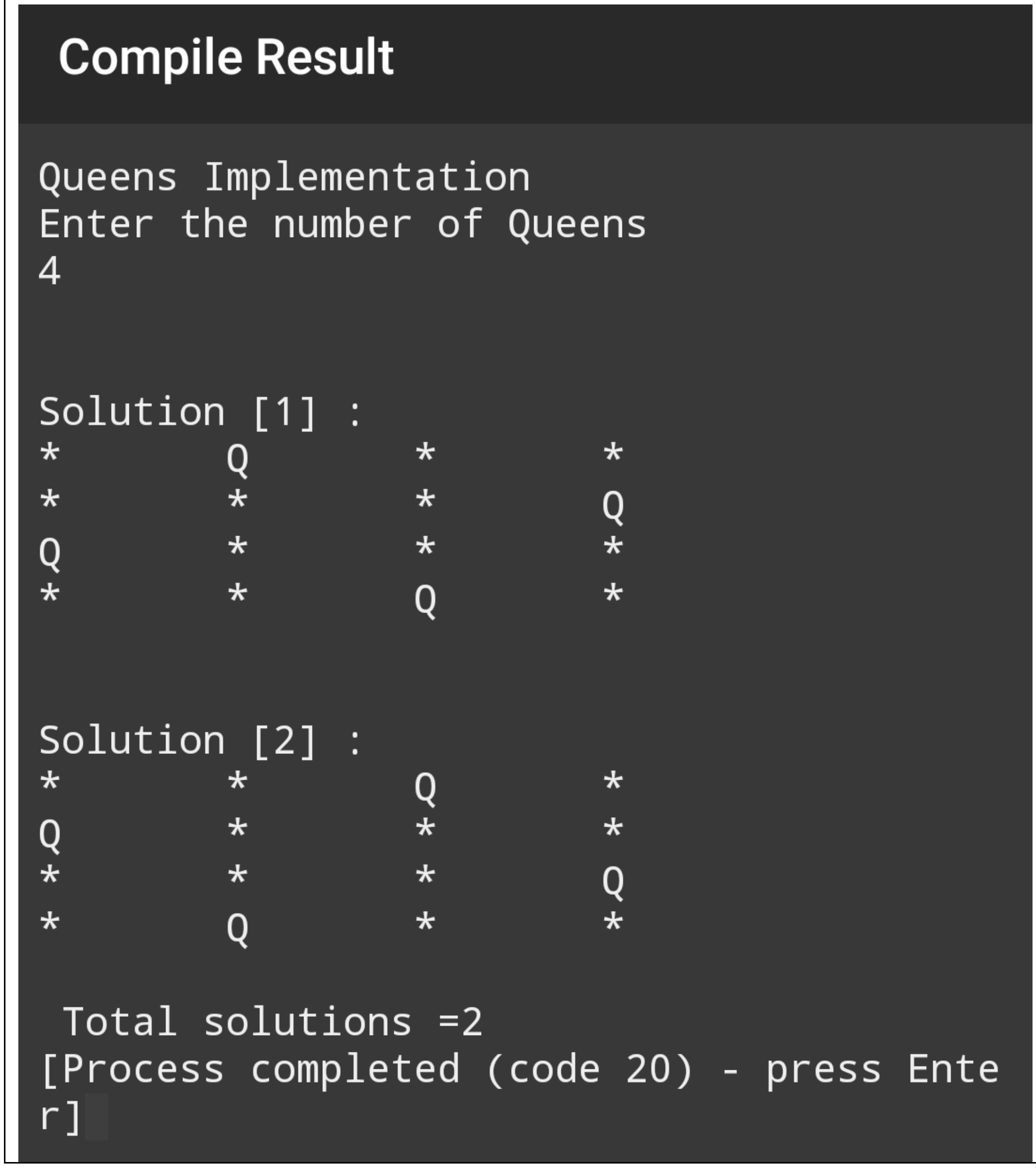
Program:-

```
//Umesh Mane
//Roll No 44
#include<stdio.h>
#include<math.h>
int board[30],count=0;
void queen(int);
int placeQueen(int);
void printSolution(int);
void main()
{
int i,n;
printf("Queens Implementation\n");
printf("Enter the number of Queens\n");
scanf("%d",&n);
queen(n);
printf("\n Total solutions =%d",count);
}
int placeQueen(int pos)
{
int i;
for(i=1;i<pos;i++)
{
if((board[i]==board[pos]) ||((abs(board[i]-board[pos])==abs(i-pos))))
return 0;
}
```

```
return 1;
}
void printSolution(int n)
{
int i,j;
count++;
printf("\n\nSolution [%d] :\n",count);
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
if(board[i]==j)
printf("Q\t");
else
printf("*\t");
}
printf("\n");
}
}
void queen(int n)
{
int k=1;
board[k]=0;
while(k!=0)
{
board[k]=board[k]+1;
while((board[k]<=n) && !placeQueen(k))
board[k]++;
if(board[k]<=n)
{
if(k==n)
printSolution(n);
else
{
k++;
board[k]=0;
}
}
else
k--;
```

}
}

OUTPUT:



Compile Result

```
Queens Implementation
Enter the number of Queens
4


Solution [1] :
*       Q       *       *
*       *       *       Q
Q       *       *       *
*       *       Q       *


Solution [2] :
*       *       Q       *
Q       *       *       *
*       *       *       Q
*       Q       *       *

 Total solutions =2
[Process completed (code 20) - press Ente
r]
```

CONCLUSION:- We have understood the program for N-Queen Problem