DRONACHARYA
Group of Institutions

B-27, Knowledge Park – III, Greater Noida, Uttar Pradesh - 201308
Approved by: All India Council for Technical Education (AICTE), New Delhi
Affiliated to: Dr. A. P. J. Abdul Kalam Technical University (AKTU), Lucknow

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Lab Manual For

Data Structures Lab

SUBJECT CODE: BCS-351

Table of Contents

DRONACHARYA GROUP OF INSTITUTIONS GREATER NOIDA

VISION

- Instilling core human values and facilitating competence to address global challenges by providing Quality Technical Education.

MISSION

- M1 - Enhancing technical expertise through innovative research and education, fostering creativity and excellence in problem-solving.
- M2 - Cultivating a culture of ethical innovation and user-focused design, ensuring technological progress enhances the well-being of society.
- M3 - Equipping individuals with the technical skills and ethical values to lead and innovate responsibly in an ever-evolving digital land

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VISION

Promoting technologists by imparting profound knowledge in information technology, all while instilling ethics through specialized technical education.

MISSION

- Delivering comprehensive knowledge in information technology, preparing technologists to excel in a rapidly evolving digital landscape.
- Building a culture of honesty and responsibility in tech, promoting smart and ethical leadership.
- Empowering individuals with specialized technical skills and ethical values to drive positive change and innovation in the tech industry.

Program Outcomes (POs)

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of engineering practice.

PO 9: Individual and teamwork: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply theseto one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Programme Educational Objectives (PEOs)

PEO1: To enable graduates to pursue higher education and research, or have a successful career in industries associated with Computer Science and Engineering, or as entrepreneurs.

PEO2: To ensure that graduates will have the ability and attitude to adapt to emerging technological changes.

PEO3: To prepare students to analyze existing literature in an area of specialization and ethically develop innovative methodologies to solve the problems identified.

Program Specific Outcomes (PSOs)

PSO1: To analyze, design and develop computing solutions by applying foundational concepts of Computer Science and Engineering.

PSO2: To apply software engineering principles and practices for developing quality software for scientific and business applications.

PSO3: To adapt to emerging Information and Communication Technologies (ICT) to innovate ideas and solutions to existing/novel problems.

CO – PO Mapping

BCS-351. 1	Practice various Sorting and Searching Algorithms.
BCS-351. 2	Analyze the recursive implementation of different sorting and searching algorithms.
BCS-351. 3	Exercise various data Structure operations using static and dynamic memory allocation.
BCS-351. 4	Demonstrate various operations like traversal, insertion, deletion on tree data structure.
BCS-351. 5	Illustrate and Implement practical applications based on graphs and shortest paths.

CO-PSO Mapping:

	PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
CO 1	3	3	3	-	2	-	-	-	-	1	1	2
CO 2	3	3	3	-	2	-	-	-	-	1	1	2
CO 3	3	2	3	-	2	-	-	-	-	1	1	2
CO 4	3	3	3	-	2	-	-	-	-	1	1	2
CO 5	3	3	3	-	2	-	-	-	-	1	1	3

PSO1	**PSO2**	**PSO3**
1	2	1
1	3	1
1	3	1
2	2	1
2	2	1

Course Overview

The students will be able to investigates abstract data types (ADTs), recursion, algorithms for searching and sorting, and basic algorithm analysis. ADTs to be covered include lists, stacks, queues, priority queues, trees, sets, and graphs. The emphasis is on the trade-offs associated with implementing alternative data structures for these ADTs.

Course Objectives

- To introduce the fundamental concept of data structures including link-list
- To emphasize the importance of data structures in implementing the algorithms
- To develop effective skills in the implementation of data structure

LIST OF PROGRAMS

S.No.	TYPE OF PROBLEM	PROGRAM
1.	Linear and 2D Array	Write a program to insert an element in array
2.		Write a program to access matrix by recursive call
3.		To implement addition and multiplication of two 2D arrays
4.		Write a program to calculate transpose a 2D array
5.	Stack using Array & Linked List	Write a program to implement stack using array.
6.		Write a program to implement Linked List insertion.
7.		Write a program to implement stack using linked list.
8.	Queue using Array & Linked List	Write a program to implement circular queue using array.
9.		Write a program to implement queue using array.
10.		Write a program to implement queue using linked list.
11.		Write a program to implement circular queue using linked list.
12.	Tree & Graph using Linked List	Write a program to implement BFS using linked list.
13.		Write a program to implement tree traversals using linked list.
14.		Write a program to implement DFS using linked list.
15.	Linear Search	Write a program on linear search
16.		Write a program to compare two numbers
17.		Write a program to access array elements
18.		Write a program on linear search based on link-list
19.	Binary Search	Write a program on Binary Search
20.		Write a program to access array elements randomly
21.		Write a program on multi-way Search
22.	Bubble sort	Write a program on Bubble sort
23.		Write a program to swap numbers
24.		Write a program to use nested loops
25.		Write a program on linear bubble sort
26.	Selection sort	Write a program on Selection sort

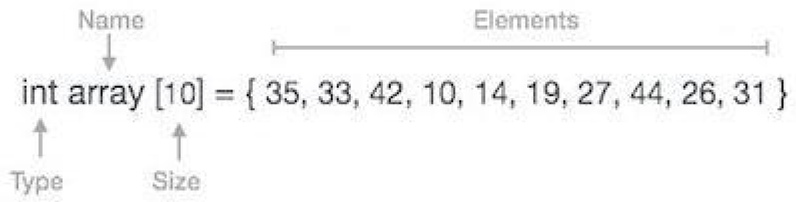
27.		Write a program to find largest from array
28.		Write a program on Selection sort based on link-list
29.	Insertion sort	Write a program on Insertion sort
30.		Write a program to insert a number into a array
31.		Write a program on Insertion sort based on link-list
32.	Merge sort	Write a program on Merge sort
33.		Write a program to merge two sorted array
34.		Write a program to use a recursive function call
35.		Write a program to display step by step merge sort
36.	Quick sort	Write a program on Quick sort
37.		Write a program to divide an array into two part
38.		Write a program to display step by step quick sort
39.	Heap sort	Write a program on Heap sort
40.		Write a program to represent binary tree using array
41.		Write a program to create Max-Heap
42.		Write a program on Heap sort in descending order

Experiment plan

Experiment No.	1.
Type of problem	Linear and 2D Array
Students are required to	Algorithm

implement/execute/draw/make document	Program Output Final document with evaluator signature.		
List of experiments	1. Write a program to insert an element in array 2. Write a program to access matrix by recursive call 3. To implement addition and multiplication of two 2D arrays 4. Write a program to calculate transpose a 2D array		

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.
Element − Each item stored in an array is called an element.
Index − Each location of an element in an array has a numerical index, which is used to identify the element.
Array Representation
Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



As per the above illustration, following are the important points to be considered.
Index starts with 0.
Array length is 10 which mean it can store 10 elements.
Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.
Basic Operations
Following are the basic operations supported by an array.
Traverse − print all the array elements one by one.
Insertion − Adds an element at the given index.
Deletion − Deletes an element at the given index.
Search − Searches an element using the given index or by the value.
Update − Updates an element at the given index.
Insertion Operation
Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.
Here, we see a practical implementation of insertion operation, where we add data at the end of the array −
Algorithm
Let Array be a linear unordered array of MAX elements.
Example
Result
Let LA be a Linear Array (unordered) with N elements and K is a positive integer such that K<=N. Following is the algorithm where ITEM is inserted into the Kth position of LA −
1. Start

```
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to delete an element available at the Kth position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J-1] = LA[J]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to find an element with a value of ITEM using sequential search.

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to update an element available at the Kth position of LA.

```
1. Start
2. Set LA[K-1] = ITEM
3. Stop
```

C Program to Insert an Element in a Specified Position in a given Array

```
/* C program to insert a particular element in a specified position
 * in a given array */
#include <stdio.h>

void main()
{
```

```
int array[10];
int i, j, n, m, temp, key, pos;

printf("Enter how many elements \n");
scanf("%d", &n);
printf("Enter the elements \n");
for (i = 0; i < n; i++)
{
   scanf("%d", &array[i]);
}
printf("Input array elements are \n");
for (i = 0; i < n; i++)
{
   printf("%d\n", array[i]);
}
for (i = 0; i < n; i++)
{
   for (j = i + 1; j < n; j++)
   {
      if (array[i] > array[j])
      {
         temp = array[i];
         array[i] = array[j];
         array[j] = temp;
      }
    }
}
printf("Sorted list is \n");
for (i = 0; i < n; i++)
{
   printf("%d\n", array[i]);
}
printf("Enter the element to be inserted \n");
scanf("%d", &key);
for (i = 0; i < n; i++)
{
   if (key < array[i])
   {
      pos = i;
      break;
   }
   if (key > array[n-1])
   {
      pos = n;
      break;
   }
}
if (pos != n)
{
   m = n - pos + 1 ;
```

```
        for (i = 0; i <= m; i++)
        {
            array[n - i + 2] = array[n - i + 1] ;
        }
    }
    array[pos] = key;
    printf("Final list is \n");
    for (i = 0; i < n + 1; i++)
    {
        printf("%d\n", array[i]);
    }
}
```

OUTPUT

```
Enter how many elements
5
Enter the elements
76
90
56
78
12
Input array elements are
76
90
56
78
12
Sorted list is
12
56
76
78
90
Enter the element to be inserted
61
Final list is
12
56
61
76
78
90
```

Write a program to access matrix by recursive call

```
#include <stdio.h>
#define MAX_SIZE 100

/* Function declaration */
void printArray(int arr[], int start, int len);
```

```
int main()
{
   int arr[MAX_SIZE];
   int N, i;

   /* Input size and elements in array */
   printf("Enter size of the array: ");
   scanf("%d", &N);
   printf("Enter elements in the array: ");
   for(i=0; i<N; i++)
   {
      scanf("%d", &arr[i]);
   }

   /* Prints array recursively */
   printf("Elements in the array: ");
   printArray(arr, 0, N);

   return 0;
}


/**
 * Prints an array recursively within a given range.
 */
void printArray(int arr[], int start, int len)
{
   /* Recursion base condition */
   if(start >= len)
      return;

   /* Prints the current array element */
   printf("%d, ", arr[start]);

   /* Recursively call printArray to print next element in the array */
   printArray(arr, start + 1, len);
}
```

OUTPUT

```
Enter size of the array: 10
Enter elements in the array: 1 2 3 4 5 6 7 8 9 10
Elements in the array: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

To implement addition and multiplication of two 2D arrays

```
#include<stdio.h>

void Matrix_Display(int a[][20],int n)
{
    int i,j;
    for(i=0; i<n; i++)
```

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

}

int main()
{
    int n,i,j,k;
    int a[20][20];
    int b[20][20];
    int c[20][20];

    printf("\n Enter the dimensions of the 2 Square matrices: ");
    scanf("%d",&n);

    printf("\n Enter elements of Matrix A: ");
    for(i=0; i<n; i++)
    for(j=0; j<n; j++)
    scanf("%d",&a[i][j]);

    printf("\n Enter elements of Matrix B: ");
    for(i=0; i<n; i++)
    for(j=0; j<n; j++)
    scanf("%d",&b[i][j]);

    printf("\n Matrix A: \n");
    Matrix_Display(a,n);

    printf("\n\n Matrix B: \n");
    Matrix_Display(b,n);

    //Addition
    for(i=0; i<n; i++)
    for(j=0; j<n; j++)
    c[i][j]=a[i][j]+b[i][j];

    printf("\n\n Addition of A and B gives: \n");
    Matrix_Display(c,n);

     //Subtraction
    for(i=0; i<n; i++)
    for(j=0; j<n; j++)
    c[i][j]=a[i][j]-b[i][j];

    printf("\n\n Subtraction of A and B gives: \n");
    Matrix_Display(c,n);

    //Multiplication
    for(i=0; i<n; i++)
    for(j=0; j<n; j++)
    {
     c[i][j]=0;
     for(k=0; k<n; k++)
     c[i][j]+=a[i][k]*b[k][j];
    }

    printf("\n\n Multiplication of A and B gives: \n");
    Matrix_Display(c,n);

}
```

OUTPUT

```
Enter the dimensions of the 2 Square matrices: 2

 Enter elements of Matrix A: 1 2 3 4

 Enter elements of Matrix B: 4 3 2 1

 Matrix A:
 1 2
 3 4

 Matrix B:
 4 3
 2 1

 Addition of A and B gives:
 5 5
 5 5
 Subtraction of A and B gives:
 -3 -1
 1 3

 Multiplication of A and B gives:
 8 5
 20 13
```

C program to transpose the 2D array

```
        /* C program to transpose the 2D array
        */
          #include<stdio.h>

          void main()
          {
```

```
    int a[10][10], b[10][10], m, n, i, j;

    printf("\nEnter number of rows & columns of aray :
");
    scanf("%d %d", &m, &n);
    printf("\nEnter elements of 2-D array:\n");
    for(i=0; i<m; i++)
    {
           for(j=0; j<n; j++)
           {
                  scanf("%d", &a[i][j]);
           }


}
    printf("\n\n2-D array before
transposing:\n\n");
    for(i=0; i<m; i++)
    {
           for(j=0; j<n; j++)
           {
                  printf("\t%d", a[i][j]);
           }
           printf("\n\n");
    }

    /* Transposing array */
    for(i=0; i<m; i++)
    {

           for(j=0; j<n; j++)
           {
                  b[j][i] = a[i][j];
           }
    }

    printf("\n\n2-D array after
transposing:\n\n");
    for(i=0; i<n; i++)
    {
           for(j=0; j<m; j++)
           {

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


  }

           printf("\n\
n");
    }
    getch();
}
```

We create the 2D array first and after that its transpose is done which is stored in new 2D array. Hence the result is printed.

Input-

```
Enter number of rows & columns of array: 3 x 3
Enter elements of 2-D array:
25   12   4   62   34   23   6   4   3
```

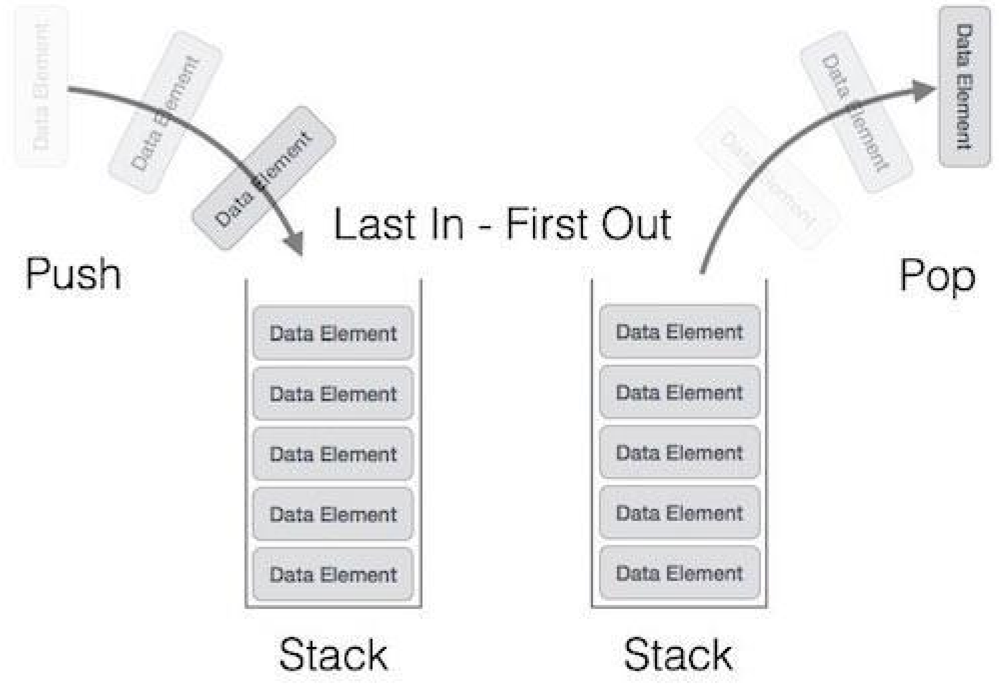
OUTPUT-

```
2-D array before transposing:
25   12  4
62   34  23
6    4    3
2-D array after transposing:
25   62   6
12   34   4
4    23    3
```

Experiment No.	2.
Type of problem	Stack using Array & Linked List
Students are required to implement/execute/draw/make document	1. Algorithm 2. Program 3. Output 4. Final document with evaluator signature.

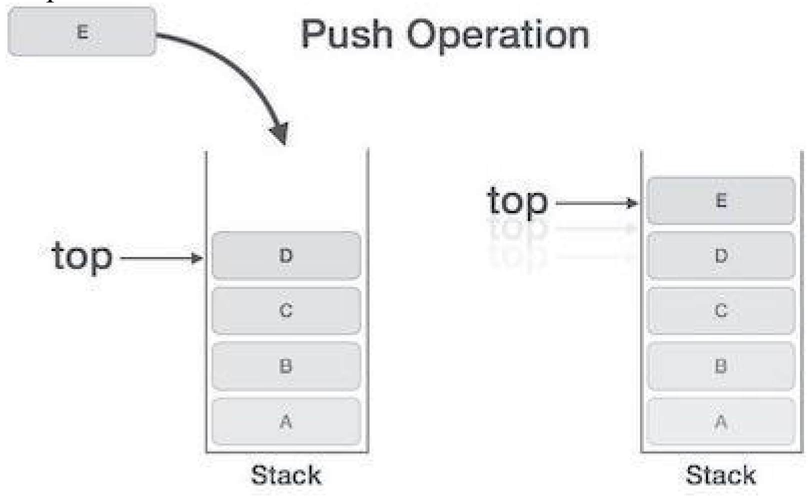
List of experiments	1. Write a program to implement stack using array. 2. Write a program to implement Linked List insertion. 3. Write a program to implement stack using linked list.		

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc. A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.
This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.
Stack Representation
The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.
Basic Operations
Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −
push() − Pushing (storing) an element on the stack.
pop() − Removing (accessing) an element from the stack.
When data is PUSHed onto stack.
To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

peek() − get the top data element of the stack, without removing it.
isFull() − check if stack is full.
isEmpty() − check if stack is empty.
At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named top. The top pointer provides top value of the stack without actually removing it.
Push Operation
The process of putting a new data element onto stack is known as a Push Operation.
Push operation involves a series of steps −
Step 1 − Checks if the stack is full.
Step 2 − If the stack is full, produces an error and exit.
Step 3 − If the stack is not full, increments top to point next empty space.
Step 4 − Adds data element to the stack location, where top is pointing.
Step 5 − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.
Algorithm for PUSH Operation
A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data

  if stack is full
    return null
  endif

  top ← top + 1

  stack[top] ← data

end procedure
```

Implementation of this algorithm in C, is very easy. See the following code −
Example

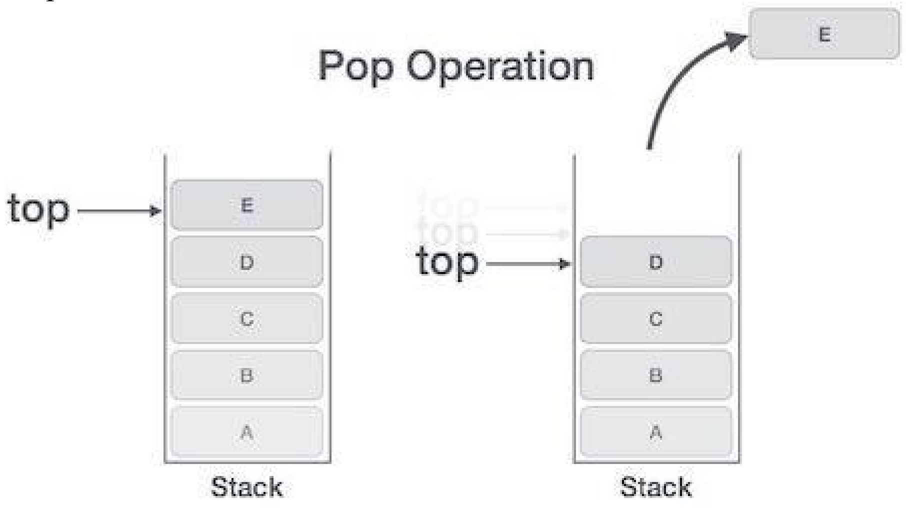
```
void push(int data) {
  if(!isFull()) {
    top = top + 1;
    stack[top] = data;
  } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

Step 1 − Checks if the stack is empty.
Step 2 − If the stack is empty, produces an error and exit.
Step 3 − If the stack is not empty, accesses the data element at which top is pointing.
Step 4 − Decreases the value of top by 1.
Step 5 − Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]

   top ← top - 1

   return data

end procedure
```

Implementation of this algorithm in C, is as follows −

Example

```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

Write a program to implement stack using array.

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

```
          }
          default:
          {
             printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
          }
          getch();
          }
       }
       while(choice!=4);
    }
    void push()
    {
       if(top>=n-1)
       {
          printf("\n\tSTACK is over flow");
          getch();
       }
       else
       {
          printf(" Enter a value to be pushed:");
          scanf("%d",&x);
          top++;
          stack[top]=x;
       }
    }
    void pop()
    {
       if(top<=-1)
       {
          printf("\n\t Stack is under flow");
       }
       else
       {
          printf("\n\t The popped elements is %d",stack[top]);
          top--;
       }
    }
    void display()
    {
       if(top>=0)
       {
          printf("\n The elements in STACK \n");
          for(i=top; i>=0; i--)
             printf("\n%d",stack[i]);
          printf("\n Press Next Choice");
       }
       else
       {
          printf("\n The STACK is empty");
       }
```

```
}
```

OUTPUT

```
Enter the size of STACK[MAX=100]:10

        STACK OPERATIONS USING ARRAY
        --------------------------------
        1.PUSH
        2.POP
        3.DISPLAY
        4.EXIT
 Enter the Choice:1
 Enter a value to be pushed:12

 Enter the Choice:1
 Enter a value to be pushed:24

 Enter the Choice:1
 Enter a value to be pushed:98

 Enter the Choice:3

 The elements in STACK

98
24
12
 Press Next Choice
 Enter the Choice:2

        The popped elements is 98
 Enter the Choice:3

 The elements in STACK

24
12
 Press Next Choice
 Enter the Choice:4

        EXIT POINT
```

Write a program to implement Linked List insertion.

```
/ A complete working C program to demonstrate all insertion methods
// on Linked List
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct Node
```

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

/* Given a reference (pointer to pointer) to the head of a list and
   an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    /* 2. put in the data  */
    new_node->data  = new_data;

    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);

    /* 4. move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Given a node prev_node, insert a new node after the given
   prev_node */
void insertAfter(struct Node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
      printf("the given previous node cannot be NULL");
      return;
    }

    /* 2. allocate new node */
    struct Node* new_node =(struct Node*) malloc(sizeof(struct Node));

    /* 3. put in the data  */
    new_node->data  = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. move the next of prev_node as new_node */
    prev_node->next = new_node;
}

/* Given a reference (pointer to pointer) to the head
   of a list and an int, appends a new node at the end  */
void append(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    struct Node *last = *head_ref;  /* used in step 5*/

    /* 2. put in the data  */
    new_node->data  = new_data;

    /* 3. This new node is going to be the last node, so make next of
          it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new node as head */
    if (*head_ref == NULL)
    {
       *head_ref = new_node;
       return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;
    return;
}

// This function prints contents of linked list starting from head
void printList(struct Node *node)
{
  while (node != NULL)
  {
     printf(" %d ", node->data);
     node = node->next;
  }
}

/* Driver program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct Node* head = NULL;
  // Insert 6.  So linked list becomes 6->NULL
  append(&head, 6);
  // Insert 7 at the beginning. So linked list becomes 7->6->NULL
  push(&head, 7);
  // Insert 1 at the beginning. So linked list becomes 1->7->6->NULL
  push(&head, 1);
  // Insert 4 at the end. So linked list becomes 1->7->6->4->NULL
  append(&head, 4);
  // Insert 8, after 7. So linked list becomes 1->7->8->6->4->NULL
  insertAfter(head->next, 8);
  printf("\n Created Linked list is: ");
  printList(head);
  return 0;
}
```

OUTPUT

```
Created Linked list is:  1  7  8  6  4
```

3. Write a program to implement stack using linked list.

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int info;
   struct node
*ptr;

}*top,*top1,*tem
p;

int
topelement(
);
void push(int data);
void pop();
void
empty();
void
display();
void
destroy();
void
stack_count(
);
void
create();
int count
= 0;
void
main()

{
   int no, ch, e;
   printf("\n 1 - Push");
   printf("\n 2 - Pop");
   printf("\n 3 - Top");
   printf("\n 4 - Empty");
   printf("\n 5 - Exit");
   printf("\n 6 - Dipslay");
   printf("\n 7 - Stack Count");
   printf("\n 8 - Destroy
stack");

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

      switch (ch)
      {
      case 1:
         printf("Enter
data : ");
```

```
            scanf("%d",
&no);
            push(no);

            break;
        Case 2:
            pop();

            break;

        case 3:
            if (top == NULL)
                printf("No elements in stack");
            else
            {

                e = topelement();
                printf("\n Top element : %d", e);
            }
            break;
        case 4:
            empty();
            break;
        case 5:
            exit(0);
        case 6:
            display();
            break;
        case 7:
            stack_count();
            break;
        case 8:
            destroy();
            break;
        default :
            printf(" Wrong choice, Please enter correct choice  ");
            break;
        }
    }
}

/* Create empty stack */

void create()
{
    top = NULL;
}
```

```
/* Count stack elements */
void stack_count()
{

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

}


/* Push data into stack */
void push(int data)
{
   if (top == NULL)
   {
      top =(struct node *)malloc(1*sizeof(struct node));
      top->ptr = NULL;
      top->info = data;
   }
   else
   {
      temp =(struct node
*)malloc(1*sizeof(struct node));
      temp->ptr = top;
      temp->info = data;
      top = temp;
   }

   count++;
}

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

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

   while (top1 != NULL)
   {


      printf("%d ", top1->info);

      top1 = top1->ptr;
   }
 }
```

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

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

   free(top);
   top = top1;
   count--;
}

/* Return top element */
int topelement()
{

   return(top->info);
}

/* Check if stack is empty or not */
void empty()
{
   if (top == NULL)

       printf("\n Stack is empty");
   else
       printf("\n Stack is not empty with %d elements",
count);
}

/* Destroy entire stack */
void destroy()
{


top1 = top;
   while (top1 != NULL)
```

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

    printf("\n All stack
elements destroyed");

    count = 0;
}
```

OUTPUT

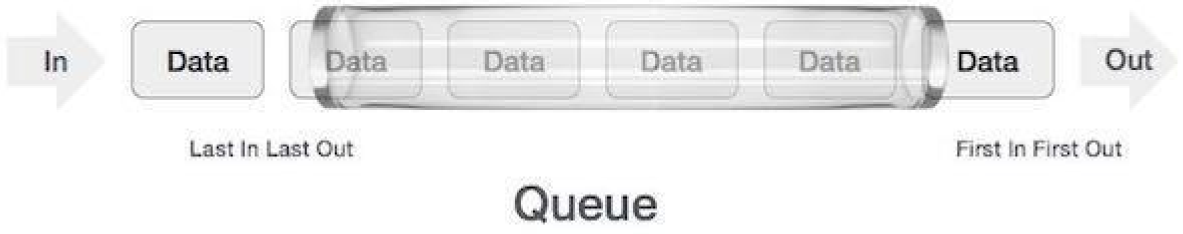
```
1 - Push
2 - Pop
3 - Top
4 - Empty
5 -
Exit
6 -
Dipsla
y
7 -
Stack
Count
8 - Destroy stack
Enter
choice :
1
Enter
data :
56
Enter
choic
e : 1
Enter
data :
80
Ente
r
choi
ce :
2
```

```
Popped value : 80
Enter choice : 3

Top element : 56

Enter choice : 1
Enter data : 78


Enter choice : 1

Enter data : 90

Enter choice : 6

90 78 56

Enter choice : 7

No. of elements in stack : 3

Enter choice : 8


All stack elements destroyed

Enter choice : 4

Stack is empty

Enter choice : 5
```

Experiment No.	3.
Type of problem	Queue using Array & Linked List
Students are required to implement/execute/draw/make document	1. Algorithm 2. Program 3. Output 4. Final document with evaluator signature.

List of experiments	1. Write a program to implement circular queue using array. 2. Write a program to implement queue using array. 3. Write a program to implement queue using linked list. 4. Write a program to implement circular queue using linked list.		

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.
A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.
Queue Representation
As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.
Basic Operations
Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −
enqueue() − add (store) an item to the queue.
dequeue() − remove (access) an item from the queue.
Few more functions are required to make the above-mentioned queue operation efficient. These are −
peek() − Gets the element at the front of the queue without removing it.
isfull() − Checks if the queue is full.
isempty() − Checks if the queue is empty.
In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear pointer.
Let's first learn about supportive functions of a queue −
peek()
This function helps to see the data at the front of the queue. The algorithm of peek() function is as follows −
Algorithm
begin procedure peek

```
  return queue[front]

end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {
  return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

Algorithm

```
begin procedure isfull

  if rear equals to MAXSIZE
    return true
  else
    return false
  endif

end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {
  if(rear == MAXSIZE - 1)
    return true;
  else
    return false;
}
```

isempty()

Algorithm of isempty() function −

Algorithm

```
begin procedure isempty

  if front is less than MIN  OR front is greater than rear
    return true
  else
    return false
  endif

end procedure
```

If the value of front is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

Example

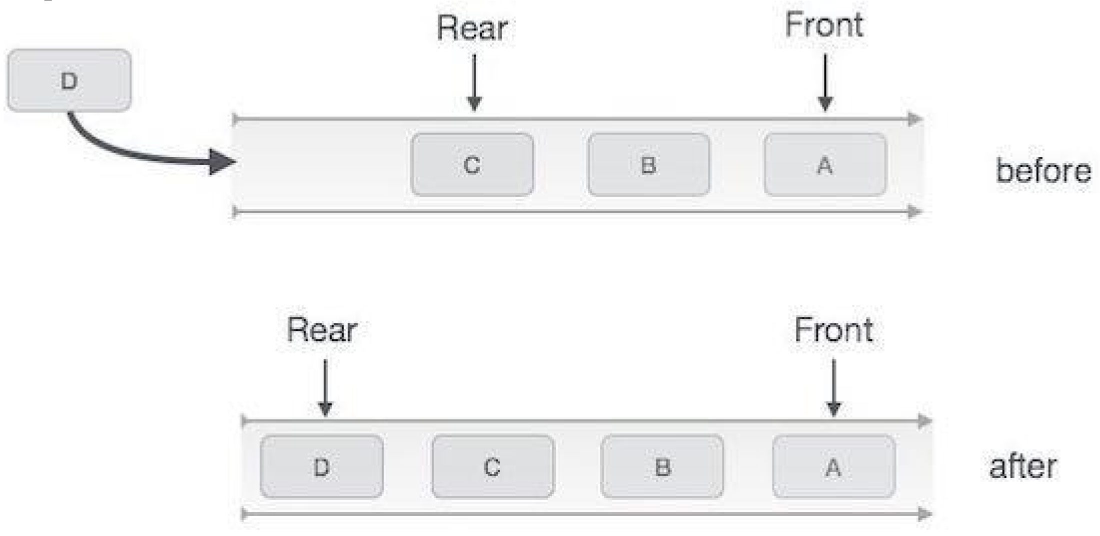
```
bool isempty() {
  if(front < 0 || front > rear)
    return true;
  else
    return false;
}
```

Enqueue Operation

Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

Step 1 − Check if the queue is full.
Step 2 − If the queue is full, produce overflow error and exit.
Step 3 − If the queue is not full, increment rear pointer to point the next empty space.
Step 4 − Add data element to the queue location, where the rear is pointing.
Step 5 − return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)
  if queue is full
    return overflow
  endif

  rear ← rear + 1

  queue[rear] ← data

  return true

end procedure
```

Implementation of enqueue() in C programming language −

Example

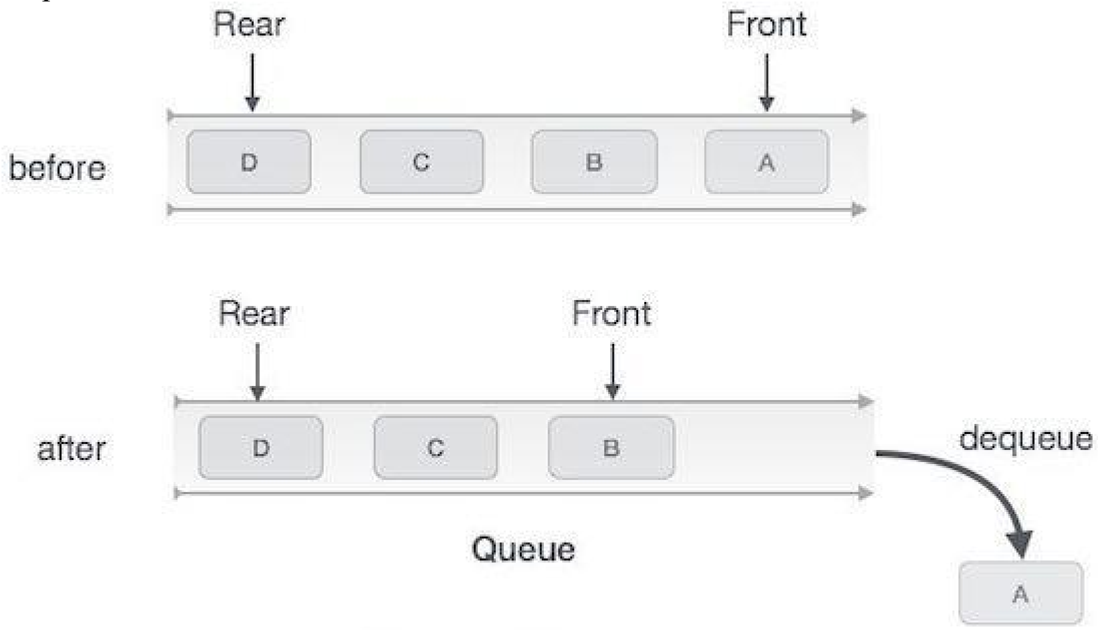
```
int enqueue(int data)
  if(isfull())
    return 0;
```

```
   rear = rear + 1;
   queue[rear] = data;

   return 1;
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation −

Step 1 − Check if the queue is empty.
Step 2 − If the queue is empty, produce underflow error and exit.
Step 3 − If the queue is not empty, access the data where front is pointing.
Step 4 − Increment front pointer to point to the next available data element.
Step 5 − Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue
   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1

   return true
end procedure
```

Implementation of dequeue() in C programming language −

Example

```
int dequeue() {

   if(isempty())
      return 0;
```

```
  int data = queue[front];
  front = front + 1;

  return data;
}
```

Write a program to implement circular queue using array

```
//Program for Circular Queue implementation through Array
#include <stdio.h>
#include<ctype.h>
#include<stdlib.h>
#define MAXSIZE 5
int cq[MAXSIZE];
int front,rear;
void main()
{
        void add(int,int);
        void del(int);
        int will=1,i,num;
        front = -1;
        rear = -1;
        clrscr();
        printf("\nProgram for Circular Queue demonstration through array");
        while(1)
        {
                printf("\n\nMAIN MENU\n1.INSERTION\n2.DELETION\n3.EXIT");
                printf("\n\nENTER YOUR CHOICE : ");
                scanf("%d",&will);
                switch(will)
                {
                case 1:
                        printf("\n\nENTER THE QUEUE ELEMENT : ");
                        scanf("%d",&num);
                        add(num,MAXSIZE);
                        break;
                case 2:
                        del(MAXSIZE);
                        break;
                case 3:
                   exit(0);
                        default: printf("\n\nInvalid Choice . ");
                }

} //end of  outer while
}            //end of main
void add(int item,int MAX)
{
        //rear++;
```

```
            //rear= (rear%MAX);
            if(front ==(rear+1)%MAX)
            {
            printf("\n\nCIRCULAR QUEUE IS OVERFLOW");
            }
            else
            {
              if(front==-1)
                front=rear=0;
                else
                rear=(rear+1)%MAX;
                cq[rear]=item;
                printf("\n\nRear = %d    Front = %d ",rear,front);
            }
}
void del(int MAX)
{
int a;
if(front == -1)
            {
            printf("\n\nCIRCULAR QUEUE IS UNDERFLOW");
            }
            else
            {
                        a=cq[front];
                        if(front==rear)
                         front=rear=-1;
                        else
                          front = (front+1)%MAX;
                printf("\n\nDELETED ELEMENT FROM QUEUE IS : %d ",a);
                        printf("\n\nRear = %d    Front = %d ",rear,front);

            }
}
```

OUTPUT

```
MAIN MENU
1. INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 1
ENTER THE QUEUE ELEMENT : 10
Rear=0     Front=0
MAIN MENU
1. INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 1
ENTER THE QUEUE ELEMENT : 20
Rear=1     Front=0
MAIN MENU
1. INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 1
ENTER THE QUEUE ELEMENT : 30
Rear=2     Front=0
MAIN MENU
1. INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 1
ENTER THE QUEUE ELEMENT : 40
Rear=3     Front=0
MAIN MENU
1. INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 1
ENTER THE QUEUE ELEMENT : 50
Rear=4    Front=0
MAIN MENU
1. INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 1
ENTER THE QUEUE ELEMENT : 60
CIRCULAR QUEUE IS OVERFLOW.
MAIN MENU
1. INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 2
DELETED ELEMENT FROM QUEUE IS : 10
Rear =4    Front=1
MAIN MENU
1. INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 2
DELETED ELEMENT FROM QUEUE IS : 20
Rear =4    Front=2
MAIN MENU
1. INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 2
DELETED ELEMENT FROM QUEUE IS : 30
```

```
Rear =4    Front=3
MAIN MENU
1. INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 2
DELETED ELEMENT FROM QUEUE IS : 40
Rear =4    Front=4
MAIN MENU
1. INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 2
DELETED ELEMENT FROM QUEUE IS : 50
Rear =-1    Front=-1
MAIN MENU
1. INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 2
CIRCULAR QUEUE IS UNDERFLOW.
```

Write a program to implement queue using array.

```c
/*
 * C Program to Implement a Queue using an Array
 */
#include <stdio.h>

#define MAX 50
int queue_array[MAX];
int rear = - 1;
int front = - 1;
main()
{
   int choice;
   while (1)
   {
      printf("1.Insert element to queue \n");
      printf("2.Delete element from queue \n");
      printf("3.Display all elements of queue \n");
      printf("4.Quit \n");
      printf("Enter your choice : ");
      scanf("%d", &choice);
      switch (choice)
      {
         case 1:
         insert();
         break;
         case 2:
         delete();
         break;
         case 3:
         display();
         break;
         case 4:
         exit(1);
         default:
         printf("Wrong choice \n");
      } /*End of switch*/
   } /*End of while*/
} /*End of main()*/
insert()
{
   int add_item;
   if (rear == MAX - 1)
   printf("Queue Overflow \n");
   else
   {
      if (front == - 1)
      /*If queue is initially empty */
      front = 0;
      printf("Inset the element in queue : ");
      scanf("%d", &add_item);
      rear = rear + 1;
      queue_array[rear] = add_item;
   }
} /*End of insert()*/

delete()
{
   if (front == - 1 || front > rear)
   {
      printf("Queue Underflow \n");
      return ;
   }
   else
   {
      printf("Element deleted from queue is : %d\n", queue_array[front]);
      front = front + 1;
   }
} /*End of delete() */
display()
{
   int i;
   if (front == - 1)
      printf("Queue is empty \n");
   else
   {
      printf("Queue is : \n");
      for (i = front; i <= rear; i++)
         printf("%d ", queue_array[i]);
      printf("\n");
   }
} /*End of display() */
```

Program Explanation
1. Ask the user for the operation like insert, delete, display and exit.
2. According to the option entered, access its respective function using switch statement. Use the variables front and rear to represent the first and last element of the queue.
3. In the function insert(), firstly check if the queue is full. If it is, then print the output as “Queue Overflow”. Otherwise take the number to be inserted as input and store it in the variable add_item. Copy the variable add_item to the array queue_array[] and increment the variable rear by 1.
4. In the function delete(), firstly check if the queue is empty. If it is, then print the output as “Queue Underflow”. Otherwise print the first element of the array queue_array[] and decrement the variable front by 1.
5. In the function display(), using for loop print all the elements of the array starting from front to rear.
6. Exit.

Runtime Test Cases

```
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 10
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 15
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 20
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 30
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
```

```
4.Quit
Enter your choice : 2
Element deleted from queue is : 10
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3
Queue is :
15 20 30
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 4
```

Write a program to implement queue using linked list.

```
/*
 * C Program to Implement Queue Data Structure using Linked List
 */
#include <stdio.h>
#include <stdlib.h>

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

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

int count = 0;

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

    printf("\n 1 - Enque");
    printf("\n 2 - Deque");
    printf("\n 3 - Front element");
    printf("\n 4 - Empty");
    printf("\n 5 - Exit");
    printf("\n 6 - Display");
    printf("\n 7 - Queue size");
    create();
    while (1)
    {
        printf("\n Enter choice : ");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            printf("Enter data : ");
            scanf("%d", &no);
            enq(no);
            break;
        case 2:
            deq();
            break;
        case 3:
            e = frontelement();
            if (e != 0)
                printf("Front element : %d", e);
            else
                printf("\n No front element in Queue as queue is empty");
            break;
        case 4:
            empty();
            break;
        case 5:
            exit(0);
        case 6:
            display();
            break;
        case 7:
            queuesize();
            break;
        default:
            printf("Wrong choice, Please enter correct choice  ");
            break;
        }
    }
}

/* Create an empty queue */
void create()
{
    front = rear = NULL;
}

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

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

        rear = temp;
    }
    count++;
}

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

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

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

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

/* Returns the front element of queue */
int frontelement()
{
    if ((front != NULL) && (rear != NULL))
        return(front->info);
    else
        return 0;
}

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

OUTPUT

```
1 - Enque
2 - Deque
3 - Front element
4 - Empty
5 - Exit
6 - Display
7 - Queue size
```

```
Enter choice : 1
Enter data : 14

Enter choice : 1
Enter data : 85

Enter choice : 1
Enter data : 38

Enter choice : 3
Front element : 14
Enter choice : 6
14 85 38
Enter choice : 7

Queue size : 3
Enter choice : 2

Dequed value : 14
Enter choice : 6
85 38
Enter choice : 7

Queue size : 2
Enter choice : 4
Queue not empty
Enter choice : 5
```

Write a program to implement circular queue using linked list.

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

struct node
{
int info;
struct node *link;
}*rear=NULL;

void insert(int item);
int del();
void display();
int isEmpty();
int peek();

int main()
{
int choice,item;
while(1)
{
printf("\n1.Insert\n");
printf("2.Delete\n");
printf("3.Peek\n");
printf("4.Display\n");
printf("5.Quit\n");
printf("\nEnter your choice : ");
scanf("%d",&choice);

switch(choice)
{
case 1:
printf("\nEnter the element for insertion : ");
scanf("%d",&item);
insert(item);
break;
case 2:
printf("\nDeleted element is %d\n",del());
break;
case 3:
printf("\nItem at the front of queue is %d\n",peek());
break;
case 4:
display();
break;
case 5:
exit(1);
default:
printf("\nWrong choice\n");
}/*End of switch*/
}/*End of while*/
}/*End of main()*/

void insert(int item)
{
struct node *tmp;
tmp=(struct node *)malloc(sizeof(struct node));
tmp->info=item;
if(tmp==NULL)
{
printf("\nMemory not available\n");
return;
}

if( isEmpty() ) /*If queue is empty */
{
rear=tmp;
tmp->link=rear;
}
else
{
tmp->link=rear->link;
```

```
rear->link=tmp;
rear=tmp;
}
}/*End of insert()*/

del()
{
int item;
struct node *tmp;
if( isEmpty() )
{
printf("\nQueue underflow\n");
exit(1);
}
if(rear->link==rear)  /*If only one element*/
{
tmp=rear;
rear=NULL;
}
else
{
tmp=rear->link;
rear->link=rear->link->link;
}
item=tmp->info;
free(tmp);
return item;
}/*End of del()*/

int peek()
{
if( isEmpty() )
{
printf("\nQueue underflow\n");
exit(1);
}
return rear->link->info;
}/* End of peek() */

int isEmpty()
{
if( rear == NULL )
return 1;
else
return 0;
}/*End of isEmpty()*/


void display()
{
```

```
struct node *p;
if(isEmpty())
{
printf("\nQueue is empty\n");
return;
}
printf("\nQueue is :\n");
p=rear->link;
do
{
printf("%d ",p->info);
p=p->link;
}while(p!=rear->link);
printf("\n");
}/*End of display()*/
```

OUTPUT

```
1.Insert
2.Delete
3.Peek
4.Display
5.Quit

Enter your choice : 1

Enter the element for insertion : 1

1.Insert
2.Delete
3.Peek
4.Display
5.Quit

Enter your choice : 1

Enter the element for insertion : 2

1.Insert
2.Delete
3.Peek
4.Display
5.Quit

Enter your choice : 1

Enter the element for insertion : 3

1.Insert
2.Delete
3.Peek
```

```
4.Display
5.Quit

Enter your choice : 4

Queue is :
1 2 3

1.Insert
2.Delete
3.Peek
4.Display
5.Quit

Enter your choice : 2

Deleted element is 1

1.Insert
2.Delete
3.Peek
4.Display
5.Quit

Enter your choice : 3

Item at the front of queue is 2

1.Insert
2.Delete
3.Peek
4.Display
5.Quit

Enter your choice : 2

Deleted element is 2

1.Insert
2.Delete
3.Peek
4.Display
5.Quit

Enter your choice : 4

Queue is :
3

1.Insert
```

```
2.Delete
3.Peek
4.Display
5.Quit

Enter your choice : 2

Deleted element is 3

1.Insert
2.Delete
3.Peek
4.Display
5.Quit

Enter your choice : 2

Queue underflow
```

Experiment No.	4.		
Type of problem	Tree & Graph using Linked List		
Students are required to implement/execute/draw/make document	1. Algorithm 2. Program 3. Output 4. Final document with evaluator signature.		
List of experiments	1. Write a program to implement BFS using linked list. 2. Write a program to implement tree traversals using linked list. 3. Write a program to implement DFS using linked list.		

Tree Node
The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

In a tree, all nodes share common construct.
BST Basic Operations
The basic operations that can be performed on a binary search tree data structure, are the following −
Insert − Inserts an element in a tree/create a tree.
Search − Searches an element in a tree.
Preorder Traversal − Traverses a tree in a pre-order manner.
Inorder Traversal − Traverses a tree in an in-order manner.
Postorder Traversal − Traverses a tree in a post-order manner.
We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.
Insert Operation
The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.
Algorithm

```
If root is NULL
   then create root node
return

If root exists then
   compare the data with node.data
```

```
  while until insertion position is located

    If data is greater than node.data
      goto right subtree
    else
      goto left subtree

  endwhile

  insert data

end If
```

Implementation

The implementation of insert function should look like this −

```
void insert(int data) {
  struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;
  struct node *parent;

  tempNode->data = data;
  tempNode->leftChild = NULL;
  tempNode->rightChild = NULL;

  //if tree is empty, create root node
  if(root == NULL) {
    root = tempNode;
  } else {
    current = root;
    parent  = NULL;

    while(1) {
      parent = current;

      //go to left of the tree
      if(data < parent->data) {
        current = current->leftChild;

        //insert to the left
        if(current == NULL) {
          parent->leftChild = tempNode;
          return;
        }
      }

      //go to right of the tree
      else {
        current = current->rightChild;

        //insert to the right
        if(current == NULL) {
               parent->rightChild = tempNode;
               return;
            }
         }
      }
   }
}
```
Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm
```
If root.data is equal to search.data
   return root
else
   while data not found

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree

      If data found
         return node

   endwhile

   return data not found

end if
```
The implementation of this algorithm should look like this.
```
struct node* search(int data) {
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data) {
      if(current != NULL)
      printf("%d ",current->data);

      //go to left tree

      if(current->data > data) {
         current = current->leftChild;
      }
      //else go to right tree
      else {
         current = current->rightChild;
      }

      //not found
      if(current == NULL) {
         return NULL;
      }

      return current;
   }
}
```

Write a program to implement BFS using linked list.

```
/*
 * C Program to Display the Nodes of a Tree using BFS Traversal
 *                40
 *                /\
 *               20 60
 *               /\  \
 *             10 30  80
 *                     \
 *                      90
 */
#include <stdio.h>
#include <stdlib.h>

struct btnode
{
    int value;
    struct btnode *left, *right;
};
typedef struct btnode node;

/* function declarations */
void insert(node *, node *);
void bfs_traverse(node *);

/*global declarations */
node *root = NULL;
int val, front = 0, rear = -1, i;
int queue[20];

void main()
{
    node *new = NULL ;
    int num = 1;
    printf("Enter the elements of the tree(enter 0 to exit)\n");
    while (1)
    {
        scanf("%d",  &num);
        if (num  ==  0)
```

```
            break;
        new = malloc(sizeof(node));
        new->left = new->right = NULL;
        new->value = num;
        if (root == NULL)
            root = new;
        else
        {
            insert(new, root);
        }
    }
    printf("elements in a tree in inorder are\n");
    queue[++rear] = root->value;
    bfs_traverse(root);
    for (i = 0;i <= rear;i++)
        printf("%d -> ", queue[i]);
    printf("%d\n", root->right->right->right->value);
}

/* inserting nodes of a tree */
void insert(node * new , node *root)
{
    if (new->value>root->value)
    {
        if (root->right == NULL)
            root->right = new;
        else
            insert (new, root->right);
    }
    if (new->value < root->value)
    {
        if (root->left  ==  NULL)
            root->left = new;
        else
            insert (new, root->left);
    }
}

/* displaying elements using BFS traversal */
void bfs_traverse(node *root)
{
    val = root->value;
    if ((front <= rear)&&(root->value == queue[front]))
    {
        if (root->left != NULL)
            queue[++rear] = root->left->value;
        if (root->right != NULL || root->right  ==  NULL)
            queue[++rear] = root->right->value;
        front++;
    }
    if (root->left != NULL)
    {
        bfs_traverse(root->left);
    }
    if (root->right != NULL)
    {
        bfs_traverse(root->right);
    }
}
```

```
OUTPUT
Enter the elements of the tree(enter 0 to exit)
40
20
10
30
60
70
80
0
elements in a tree in inorder are
40 -> 20 -> 60 -> 10 -> 30 -> 70 -> 80
```

Write a program to implement tree traversals using linked list.

```
/*
 * C Program to Construct a Binary Search Tree and perform deletion, inorder
traversal on it
 */
#include <stdio.h>
#include <stdlib.h>

struct btnode
{
    int value;
    struct btnode *l;
    struct btnode *r;
}*root = NULL, *temp = NULL, *t2, *t1;

void delete1();
void insert();
void delete();
void inorder(struct btnode *t);
void create();
void search(struct btnode *t);
void preorder(struct btnode *t);
void postorder(struct btnode *t);
void search1(struct btnode *t,int data);
int smallest(struct btnode *t);
int largest(struct btnode *t);

int flag = 1;

void main()
{
   int ch;

   printf("\nOPERATIONS ---");
   printf("\n1 - Insert an element into tree\n");
   printf("2 - Delete an element from the tree\n");
   printf("3 - Inorder Traversal\n");
   printf("4 - Preorder Traversal\n");
   printf("5 - Postorder Traversal\n");
   printf("6 - Exit\n");
   while(1)
   {
      printf("\nEnter your choice : ");
      scanf("%d", &ch);
      switch (ch)
      {
      case 1:
         insert();
         break;
      case 2:
         delete();
         break;
      case 3:
         inorder(root);
         break;
      case 4:
         preorder(root);
         break;
      case 5:
         postorder(root);
         break;
      case 6:
         exit(0);
      default :
         printf("Wrong choice, Please enter correct choice  ");
         break;
      }
   }
}

/* To insert a node in the tree */
void insert()
{
    create();
    if (root == NULL)
        root = temp;
    else
        search(root);
}

/* To create a node */
void create()
{
    int data;

    printf("Enter data of node to be inserted : ");
    scanf("%d", &data);
    temp = (struct btnode *)malloc(1*sizeof(struct btnode));
    temp->value = data;
    temp->l = temp->r = NULL;
}

/* Function to search the appropriate position to insert the new node */
void search(struct btnode *t)
{
    if ((temp->value > t->value) && (t->r != NULL))      /* value more than root node
value insert at right */
       search(t->r);
   else if ((temp->value > t->value) && (t->r == NULL))
        t->r = temp;
    else if ((temp->value < t->value) && (t->l != NULL))    /* value less than root
node value insert at left */
       search(t->l);
    else if ((temp->value < t->value) && (t->l == NULL))
        t->l = temp;
}

/* recursive function to perform inorder traversal of tree */
void inorder(struct btnode *t)
{
    if (root == NULL)
    {
        printf("No elements in a tree to display");
        return;
    }
    if (t->l != NULL)
        inorder(t->l);
    printf("%d -> ", t->value);
    if (t->r != NULL)
        inorder(t->r);
}

/* To check for the deleted node */
```

```
void delete()
{
   int data;

   if (root == NULL)
   {
      printf("No elements in a tree to delete");
      return;
   }
   printf("Enter the data to be deleted : ");
   scanf("%d", &data);
   t1 = root;
   t2 = root;
   search1(root, data);
}

/* To find the preorder traversal */
void preorder(struct btnode *t)
{
   if (root == NULL)
   {
      printf("No elements in a tree to display");
      return;
   }
   printf("%d -> ", t->value);
   if (t->l != NULL)
      preorder(t->l);
   if (t->r != NULL)
      preorder(t->r);
}

/* To find the postorder traversal */
void postorder(struct btnode *t)
{
   if (root == NULL)
   {
      printf("No elements in a tree to display ");
      return;
   }
   if (t->l != NULL)
      postorder(t->l);
   if (t->r != NULL)
      postorder(t->r);
   printf("%d -> ", t->value);
}

/* Search for the appropriate position to insert the new node */
void search1(struct btnode *t, int data)
{
   if ((data>t->value))
```

```
    {
       t1 = t;
       search1(t->r, data);
    }
    else if ((data < t->value))
    {
       t1 = t;
       search1(t->l, data);
    }
    else if ((data==t->value))
    {
       delete1(t);
    }
}

/* To delete a node */
void delete1(struct btnode *t)
{
    int k;

    /* To delete leaf node */
    if ((t->l == NULL) && (t->r == NULL))
    {
       if (t1->l == t)
       {
          t1->l = NULL;
       }
       else
       {
          t1->r = NULL;
       }
       t = NULL;
       free(t);
       return;
    }

    /* To delete node having one left hand child */
    else if ((t->r == NULL))
    {
       if (t1 == t)
       {
          root = t->l;
          t1 = root;
       }
       else if (t1->l == t)
       {
          t1->l = t->l;

       }
       else
```

```
        {
            t1->r = t->l;
        }
        t = NULL;
        free(t);
        return;
    }

    /* To delete node having right hand child */
    else if (t->l == NULL)
    {
        if (t1 == t)
        {
            root = t->r;
            t1 = root;
        }
        else if (t1->r == t)
            t1->r = t->r;
        else
            t1->l = t->r;
        t == NULL;
        free(t);
        return;
    }

    /* To delete node having two child */
    else if ((t->l != NULL) && (t->r != NULL))
    {
        t2 = root;
        if (t->r != NULL)
        {
            k = smallest(t->r);
            flag = 1;
        }
        else
        {
            k =largest(t->l);
            flag = 2;
        }
        search1(root, k);
        t->value = k;
    }

}

/* To find the smallest element in the right sub tree */
int smallest(struct btnode *t)
{
    t2 = t;
    if (t->l != NULL)
    {
        t2 = t;
        return(smallest(t->l));
    }
    else
        return (t->value);
}

/* To find the largest element in the left sub tree */
int largest(struct btnode *t)
{
    if (t->r != NULL)
    {
        t2 = t;
        return(largest(t->r));
    }
    else
        return(t->value);
}
```

OUTPUT

```
OPERATIONS ---
1 - Insert an element into tree
2 - Delete an element from the tree
3 - Inorder Traversal
4 - Preorder Traversal
5 - Postorder Traversal
6 - Exit

Enter your choice : 1
Enter data of node to be inserted : 40

Enter your choice : 1
Enter data of node to be inserted : 20

Enter your choice : 1
Enter data of node to be inserted : 10

Enter your choice : 1
Enter data of node to be inserted : 30

Enter your choice : 1
Enter data of node to be inserted : 60

Enter your choice : 1
Enter data of node to be inserted : 80

Enter your choice : 1
Enter data of node to be inserted : 90
Enter your choice : 3
10 -> 20 -> 30 -> 40 -> 60 -> 80 -> 90 ->

         40
         /\
        /  \
      20    60
      / \     \
    10  30    80
                \
                90
```

3. Write a program to implement DFS using linked list.

```
/*
 * C Program for Depth First Binary Tree Search using Recursion
 */
#include <stdio.h>
#include <stdlib.h>

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

void generate(struct node **, int);
void DFS(struct node *);
void delete(struct node **);

int main()
{
   struct node *head = NULL;
   int choice = 0, num, flag = 0, key;

   do
   {
      printf("\nEnter your choice:\n1. Insert\n2. Perform DFS Traversal\n3.
Exit\nChoice: ");
      scanf("%d", &choice);
      switch(choice)
      {
      case 1:
         printf("Enter element to insert: ");
         scanf("%d", &num);
         generate(&head, num);
         break;
```

```
        case 2:
            DFS(head);
            break;
        case 3:
            delete(&head);
            printf("Memory Cleared\nPROGRAM TERMINATED\n");
            break;
        default:
            printf("Not a valid input, try again\n");
        }
    } while (choice != 3);
    return 0;
}

void generate(struct node **head, int num)
{
    struct node *temp = *head, *prev = *head;

    if (*head == NULL)
    {
        *head = (struct node *)malloc(sizeof(struct node));
        (*head)->a = num;
        (*head)->left = (*head)->right = NULL;
    }
    else
    {
        while (temp != NULL)
        {
            if (num > temp->a)
            {
                prev = temp;
                temp = temp->right;
            }
            else
            {
                prev = temp;
                temp = temp->left;
            }
        }
        temp = (struct node *)malloc(sizeof(struct node));
        temp->a = num;
        if (num >= prev->a)
        {
            prev->right = temp;
        }
        else
        {
            prev->left = temp;
        }
    }
```

```
}

void DFS(struct node *head)
{
   if (head)
   {
      if (head->left)
      {
         DFS(head->left);
      }
      if (head->right)
      {
         DFS(head->right);
      }
      printf("%d  ", head->a);
   }
}

void delete(struct node **head)
{
   if (*head != NULL)
   {
      if ((*head)->left)
      {
         delete(&(*head)->left);
      }
      if ((*head)->right)
      {
         delete(&(*head)->right);
      }
      free(*head);
   }
}
```

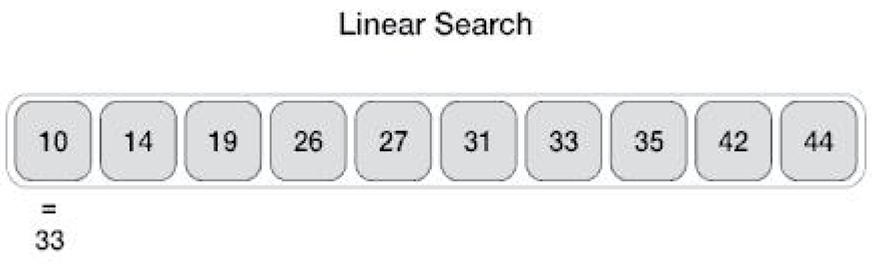
OUTPUT

```
Enter your choice:
1. Insert
2. Perform DFS Traversal
3. Exit
Choice: 1
Enter element to insert: 5

Enter your choice:
1. Insert
2. Perform DFS Traversal
3. Exit
Choice: 1
Enter element to insert: 3

Enter your choice:
```

```
1. Insert
2. Perform DFS Traversal
3. Exit
Choice: 1
Enter element to insert: 4

Enter your choice:
1. Insert
2. Perform DFS Traversal
3. Exit
Choice: 1
Enter element to insert: 2

Enter your choice:
1. Insert
2. Perform DFS Traversal
3. Exit
Choice: 1
Enter element to insert: 7

Enter your choice:
1. Insert
2. Perform DFS Traversal
3. Exit
Choice: 1
Enter element to insert: 8

Enter your choice:
1. Insert
2. Perform DFS Traversal
3. Exit
Choice: 1
Enter element to insert: 6

Enter your choice:
1. Insert
2. Perform DFS Traversal
3. Exit
Choice: 2
2  4  3  6  8  7  5
Enter your choice:
1. Insert
2. Perform DFS Traversal
3. Exit
Choice: 3
Memory Cleared
PROGRAM TERMINATED
```

Experiment No.	5.		
Type of problem	Linear Search		
Students are required to implement/execute/draw/make document	1. Algorithm 2. Program 3. Output 4. Final document with evaluator signature.		
List of experiments	1.Write a program on linear search 2.Write a program to compare two numbers 3.Write a program to access array elements 4.Write a program on linear search based on link-list		

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



Algorithm
Linear Search (Array A, Value x)

```
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
```

Pseudocode

```
procedure linear_search (list, value)

  for each item in the list

    if match item == value

      return the item's location
    end if

  end for

end procedure
```

Write a program on linear search

```
/*
 * C program to input N numbers and store them in an array.
 * Do a linear search for a given key and report success
 * or failure.
 */
#include <stdio.h>

void main()
{
   int array[10];
   int i, num, keynum, found = 0;

   printf("Enter the value of num \n");
   scanf("%d", &num);
   printf("Enter the elements one by one \n");
   for (i = 0; i < num; i++)
   {
      scanf("%d", &array[i]);
   }
   printf("Input array is \n");
   for (i = 0; i < num; i++)
   {
      printf("%dn", array[i]);
   }
   printf("Enter the element to be searched \n");
   scanf("%d", &keynum);
   /*  Linear search begins */
   for (i = 0; i < num ; i++)
   {
      if (keynum == array[i] )
      {
         found = 1;
         break;
      }
   }
   if (found == 1)
      printf("Element is present in the array\n");
   else
      printf("Element is not present in the array\n");
}
```

OUTPUT

```
Enter the value of num
```

```
5
Enter the elements one by one
456
78
90
40
100
Input array is
456
78
90
40
100
Enter the element to be searched
70
Element is not present in the array

Enter the value of num
7
Enter the elements one by one
45
56
89
56
90
23
10
Input array is
45
56
89
56
90
23
10
Enter the element to be searched
45
Element is present in the array
```

2. Write a program to compare two numbers

```
/*
 * C program to accept two integers and check if they are equal
 */
#include <stdio.h>
void main()
{
   int m, n;
```

```
    printf("Enter the values for M and N\n");
    scanf("%d %d", &m, &n);
    if (m == n)
        printf("M and N are equal\n");
    else
        printf("M and N are not equal\n");
}
```

Program Explanation

1. Take the two integers as input and store it in the variables m and n respectively.
2. Using if, else statements check if m is equal to n.
3. If they are equal, then print the output as “M and N are equal”.
4. Otherwise print it as “M and N are not equal”.

OUTPUT

```
Case:1
Enter the values for M and N
3 3
M and N are equal

Case:2
Enter the values for M and N
5 8
M and N are not equal
```

3. Write a program to access array elements

```
*/
* C program to read and print n elements in an array
*/

#include <stdio.h>
#define MAX_SIZE 1000 //Maximum size of the array

int main()
{
    int arr[MAX_SIZE]; //Declares an array of MAX_SIZE
    int i, N;

    /* Input size of the array */
    printf("Enter size of array: ");
    scanf("%d", &N);

    /* Input elements in array */
    printf("Enter %d elements in the array : ", N);
    for(i=0; i<N; i++)
    {
        scanf("%d", &arr[i]);
    }

    /*
     * Print all elements of array
     */
    printf("\nElements in array are: ");
    for(i=0; i<N; i++)
    {
        printf("%d, ", arr[i]);
    }

    return 0;
}
```

OUTPUT

```
Enter size of array: 10
Enter 10 elements in the array: 10
20
30
40
50
60
70
80
90
100

Elements in array are: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
```

4. Write a program on linear search based on link-list

```
/*
  * C Program to Search for an Element in the Linked List without
   * using Recursion       */

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

  struct node
  {
      int a;
      struct node *next;
  };

  void generate(struct node **, int);
  void search(struct node *, int);
  void delete(struct node **);
```

```
int main()
{
   struct node *head = NULL;
   int key, num;

   printf("Enter the number of nodes: ");
   scanf("%d", &num);
   printf("\nDisplaying the list\n");
   generate(&head, num);
   printf("\nEnter key to search: ");
   scanf("%d", &key);
   search(head, key);
   delete(&head);

   return 0;
}

void generate(struct node **head, int num)
{
   int i;
   struct node *temp;

   for (i = 0; i < num; i++)
   {
      temp = (struct node *)malloc(sizeof(struct node));
      temp->a = rand() % num;
      if (*head == NULL)
      {
         *head = temp;
         temp->next = NULL;
      }
      else
      {
         temp->next = *head;
         *head = temp;
      }
      printf("%d  ", temp->a);
   }
}

void search(struct node *head, int key)
{
   while (head != NULL)
   {
      if (head->a == key)
      {
         printf("key found\n");
         return;
      }
```

```
        head = head->next;
    }
    printf("Key not found\n");
}

void delete(struct node **head)
{
    struct node *temp;

    while (*head != NULL)
    {
        temp = *head;
        *head = (*head)->next;
        free(temp);
    }
}
```

OUTPUT

```
Enter the number of nodes: 10

Displaying the list
3  6  7  5  3  5  6  2  9  1
Enter key to search: 2
key found
```

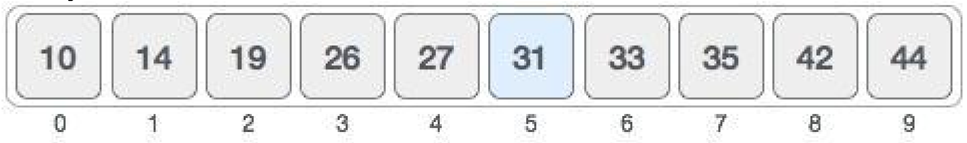
Experiment No.	6.		
Type of problem	Binary Search		
Students are required to implement/execute/draw/make document	1. Algorithm 2. Program 3. Output 4. Final document with evaluator signature.		
List of experiments	1.Write a program on Binary Search 2.Write a program to access array elements randomly 3.Write a program on multi-way Search		

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

How Binary Search Works?

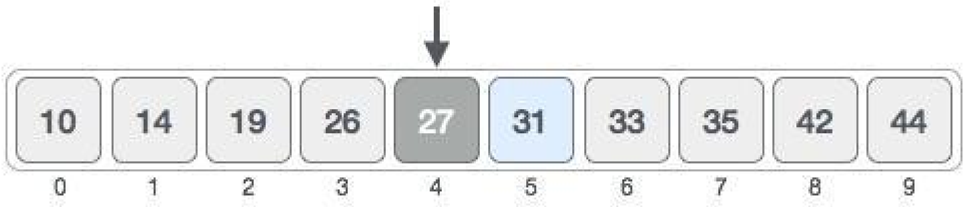
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



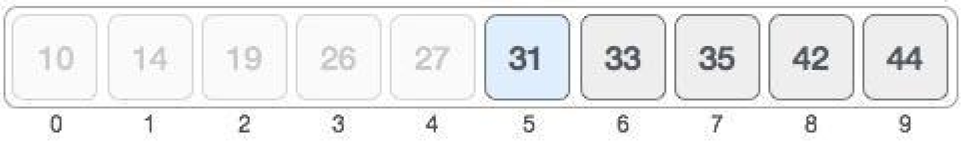
First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

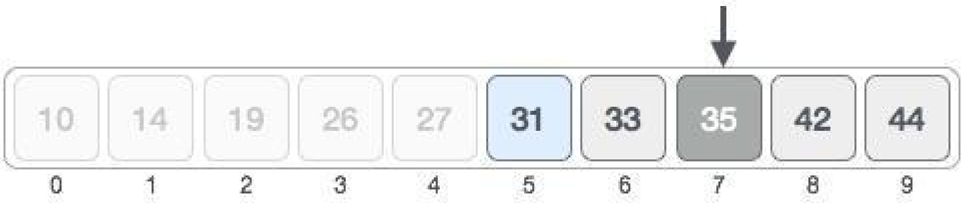
Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



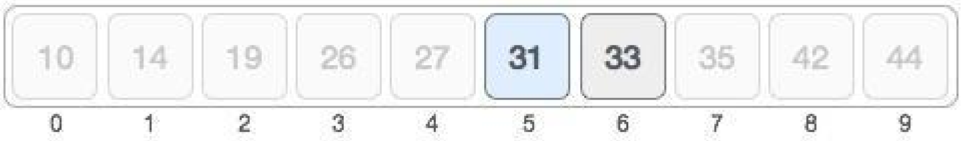
Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



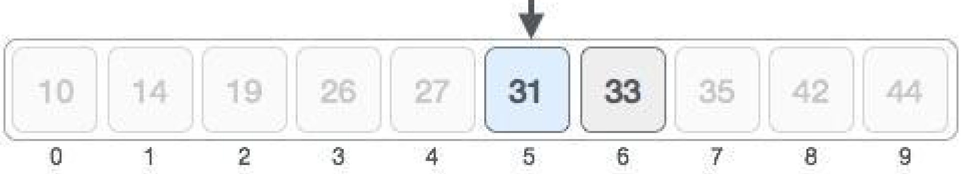
We change our low to mid + 1 and find the new mid value again.
low = mid + 1
mid = low + (high - low) / 2
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



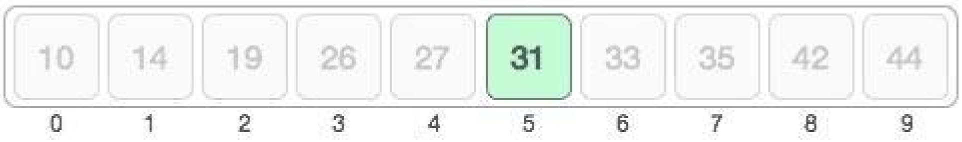
The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.
Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.
Pseudocode
The pseudocode of binary search algorithms should look like this −

```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 1
  Set upperBound = n

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exists.
      set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

      if A[midPoint] < x
         set lowerBound = midPoint + 1

      if A[midPoint] > x
         set upperBound = midPoint - 1

      if A[midPoint] = x
         EXIT: x found at location midPoint

   end while

end procedure
```

Write a program on Binary Search

```
/* C Program - Binary Search */

#include<stdio.h>
#include<conio.h>
void main()
{
        clrscr();
        int n, i, arr[50], search, first, last, middle;
        printf("Enter total number of elements :");
        scanf("%d",&n);
        printf("Enter %d number :", n);
        for (i=0; i<n; i++)
        {
                scanf("%d",&arr[i]);
        }
        printf("Enter a number to find :");
        scanf("%d", &search);
        first = 0;
        last = n-1;
        middle = (first+last)/2;
        while (first <= last)
        {
                if(arr[middle] < search)
                {
                        first = middle + 1;

                }
                else if(arr[middle] == search)
                {
                        printf("%d found at location %d\n", search, middle+1);
                        break;
                }
                else
```

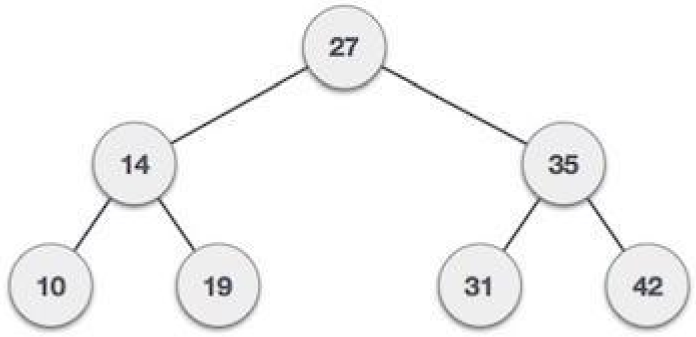
```
                    {
                            last = middle - 1;
                    }
                    middle = (first + last)/2;
            }
            if(first > last)
            {
                    printf("Not found! %d is not present in the list.",search);
            }
            getch();
    }
```

OUTPUT

```
Enter total number of elements: 5
Enter 5 numbers: 12
23
34
45
56
Enter a number to find: 34
34 found at location 3
```

Experiment No.	7.		
Type of problem	Binary Search Tree (BST)		
Students are required to implement/execute/draw/make document	1. Algorithm 2. Program 3. Output 4. Final document with evaluator signature.		
List of experiments	1.Write a program to implement BST 2.Write a program to create binary tree using pointers 3.Write a program to traverse binary tree		

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −
The left sub-tree of a node has a key less than or equal to its parent node's key.
The right sub-tree of a node has a key greater than to its parent node's key.
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −
left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)
Representation
BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.
Following is a pictorial representation of BST −



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.
Basic Operations
Following are the basic operations of a tree −
Search − Searches an element in a tree.
Insert − Inserts an element in a tree.
Pre-order Traversal − Traverses a tree in a pre-order manner.
In-order Traversal − Traverses a tree in an in-order manner.
Post-order Traversal − Traverses a tree in a post-order manner.
Node
Define a node having some data, references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
```

```
  struct node *rightChild;
};
```

Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
struct node* search(int data){
  struct node *current = root;
  printf("Visiting elements: ");

  while(current->data != data){

    if(current != NULL) {
      printf("%d ",current->data);

      //go to left tree
      if(current->data > data){
        current = current->leftChild;
      }//else go to right tree
      else {
        current = current->rightChild;
      }

      //not found
      if(current == NULL){
        return NULL;
      }
    }
  }
  return current;
}
```

Insert Operation

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
void insert(int data) {
  struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;
  struct node *parent;

  tempNode->data = data;
  tempNode->leftChild = NULL;
  tempNode->rightChild = NULL;

  //if tree is empty
  if(root == NULL) {
    root = tempNode;
```

```
   } else {
      current = root;
      parent = NULL;

      while(1) {
         parent = current;

         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;
            //insert to the left

            if(current == NULL) {
               parent->leftChild = tempNode;
               return;
            }
         }//go to right of the tree
         else {
            current = current->rightChild;

            //insert to the right
            if(current == NULL) {
               parent->rightChild = tempNode;
               return;
            }
         }
      }
   }
}
```

Write a program to implement BST

```
/*
 * C Program to Construct a Binary Search Tree and perform deletion, inorder
traversal on it
 */
#include <stdio.h>
#include <stdlib.h>

struct btnode
{
    int value;
    struct btnode *l;
    struct btnode *r;
}*root = NULL, *temp = NULL, *t2, *t1;

void delete1();
void insert();
void delete();
void inorder(struct btnode *t);
void create();
void search(struct btnode *t);
void preorder(struct btnode *t);
void postorder(struct btnode *t);
void search1(struct btnode *t,int data);
int smallest(struct btnode *t);
int largest(struct btnode *t);

int flag = 1;

void main()
{
   int ch;

   printf("\nOPERATIONS ---");
   printf("\n1 - Insert an element into tree\n");
   printf("2 - Delete an element from the tree\n");
   printf("3 - Inorder Traversal\n");
   printf("4 - Preorder Traversal\n");
   printf("5 - Postorder Traversal\n");
   printf("6 - Exit\n");
   while(1)
   {
      printf("\nEnter your choice : ");
      scanf("%d", &ch);
      switch (ch)
      {
      case 1:
         insert();
         break;
      case 2:
         delete();
         break;
      case 3:
         inorder(root);
         break;
      case 4:
         preorder(root);
         break;
      case 5:
         postorder(root);
         break;
      case 6:
         exit(0);
      default :
         printf("Wrong choice, Please enter correct choice  ");
         break;
      }
   }
}
```

```
/* To insert a node in the tree */
void insert()
{
   create();
   if (root == NULL)
      root = temp;
   else
      search(root);
}

/* To create a node */
void create()
{
   int data;

   printf("Enter data of node to be inserted : ");
   scanf("%d", &data);
   temp = (struct btnode *)malloc(1*sizeof(struct btnode));
   temp->value = data;
   temp->l = temp->r = NULL;
}
 /* Function to search the appropriate position to insert the new node */
void search(struct btnode *t)
{
   if ((temp->value > t->value) && (t->r != NULL))      /* value more than root node
value insert at right */
      search(t->r);
   else if ((temp->value > t->value) && (t->r == NULL))
      t->r = temp;
   else if ((temp->value < t->value) && (t->l != NULL))    /* value less than root
node value insert at left */
      search(t->l);
   else if ((temp->value < t->value) && (t->l == NULL))
      t->l = temp;
}

/* recursive function to perform inorder traversal of tree */
void inorder(struct btnode *t)
{
   if (root == NULL)
   {
      printf("No elements in a tree to display");
      return;
   }
   if (t->l != NULL)
      inorder(t->l);
   printf("%d -> ", t->value);
   if (t->r != NULL)
      inorder(t->r);
}
```

```
/* To check for the deleted node */
void delete()
{
   int data;

   if (root == NULL)
   {
      printf("No elements in a tree to delete");
      return;
   }
   printf("Enter the data to be deleted : ");
   scanf("%d", &data);
   t1 = root;
   t2 = root;
   search1(root, data);
}

/* To find the preorder traversal */
void preorder(struct btnode *t)
{
   if (root == NULL)
   {
      printf("No elements in a tree to display");
      return;
   }
   printf("%d -> ", t->value);
   if (t->l != NULL)
      preorder(t->l);
   if (t->r != NULL)
      preorder(t->r);
}

/* To find the postorder traversal */
void postorder(struct btnode *t)
{
   if (root == NULL)
   {
      printf("No elements in a tree to display ");
      return;
   }
   if (t->l != NULL)
      postorder(t->l);
   if (t->r != NULL)
      postorder(t->r);
   printf("%d -> ", t->value);
}

/* Search for the appropriate position to insert the new node */
void search1(struct btnode *t, int data)
```

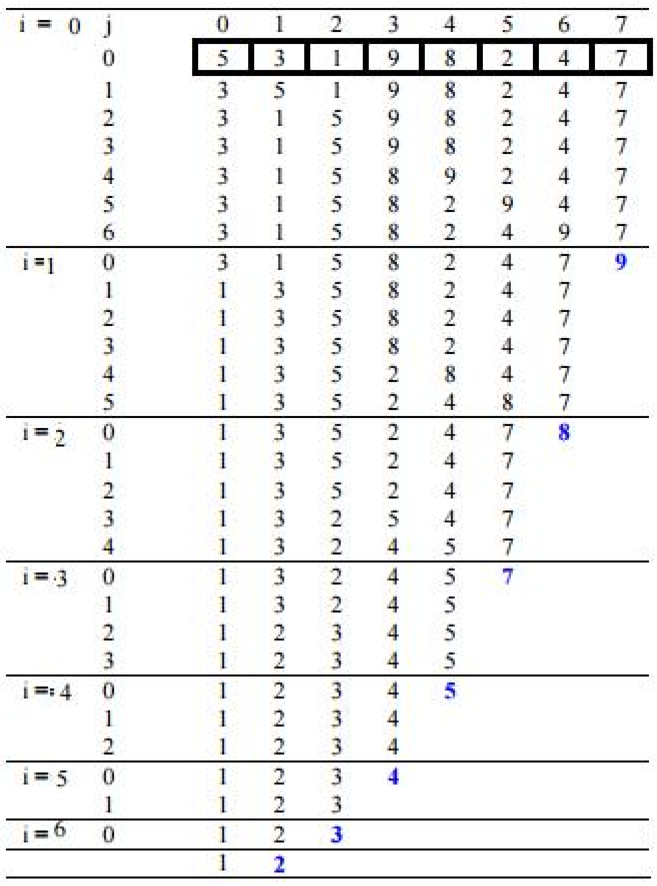
```
{
   if ((data>t->value))
   {
      t1 = t;
      search1(t->r, data);
   }
   else if ((data < t->value))
   {
      t1 = t;
      search1(t->l, data);
   }
   else if ((data==t->value))
   {
      delete1(t);
   }
}

/* To delete a node */
void delete1(struct btnode *t)
{
   int k;

   /* To delete leaf node */
   if ((t->l == NULL) && (t->r == NULL))
   {
      if (t1->l == t)
      {
         t1->l = NULL;
      }
      else
      {
         t1->r = NULL;
      }
      t = NULL;
      free(t);
      return;
   }

   /* To delete node having one left hand child */
   else if ((t->r == NULL))
   {
      if (t1 == t)
      {
         root = t->l;
         t1 = root;
      }
      else if (t1->l == t)
      {
         t1->l = t->l;
```

```
        }
        else
        {
            t1->r = t->l;
        }
        t = NULL;
        free(t);
        return;
    }

    /* To delete node having right hand child */
    else if (t->l == NULL)
    {
        if (t1 == t)
        {
            root = t->r;
            t1 = root;
        }
        else if (t1->r == t)
            t1->r = t->r;
        else
            t1->l = t->r;
        t == NULL;
        free(t);
        return;
    }

    /* To delete node having two child */
    else if ((t->l != NULL) && (t->r != NULL))
    {
        t2 = root;
        if (t->r != NULL)
        {
            k = smallest(t->r);
            flag = 1;
        }
        else
        {
            k =largest(t->l);
            flag = 2;
        }
        search1(root, k);
        t->value = k;
    }

}

/* To find the smallest element in the right sub tree */
int smallest(struct btnode *t)
{
    t2 = t;
    if (t->l != NULL)
    {
        t2 = t;
        return(smallest(t->l));
    }
    else
        return (t->value);
}

/* To find the largest element in the left sub tree */
int largest(struct btnode *t)
{
    if (t->r != NULL)
    {
        t2 = t;
        return (largest(t->r));
    }
    else
        return (t->value);
}
```

```
OUTPUT
OPERATIONS ---
1 - Insert an element into tree
2 - Delete an element from the tree
3 – In order Traversal
4 – Pre order Traversal
5 – Post order Traversal
6 - Exit

Enter your choice: 1
Enter data of node to be inserted: 40

Enter your choice: 1
Enter data of node to be inserted: 20

Enter your choice: 1
Enter data of node to be inserted: 10

Enter your choice: 1
Enter data of node to be inserted: 30

Enter your choice: 1
Enter data of node to be inserted: 60

Enter your choice: 1
Enter data of node to be inserted: 80

Enter your choice: 1
Enter data of node to be inserted: 90

Enter your choice: 3
10 -> 20 -> 30 -> 40 -> 60 -> 80 -> 90 ->

            40
            /\
           /  \
         20    60
         / \     \
       10  30   80
                  \
                   90
```

Experiment No.	8.		
Type of problem	Bubble sort		
Students are required to implement/execute/draw/make document	1. Algorithm 2. Program 3. Output 4. Final document with evaluator signature.		
List of experiments	1.Write a program on Bubble sort 2.Write a program to swap numbers 3.Write a program to use nested loops 4.Write a program on linear bubble sort		

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
Example:
First Pass:
(5 1 4 2 8) –> (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 5 4 2 8) –> (1 4 5 2 8), Swap since 5 > 4
(1 4 5 2 8) –> (1 4 2 5 8), Swap since 5 > 2
(1 4 2 5 8) –> (1 4 2 5 8), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
Second Pass:
(1 4 2 5 8) –> (1 4 2 5 8)
(1 4 2 5 8) –> (1 2 4 5 8), Swap since 4 > 2
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.
Third Pass:
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)
Worst and Average Case Time Complexity: O(n*n). Worst case occurs when array is reverse sorted.
Best Case Time Complexity: O(n). Best case occurs when array is already sorted.
Auxiliary Space: O(1)
Boundary Cases: Bubble sort takes minimum time (Order of n) when elements are already sorted.
Sorting In Place: Yes
Stable: Yes



i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 1	0	3	1	5	8	2	4	7	**9**
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	**8**	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	**7**		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	**5**			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	**4**				
	1	1	2	3					
i = 6	0	1	2	**3**					
		1	**2**						

Write a program on Bubble sort

C program for implementation of Bubble sort

```
#include <stdio.h>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)

       // Last i elements are already in place
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}
```

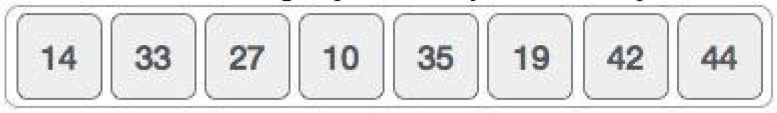
```
/* Function to print an array */
void printArray(int arr[], int size)
{
   int i;
   for (i=0; i < size; i++)
      printf("%d ", arr[i]);
   printf("n");
}

// Driver program to test above functions
int main()
{
   int arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr)/sizeof(arr[0]);
   bubbleSort(arr, n);
   printf("Sorted array: \n");
   printArray(arr, n);
   return 0;
}
```

OUTPUT:

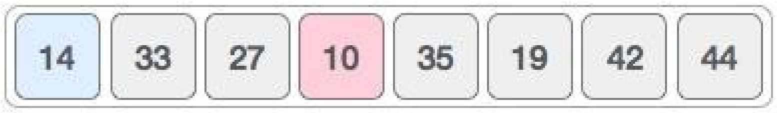
```
Sorted array:
11 12 22 25 34 64 90
```

Experiment No.	9.		
Type of problem	Selection sort		
Students are required to implement/execute/draw/make document	1. Algorithm 2. Program 3. Output 4. Final document with evaluator signature.		
List of experiments	1.Write a program on Selection sort 2.Write a program to find largest from array 3.Write a program on Selection sort based on link-list		

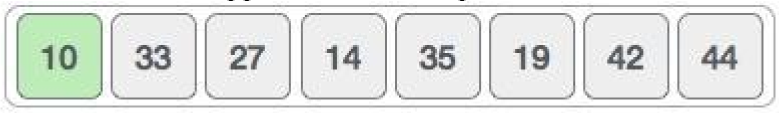
Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.
This algorithm is not suitable for large data sets as its average and worst case complexities are of O(n2), where n is the number of items.
How Selection Sort Works?
Consider the following depicted array as an example.



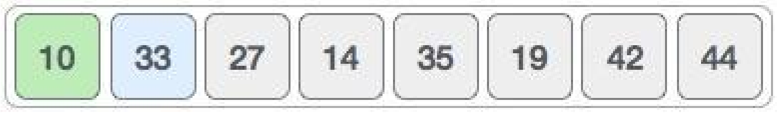
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



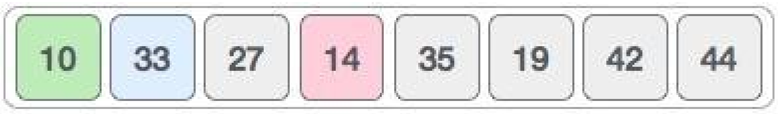
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



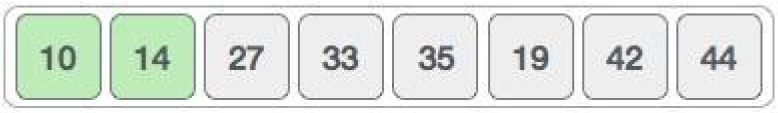
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

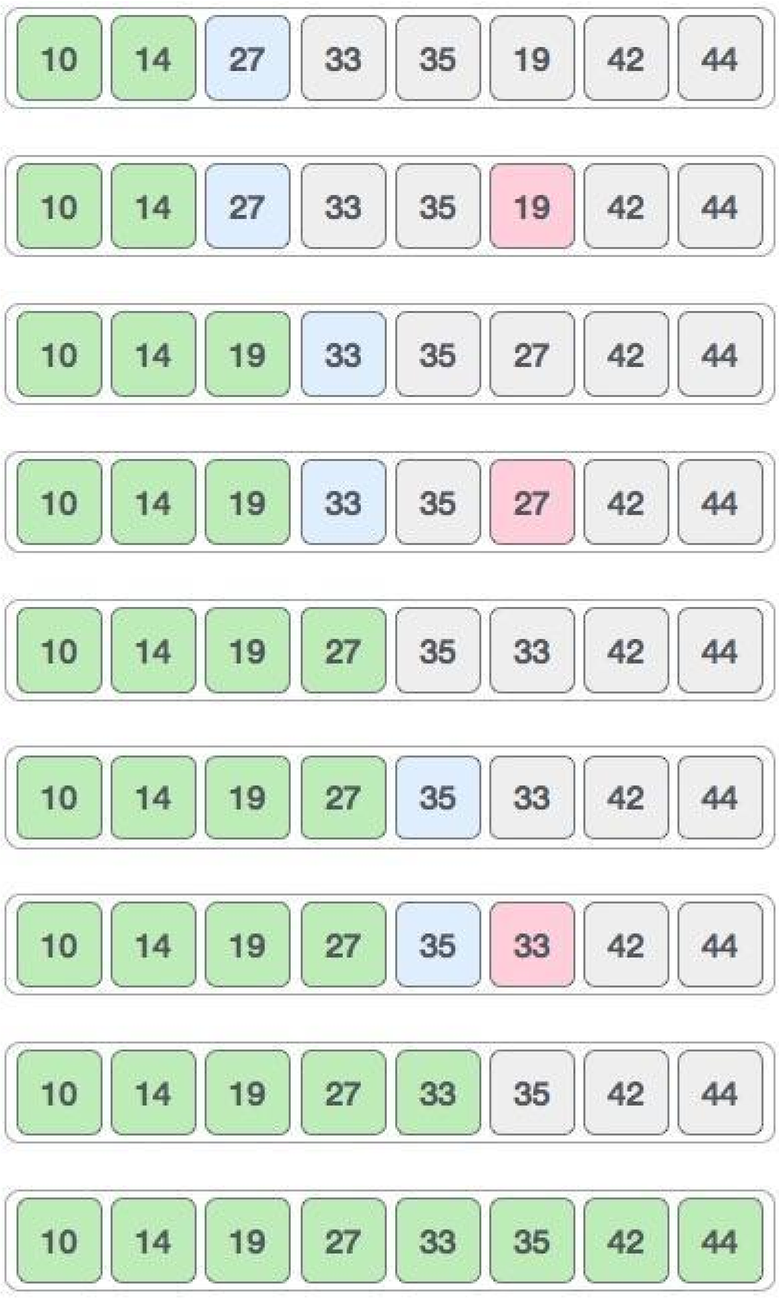


After two iterations, two least values are positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array.
Time Complexity: O(n2) as there are two nested loops.
Auxiliary Space: O(1)
The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

Following is a pictorial depiction of the entire sorting process −



Now, let us learn some programming aspects of selection sort.
Algorithm
Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted
Pseudocode

```
procedure selection sort
  list  : array of items
  n     : size of list

  for i = 1 to n - 1
  /* set current element as minimum*/
    min = i
```

```
      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if

   end for

end procedure
```

Write a program on Selection sort

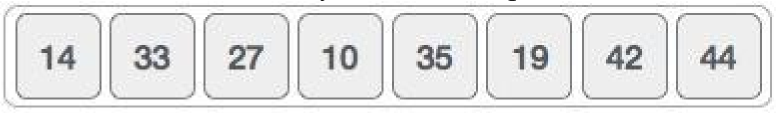
```
/ C program for implementation of selection sort
#include <stdio.h>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}

/* 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[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

OUTPUT

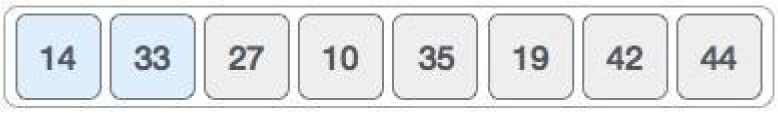
```
Sorted array:
11 12 22 25 64
```

Experiment No.	10.		
Type of problem	Insertion sort		
Students are required to implement/execute/draw/make document	1. Algorithm 2. Program 3. Output 4. Final document with evaluator signature.		
List of experiments	1.Write a program on Insertion sort 2.Write a program to insert a number into a array 3.Write a program on Insertion sort based on link-list		

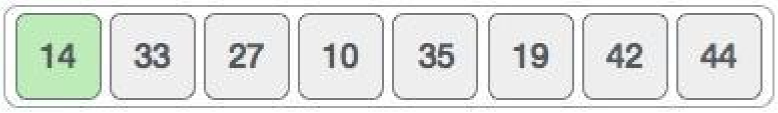
This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort. The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2), where n is the number of items.
How Insertion Sort Works?
We take an unsorted array for our example.



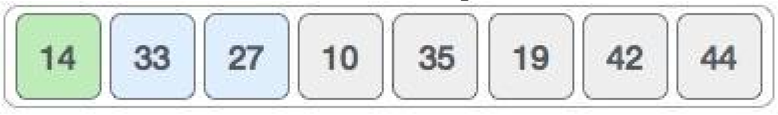
Insertion sort compares the first two elements.



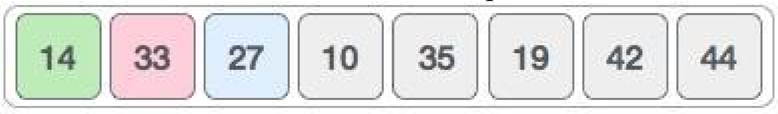
It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



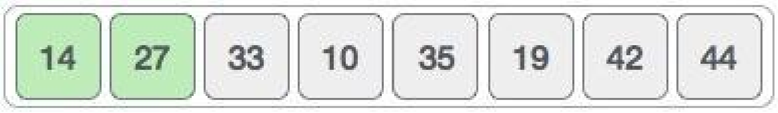
Insertion sort moves ahead and compares 33 with 27.



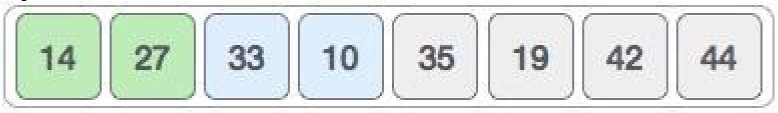
And finds that 33 is not in the correct position.



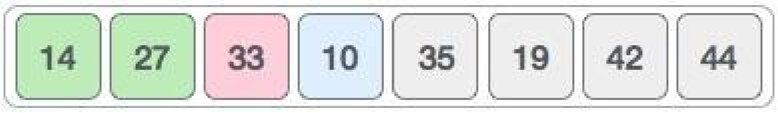
It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



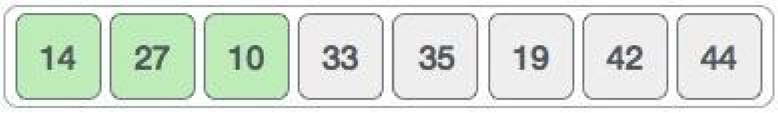
By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



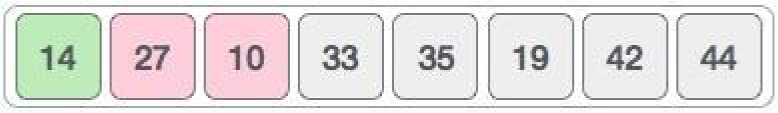
These values are not in a sorted order.



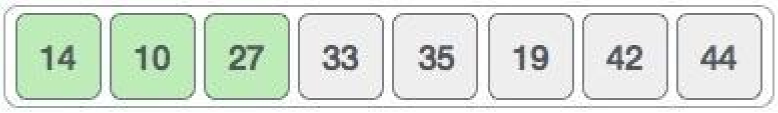
So we swap them.



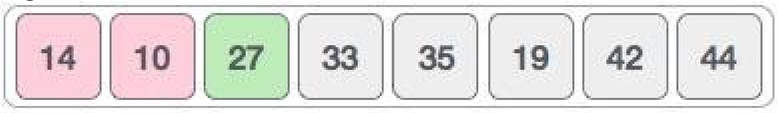
However, swapping makes 27 and 10 unsorted.



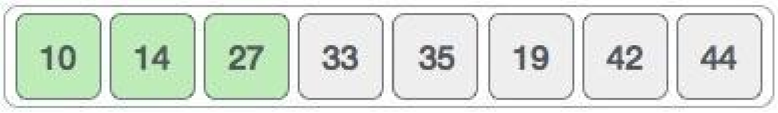
Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

```
Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the
         value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted
```

Pseudocode

```
procedure insertionSort( A : array of items )
   int holePosition
   int valueToInsert

   for i = 1 to length(A) inclusive do:
    /* select value to be inserted */
    valueToInsert = A[i]
    holePosition = i

    /*locate hole position for the element to be inserted */

    while holePosition > 0 and A[holePosition-1] > valueToInsert do:
      A[holePosition] = A[holePosition-1]
      holePosition = holePosition -1
    end while

    /* insert the number at hole position */
    A[holePosition] = valueToInsert

  end for

end procedure
```

Time Complexity: O(n*n)
Auxiliary Space: O(1)
Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.
Algorithmic Paradigm: Incremental Approach
Sorting In Place: Yes
Stable: Yes
Online: Yes
Uses: Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

Write a program on Insertion sort

```c
// C program for insertion sort
#include <stdio.h>
#include <math.h>

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
  int i, key, j;
  for (i = 1; i < n; i++)
  {
     key = arr[i];
     j = i-1;

     /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
     while (j >= 0 && arr[j] > key)
```

```
        {
           arr[j+1] = arr[j];
           j = j-1;
        }
        arr[j+1] = key;
    }
}

// A utility function ot print an array of size n
void printArray(int arr[], int n)
{
   int i;
   for (i=0; i < n; i++)
       printf("%d ", arr[i]);
   printf("\n");
}



/* Driver program to test insertion sort */
int main()
{
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

OUTPUT:

```
5 6 11 12 13
```

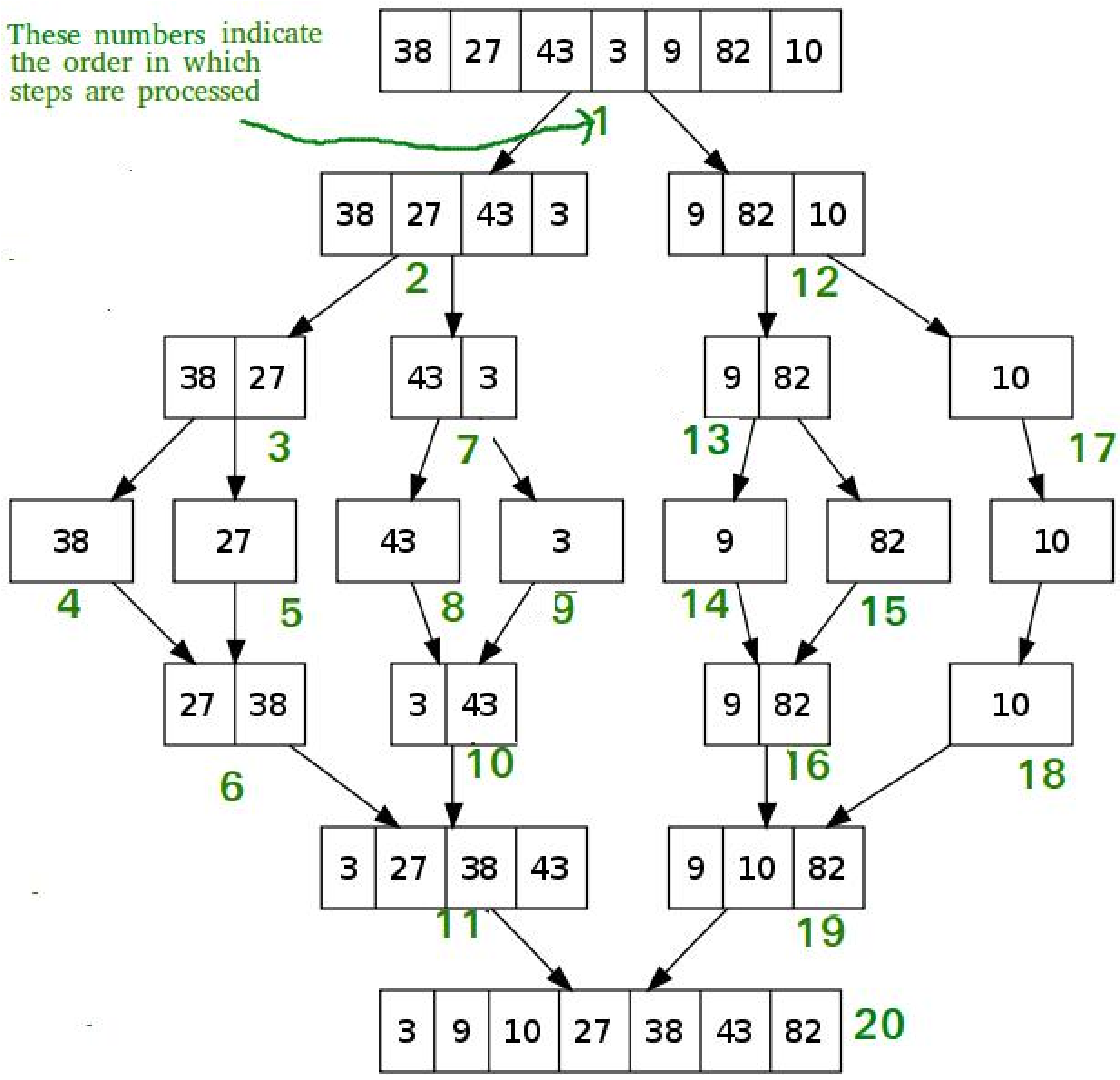
Experiment No.	11.
Type of problem	Merge sort
Students are required to implement/execute/draw/make document	1. Algorithm 2. Program 3. Output 4. Final document with evaluator signature.
List of experiments	1.Write a program on Merge sort 2.Write a program to merge two sorted array 3.Write a program to use a recursive function call 4.Write a program to display step by step merge sort

Merge Sort

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in 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 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 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 comes into action and starts merging arrays back till the complete array is merged.



Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.
T(n) = 2T(n/2) +
The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is .
Time complexity of Merge Sort is in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.
Auxiliary Space: O(n)
Algorithmic Paradigm: Divide and Conquer
Sorting In Place: No in a typical implementation
Stable: Yes
Applications of Merge Sort
Merge Sort is useful for sorting linked lists in O(nLogn) time. In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra

space for linked lists.
In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.
Inversion Count Problem
Used in External Sorting

Write a program on Merge sort
```
/* C program for Merge Sort */
#include<stdlib.h>
#include<stdio.h>

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
   int i, j, k;
   int n1 = m - l + 1;
   int n2 =  r - m;

   /* create temp arrays */
   int L[n1], R[n2];

   /* Copy data to temp arrays L[] and R[] */
   for (i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[m + 1+ j];

   /* Merge the temp arrays back into arr[l..r]*/
   i = 0; // Initial index of first subarray
   j = 0; // Initial index of second subarray
   k = l; // Initial index of merged subarray
   while (i < n1 && j < n2)
   {
      if (L[i] <= R[j])
      {
         arr[k] = L[i];
         i++;
      }
      else
      {
         arr[k] = R[j];
         j++;
```

```
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
       are any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
       are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

/* l is for left index and r is right index of the
   sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}

/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}
```

```
/* Driver program to test above functions */
int main()
{
   int arr[] = {12, 11, 13, 5, 6, 7};
   int arr_size = sizeof(arr)/sizeof(arr[0]);

   printf("Given array is \n");
   printArray(arr, arr_size);

   mergeSort(arr, 0, arr_size - 1);

   printf("\nSorted array is \n");
   printArray(arr, arr_size);
   return 0;
}
```

OUTPUT:

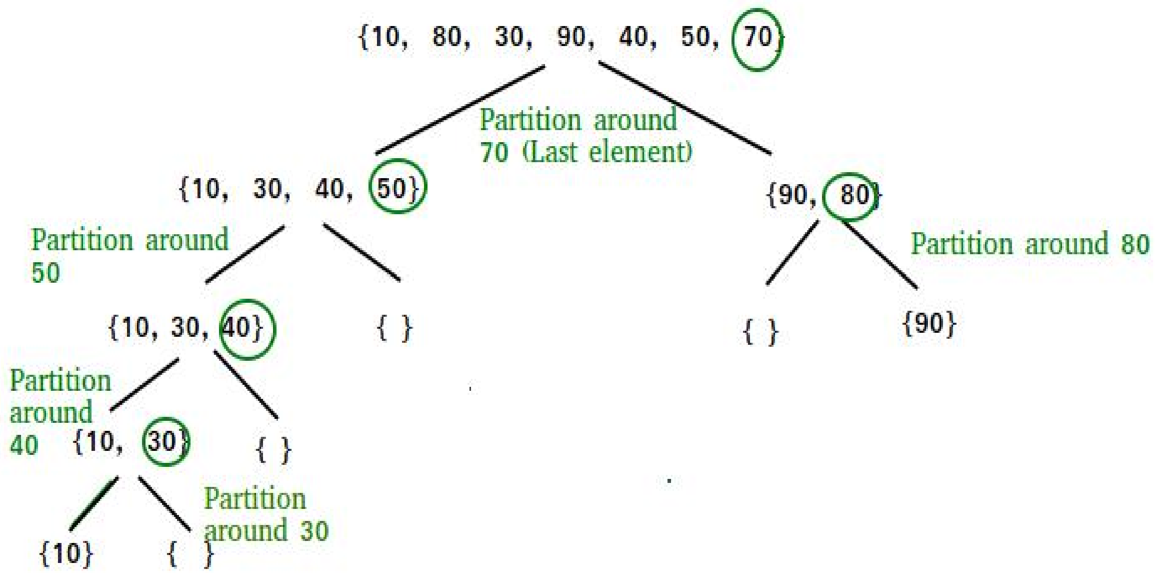
```
Given array is
12 11 13 5 6 7

Sorted array is
5 6 7 11 12 13
```

Experiment No.	12.

Type of problem	Quick sort		
Students are required to implement/execute/draw/make document	1. Algorithm 2. Program 3. Output 4. Final document with evaluator signature.		
List of experiments	1.Write a program on Quick sort 2.Write a program to divide an array into two part 3.Write a program to display step by step quick sort		

QuickSort
Like Merge Sort, 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.
Always pick first element as pivot.
Always pick last element as pivot (implemented below)
Pick a random element as pivot.
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.
Pseudo Code for recursive Quick Sort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[p] is now
         at right place */
      pi = partition(arr, low, high);

      quickSort(arr, low, pi - 1);  // Before pi
      quickSort(arr, pi + 1, high); // After pi
   }
}
```



Partition Algorithm
There can be many ways to do partition; following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()
/* 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 */

```
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element
    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

```
Illustration of partition() :
arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes:  0   1   2   3   4   5   6

low = 0, high =  6, pivot = arr[h] = 70
Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j
                                     // are same

j = 1 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped
```

We come out of loop because j is now equal to high-1. Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot) arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

Analysis of QuickSort
Time taken by QuickSort in general can be written as following.
T(n) = T(k) + T(n-k-1) + (n)

The first two terms are for two recursive calls; the last term is for the partition process. k is the number of elements which are smaller than pivot.
The time taken by Quick Sort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

T(n) = T(0) + T(n-1) + (n)

which is equivalent to

T(n) = T(n-1) + (n)

The solution of above recurrence is (n2).

Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

T(n) = 2T(n/2) + (n)

The solution of above recurrence is (nLogn). It can be solved using case 2 of Master Theorem.

Average Case:

To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn’t look easy.

We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

T(n) = T(n/9) + T(9n/10) + (n)

Solution of above recurrence is also O(nLogn)

Although the worst case time complexity of QuickSort is O(n2) which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

What is 3-Way QuickSort?
In simple QuickSort algorithm, we select an element as pivot, partition the array around pivot and recur for subarrays on left and right of pivot.

Consider an array which has many redundant elements. For example, {1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 2, 2, 4, 1, 4, 4, 4}. If 4 is picked as pivot in Simple QuickSort, we fix only one 4 and recursively process remaining occurrences. In 3 Way QuickSort, an array arr[l..r] is divided in 3 parts:

a) arr[l..i] elements less than pivot.
b) arr[i+1..j-1] elements equal to pivot.
c) arr[j..r] elements greater than pivot.

How to implement QuickSort for Linked Lists?

QuickSort on Singly Linked List
QuickSort on Doubly Linked List
Can we implement QuickSort Iteratively?

Yes, please refer Iterative Quick Sort.

Why Quick Sort is preferred over MergeSort for sorting Arrays

Quick Sort in its general form is an in-place sort (i.e. it doesn’t require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn’t occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

Quick Sort is also tail recursive, therefore tail call optimizations is done.

Why MergeSort is preferred over QuickSort for Linked Lists?

In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we cannot do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

Write a program on 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 or
      // equal to 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:

```
Sorted array:
1 5 7 8 9 10
```

Experiment No.	13.		
Type of problem	Heap sort		
Students are required to implement/execute/draw/make document	1. Algorithm 2. Program 3. Output 4. Final document with evaluator signature.		
List of experiments	1.Write a program on Heap sort 2.Write a program to represent binary tree using array 3.Write a program to create Max-Heap 4.Write a program on Heap sort in descending order		

Heap Sort
Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.
What is Binary Heap?
Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source Wikipedia)
A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.
Why array based representation for Binary Heap?
Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).
Heap Sort Algorithm for sorting in increasing order:
1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.
How to build the heap?
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.
Lets understand with the help of an example:
Input data: 4, 10, 3, 5, 1

```
          4(0)
              /   \
          10(1)   3(2)
      /   \
    5(3)    1(4)
```

The numbers in bracket represent the indices in the array representation of data.

Applying heapify procedure to index 1:

```
                4(0)
                /   \
        10(1)    3(2)
       /   \
        5(3)    1(4)
```

Applying heapify procedure to index 0:

```
             10(0)
                /  \
           5(1)  3(2)
        /   \
      4(3)    1(4)
```

The heapify procedure calls itself recursively to build heap in top down manner.

Heap sort is an in-place algorithm.
Its typical implementation is not stable, but can be made stable (See this)
Time Complexity: Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).
Applications of HeapSort
1. Sort a nearly sorted (or K sorted) array
2. k largest(or smallest) elements in an array
Heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used.

Write a program on Heap sort

```
// C++ program for implementation of Heap Sort
#include <iostream>
using namespace std;

// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
   int largest = i;  // Initialize largest as root
   int l = 2*i + 1;  // left = 2*i + 1
   int r = 2*i + 2;  // right = 2*i + 2

   // If left child is larger than root
   if (l < n && arr[l] > arr[largest])
       largest = l;

   // If right child is larger than largest so far
   if (r < n && arr[r] > arr[largest])
       largest = r;
```

```
    // If largest is not root
    if (largest != i)
    {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// main function to do heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i=n-1; i>=0; i--)
    {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i=0; i<n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

// Driver program
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    heapSort(arr, n);

    cout << "Sorted array is \n";
    printArray(arr, n);
```

OUTPUT:

```
Sorted array is
5 6 7 11 12 1
```