Data Structures (CSC-214)

Lab. Mannual

[Student's Version]

by

Shahid Iqbal Lone
Lecturer

College of Computer

Qassim University
KINGDOM OF SAUDI ARABIA

Table of Contents

LAB. WORK OBJECTIVES:

With a dynamic learn-by-doing focus, this laboratory manual encourages students to explore data structures by implementing them, a process through which students discover how data structures work and how they can be applied. Providing a framework that offers feedback and support, this text challenges students to exercise their creativity in both programming and analysis. Each laboratory unit consists of four parts: the Pre_lab, the Bridge, the In-lab, and the Post_lab, which create an excellent hands-on learning opportunity for students in supervised labs and students engaged in independent study.

This course couples work on program design, analysis, and verification with an introduction to the study of data structures. Data structures capture common ways to store and manipulate data, and they are important in the construction of sophisticated computer programs. Students will be expected to write C++ programs, ranging from very short programs to more elaborate systems. Since one of the goals of this course is to teach how to write large, reliable programs composed from reusable pieces, we will be emphasizing the development of clear, modular programs that are easy to read, debug, verify, analyze, and modify.

1. To learn how the choice of data structures and algorithm design methods impacts the performance of programs.
2. To learn object-oriented design principles.
3. To study specific data structures such as linear lists, stacks, queues, hash tables, binary trees, heaps, trees, binary search trees, and graphs.
4. To study specific algorithm design methods such as the greedy method, divide and conquer, dynamic programming, backtracking, and branch and bound.
5. To gain experience writing programs in C++.

ARRAY

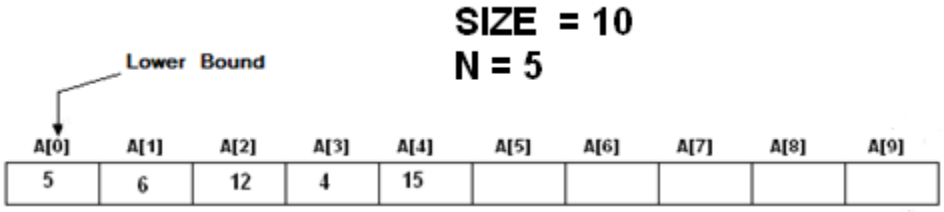
An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier. As discussed in the class/lecture room, there are different algorithms related to an array (Traversing, Insertion, Deletion, Modify, Sorting, Searching (Linear, Binary). Following algorithms helps you to understand and write programs of these algos.

AIM:

Write a program for **Traversing** in Linear Array:

Description: It means processing or visiting each element in the array exactly once for one operation. Let '**A**' is an array stored in the computer's memory and we want to display the contents of '**A**', then it has to be traversed i.e. by accessing and processing each element of '**A**' exactly once.

Linear array '**A**' with lower boundary **LB** (0 in C++) and upper boundary (**UB** = **N - 1** in C++) where **N** is the number of values stored in the array. If the array is empty then **N** will be **equal 0.** **SIZE** is the total locations/positions available in array. This algorithm traverses '**A**' applying process to each element of '**A**'.



Algorithm: (Traverse a Linear Array) Here **A** is an array has **N** elements stored in it, with lower bound **LB=0** and upper bound **UB= N - 1**.

1. K = LB and UB = N - 1
2. Repeat Steps 3 and 4 while K ≤ UB.
3. Apply PROCESS to **A**[K].
4. K = K+1.
   [End of Step 2 loop.]
5. End.

The alternate algorithm using for-loop to traverse **A**:

Algorithm: (Traverse a Linear Array) This algorithm traverse a linear array **A** with lower bound **LB=0** and upper bound **UB= N - 1**.

1. Repeat for K = LB to UB
   Apply PROCESS to **A**[K].
   [End of loop].
2. End.

AIM: Write a program to **Insert** an Item into Linear Array:

Description: Inserting refers to the addition of a new element in an array, say '**A**'. Inserting an element at the end of the array can be easily done. On the other hand, if an element is to be inserted at the beginning or in the middle of array '**A**', then the elements (from insertions point inclusively) must be moved downward to new locations, to accommodate the new element. Assume, values inside the '**A**' are in sorted order and it is needed to insert the new item at such location in '**A**' so that the sorted order of '**A**' should not be disturbed. See the algorithm:

The following set of algorithms inserts a data element **ITEM** at **Kth** position in a linear array.

Here **A** is a Linear Array with **SIZE** locations, **N** is the number of total values stored in **A** and **K** is a positive integer (location where to insert) such that **K<= N**. This algorithm inserts an element **ITEM** into the **Kth** position in **A**.

Algorithm:

1. if N = SIZE then write: "Overflow, Array is Full. " and End
2. Get **ITEM** from user to be inserted in array
   [Find location in sorted array]
3. Call Find_Location_To_Insert (A, N, K, ITEM)
   [Above algorithm finds the location **K** where to insert]
4. Call INSERT (A, N, K, ITEM)
5. End

Algorithm: Find_Location_To_Insert(A, N, K, ITEM)

1. LB = 0 and UB = N – 1
2. Repeat Step-3 for K = LB to UB.
3. If **A**[K] > **ITEM** then return K. [Insertion at front or in middle]
   [End of Step 2 loop.]
4. return K. [Insertion at the end of array]
5. End.

Algorithm: INSERT (A, N, K, ITEM)

1. Set J = N - 1. [Initialize counter.]
2. Repeat Step-3 and 4 while J≥K.
3. Set A[J+1] = A[J]. [Move Jth element downward.]
4. Set J = J-1. [Decrease counter.]
   [End of Step 2 loop.]
5. Set A[K] = ITEM. [Insert element.]
6. Set N = N+1. [Reset N.]
7. End.

AIM: Write a program to **Delete** an Item from Linear Array:

Description: Deletion refers to the operation of removing one of the element from the array '**A**'. Deleting an item at the end of the array presents no difficulties. But, deleting an item from beginning or somewhere in the middle of array would require that each subsequent element be moved one location upward in order to fill up the array.

The following algorithm deletes the K^{th} element from a linear array **A** with **N** values stored in it.

Here **A** is a Linear Array with **SIZE** locations, **N** is the number of total values stored in **A** and **K** is a positive integer such that **K<=N**. This algorithm deletes the K^{th} element from **A**.

Algorithm:

1. If **N = 0** then write: "Underflow. Array is Empty. " and End
2. Get **ITEM** from user to be deleted from array
   [Find location in sorted array]
3. Call Find_Location_To_Delete(A, N, K, ITEM)
4. If found Call DELETE (A, N, K, ITEM) [if K = -1 then not found]
   Else write: "Not Found in array"
5. End

Algorithm: Find_Location_To_Delete(A, N, K, ITEM)

1. Repeat Step 2 for I = LB to UB.
2. If **A**[I] = **ITEM** then K = I and return K. [Deletion from front or from middle]
   [End of Step 2 loop.]
3. K = -1 and return K. [**ITEM** Not found in array]
4. End.

Algorithm: DELETE (A, N, K, ITEM)

1. Set ITEM = A[K].
2. Repeat for J=K to N-1:
   Set A[J] = A[J+1] [Move J + 1 element upward.].
   [End of loop.]
3. Set N = N-1. [Reset the number "**N**" of element in **A**.]
4. End.

Source code:

```
#include<iostream.h>
#include<process.h>
int const SIZE=10;

 int A[SIZE], N=0;  //   global variables,
                    // no need to use them as arguments/parameters

// Function  definition before main( ) function

bool IsFull( )
{ if (N==SIZE) return true;      else return false;}

bool IsEmpty()
{   if (N==0) return true;     else return false; }

int  Find_Location_To_Insert(int ITEM )
{









}

void Insert( int K, int ITEM)
{




}

void Traverse()
{


}

int Find_Location_To_Delete(int ITEM )
{









}
```

```
int  Delete( int K, int ITEM )




}


int main()
{
        int ch, ITEM, K;
        while(1)
        {cout<<"\n\n\n\n\n";
        cout<<"\t1- insert value\n";
        cout<<"\t2- delete value\n";
        cout<<"\t3- traverse array\n";
        cout<<"\t4- exit\n\n";
        cout<<"\n\t\tyour choice :";
        cin>>ch;
        switch(ch)
        {
        case 1: if(IsFull( )) {cout<<"\n\nArray is full\n\n";  break;}
                cout<<"\n\n For Insertion,   Put a value : ";
                cin>>ITEM;
                K=Find_Location_To_Insert(ITEM);
                Insert(K, ITEM );
                break;

        case 2: if(IsEmpty( )) {cout<<"\n\nArray is Empty\n\n";  break;}
                cout<<"\n\n For Deletion,   Put a value : ";
                cin>>ITEM;
                K= Find_Location_To_Delete(ITEM);
                if(K == -1 ) cout<<"\n\n"<<ITEM<<" Not Found in array \n";
                else cout<<"\n\n "<<Delete(K, ITEM )<< "  deleted from array\n ";
                break;

        case 3: if(IsEmpty( )) {cout<<"\n\nArray is Empty\n\n";  break;}
                Traverse();
                break;

        case 4: exit(0);

        default: cout<<"\n\nInvalid choice\n";
        }// end of switch
        }// end of while loop
        return 0;
}// end of main( )
```

AIM: Write a program to **Sort** values in Ascending / Increasing Order using **Bubble Sort** Technique in Linear Array:

Description: Sorting refers to the operation of re-arrangements of values in Ascending or Descending order of the Linear Array '**A**'. Here '**A**' is Linear Array and **N** is the number of values stored in **A.**

Algorithm: (Bubble Sort) BUBBLE (A, N)
Here ***A*** is an Array with ***N*** elements stored in it. This algorithm sorts the elements in ***A***.

1. Repeat Steps 2 to 4 for Pass = 1 to N – 1
2. Set Swapped = 0 and K = 0
3. Repeat while K < (N – Pass)
   (a) if ***A***[K] > ***A***[K + 1] then
   Interchange ***A***[K] and ***A***[K + 1] and
   Set Swapped = 1
   [End of if structure.]
   (b) Set K = K + 1
   [End of inner loop of step-3.]
4. if Swapped = 0 then break the outer loop of step-1
   [End of outer loop of step-1.]
5. End

Source code:

```
/* This program sorts the array elements in the ascending order using bubble
sort  method */

#include <iostream.h>
int const  N = 6

void BubbleSort(int [ ], int); // function prototyping

int main()
{
int A[N]={77,42,35,12,101,6}; //You can also get values from user for the array.
int i;
cout<< "The elements of the array before sorting\n";
for (i=0; i< N ; i++) cout<< A[i]<<",  ";
BubbleSort(A, N);
cout<< "\n\nThe elements of the array after sorting\n";
for (i=0; i< N; i++) cout<<A[i]<< ",  ";
return 0;
}
```

```
void BubbleSort(int A[N],  int N)
{




}
```

AIM: Write a program to do **Linear Search** using a given **Search Key** in an un-sorted Linear Array '**A**' which has **N** values in it.

Description:

The linear search compares each element of the array with the **search key** until the **search key** is found. To determine that a value is not in the array, the program must compare the **search key** to every element in the array '**A**'. It is also called "***Sequential Search***" because it traverses the data sequentially to locate the element.

Algorithm: (Linear Search)
LINEAR_SEARCH (A, N, SKEY)
Here **A** is a Linear Array with **N** elements and ***SKEY*** is a given item of information to search. This algorithm finds the location of ***SKEY*** in **A** and if successful, it returns its location otherwise it returns -1 for unsuccessful.

1. Repeat step-2 for K = 0 to N-1
2. if(**A** [K] = SKEY) return K [Successful Search]
   [End of loop of step-1]
3. return -1 [Un-Successful]
4. End.

Source code:

```
/* This program use linear search in an array to find the LOCATION of the given
Key value */

#include <iostream.h>
int const N=10;
int LinearSearch(int [N ], int);   // Function Prototyping
int main()
 {  int A[N]= {9, 4, 5, 1, 7, 78, 22, 15, 96, 45}, Skey, LOC;
    // you can modify code to get values from user
    cout<<"\n  Enter the Search Key: ";
    cin>>Skey;
    LOC = LinearSearch( A, Skey);  // call a function
    if(LOC == -1)
       cout<<" \n The search key is not in the array\n Un-Successful Search\n";
    else
     cout<<"\n The search key  "<<Skey<< "  exists at location "<<LOC<<endl;
    return 0;
}

int LinearSearch (int A[N], int skey)  // function definition
{




}
```

AIM: Write a program to do **Binary Search** using a given **Search Key** in a **Sorted** Linear Array '**A**' which has **N** values in it.

Description: The binary search algorithm can only be used with ***sorted array*** and eliminates one half of the elements in the array being searched after each comparison. The binary search algorithm applied to our array **A** works as follows:

Algorithm: (Binary Search)
Here **A** is a sorted Linear Array with **N** elements stored in it in sorted order and **SKEY** is a given item of information to search. This algorithm finds the location of SKEY in **A** and if successful, it returns its location otherwise it returns -1 for unsuccessful.

```
BinarySearch (A, N, SKEY)
    [Initialize segment variables.]
1.  Set START=0, END=N-1 and MID=int((START+END)/2).
2.  Repeat Steps 3 and 4 while START ≤ END and A[MID]≠SKEY.
3.      If SKEY< A[MID] Then
                Set END=MID-1.
        Else    Set START=MID+1.
        [End of If Structure.]
4.      Set MID=int((START +END)/2).
    [End of Step 2 loop.]
5.  If  A[MID]= SKEY then  Set LOC= MID
    Else
        Set LOC = -1
    [End of IF structure.]
6.  return LOC and End
```

Source code:

```
// C++ Code  for Binary Search
#include <iostream.h>
int const N=10;
int BinarySearch(int [ ], int);  //  Function Prototyping
int main()
{       int A[N]= {3, 5, 9, 11, 15, 17, 22, 25, 37, 68}, SKEY, LOC;
       // You can modify  the code to get values from user

        cout<<"\n Enter the Search Key:  ";
        cin>>SKEY;
        LOC = BinarySearch(A, SKEY); // Function call
        if(LOC == -1)
             cout<<"\n The search key is not found in the array\n";
        else
        cout<<"\n The search key  "<<SKEY<< "  exist at location  "<<LOC<<endl;
return 0;
}

int BinarySearch (int A[N], int skey)  // function definition
{









}
```

AIM: Multiplication of two Matrices

Description: Here **A** is a two – dimensional array with **M** rows and **N** columns. **B** is also a two – dimensional array with **X** rows and **Y** columns. This algorithm multiplies these two arrays. The output of the process will be matrix **P** with **M** rows and **Y** columns.

Algorithm: (Multiplication of two Matrices)

1. Get dimensions of Matrix- A from user i.e. **M** x **N**
[Read values for Matrix- A]
2. Repeat for I = 0 to M-1
3. Repeat for J = 0 to N-1
4. Read A[I][J] [Get value from user]
[End of Step-3 for Loop]
[End of Step-2 for Loop]

5. Get dimensions of Matrix- B from user i.e. **X** x **Y**
[Read values for Matrix- B]
6. Repeat for I = 0 to X-1
7. Repeat for J = 0 to Y-1
8. Read B[I][J] [Get value from user]
[End of Step-7 for Loop]
[End of Step-6 for Loop]

9. If (N ≠ X) then
Write: "Multiplication is not possible, incompatible dimensions". **and** exit

10. Repeat for I = 0 to M-1
11. Repeat for J = 0 to Y-1
12. Set P[I][J] = 0
13. Repeat For K = 0 to X-1
14. Set P[I][J] = P[I][J] + A[I][K] * B[K][J]
[End of Step-13 for Loop]
[End of Step-11 for Loop]
[End of Step-10 for Loop]
[End of If of step-9]

15. Print the Matrix-P
16. End

Source code:

```
#include<iostream.h>
#include<iomanip.h>
int main()
{       int M,N,X,Y,I,J,K;
        int  A[20][20], B[20][20], P[20][20];

        cout<<"\n\n  *** First Matrix ***"<<endl;
        cout<<"Number of Rows :  ";
        cin>>M;
        cout<<"Number of Columns :  ";
        cin>>N;
        cout<<"\n Enter The First Matrix"<<endl;

        for (I=0 ; I<M ; I++ )
                for (J=0 ; J<N ; J++) cin>>A[I][J];

        cout<<"\n\n\n *** Second  Matrix ***"<<endl;
        cout<<"Number of Rows :  ";
        cin>>X;
        cout<<"Number of Columns :  ";
        cin>>Y;
        cout<<"\n \n Enter The Second Matrix"<<endl;

        for (I=0 ; I<X ; I++ )
                for (J=0 ; J<Y ; J++) cin>>B[I][J];

        cout<<"\n\n The First Matrix You Entered"<<endl;
        for (I=0 ; I<M ; I++)
        {
                for (J=0 ; J<N ; J++) cout<<A[I][J]<<"\t ";
            cout<<"\n";
        }
        cout<<"\n\n The Second Matrix You Entered"<<endl;
        for (I=0 ; I<X ; I++)
        {
                for (J=0 ; J<Y ; J++) cout<<B[I][J]<<"\t ";
            cout<<endl"\n";
        }
        // (Column of first matrix-A  must equal to Row of second matrix-B)
        // The Dimensions of Product matrix-P   will be     M  x  Y
        if( N == X ) {
```

```
        }
        else   cout<<"\n dimensions of matrics are not valid for  product \n";


   cout<<"\n\n\n";
        return 0;
}
```

Exercise:

1- Write a Program to create a dynamic array has N elements. Get integer values into array and then provide the following functionality:

- **a:** sort them in ascending/descending order depending on request, using bubble sort method.
- **b:** Print all even numbers available in the array.
- **c:** Print all Odd numbers available in the array.
- **d:** Print all prime numbers available in the array.
- **e:** Print all complete square numbers (e.g. 4, 9, 16, 25 etc.) available in the array.

2- Write a Program which takes two matrices (i.e. Matrix_A and Matrix_B) of M x N dimensions and perform the following functionality:

- **a:** Sum of Matrix-A and B.
- **b:** Difference of Matrix-A and B.

S T A C K

This Lab. Work introduces the stack data type and the **Last In, First Out** data access that it supports. Contiguous (array) and linked structures are used for implementations. LIFO is short for Last in, First out. That is, the last element being added to the array will be the first to be removed.

Here are the minimal operations we'd need for an abstract stack (and their typical names):

- `Push( )`: Places a value/object on the **Top** of the stack.
- `Pop( )`: Removes a value/object from the **Top** of the stack and produces that value/object.
- `IsEmpty( )`: Reports whether the stack is empty or not.
- `IsFull( )`: Reports whether the stack is Full or not (for array implementation).
- `Peak( )`: produces Top value/object of the stack without removing it.
- `Traverse( )`: visit all elements from Top to Bottom without removing them.

Aim: Write a program to create a Stack and different functionality related to it as stated above and implement it using one dimensional Array.

Description: Here **STACK** is an array with **STACKSIZE** locations. **TOP** points to the top most element and **ITEM** is the value to be inserted.

Algorithm for PUSH using array:

Algorithm: PUSH( )
[STACK already filled?]
1. If TOP=STACKSIZE-1, then: Print: OVERFLOW / Stack Full, and End.
2. Set TOP =TOP+1. [Increase TOP by 1.]
3. Set STACK[TOP]=ITEM. [Insert ITEM in new TOP position.]
4. End.

Algorithm for POP using array:

Description: Here **STACK** is an array with **STACKSIZE** locations. **TOP** points to the top most element. This procedure deletes the top element of STACK and assigns it to the variable **ITEM**.

Algorithm: POP( )
[Does STACK has an item to be removed? Check for empty stack]
1. If TOP=-1, then Print: UNDERFLOW/ Stack is empty, and End.
2. Set ITEM = STACK[TOP]. [Assign TOP element to ITEM.]
3. Set TOP=TOP-1. [Decrease TOP by 1.]
4. End.

Algorithm for PEAK using array:

Description: Here **STACK** is an array with **STACKSIZE** locations. **TOP** points to the top most element. This procedure produces the top element of STACK and assigns it to the variable **ITEM** without removing it.

Algorithm: PEAK( )
[Does STACK has an item to Peak? Check for empty stack]
1. If TOP=-1, then Print: UNDERFLOW/ Stack is empty, and End.
2. Set ITEM = STACK[TOP]. [Assign TOP element to ITEM.]
3. End.

Algorithm for Traverse STACK implemented as array:

Description: Here **STACK** is an array with **STACKSIZE** locations. **TOP** points to the top most element. This procedure produces all the values (pushed on stack) without removing them.

Algorithm: Traverse( )
[Does STACK has any item to Traverse? Check for empty stack]
1. If TOP=-1, then Print: UNDERFLOW/ Stack is empty, and End.
2. T = TOP [Put TOP into T so that its value not to be changed]
3. Repeat While T ≥ 0
   a) Write: STACK[T]
   b) Set T = T – 1
   [End of Loop of step-3.]
4. End.

Source code:

```
// A Program that exercise the operations on Stack Implementing Array
// i.e. (Push, Pop, Peak and Traverse)
#include <conio.h>
#include <iostream.h>
#include <process.h>

int const STACKSIZE = 20;

//  global  variable and array  declaration
int TOP=-1;   // Initially  Stack is empty
int Stack[STACKSIZE];  // Array for Stack

void Push(int);    //   functions prototyping
int  POP( );
int  Peak( );
bool IsEmpty( );
bool IsFull( );
void Traverse( );
```

```
int main( )
{ int item,  choice;
  while( 1 )
   {
    cout<< "\n\n\n\n\n";
    cout<< "    *******  STACK  OPERATIONS   ********* \n\n";
    cout<< "  1-  Push item \n  2-  Pop Item \n";
    cout<< "  3- Peak (Top Item) \n  4-  Traverse / Display  Stack Items \n";
    cout<< "  5-  Exit.";
    cout<< " \n\n\t  Your choice ---> ";
    cin>> choice;
    switch(choice)
       { case 1: if(IsFull())cout<< "\n Stack Full/Overflow\n";
                 else
                  { cout<< "\n Enter a number: ";    cin>>item;
                   Push(item); }
                 break;

        case 2: if(IsEmpty())cout<< "\n Stack is empty) \n";
                          else
                {item=POP();
                         cout<< "\n deleted from Stack = "<<item<<endl;}
                  break;

        case 3: if(IsEmpty())cout<< "\n Stack is empty) \n";
                          else
                {item=Peak();
                         cout<< "\n Peak of Stack (Top) = "<<item<<endl;}
                  break;

         case 4: if(IsEmpty())cout<< "\n Stack is empty) \n";
                 else
                  { cout<< "\n List of Item pushed on Stack:\n";
                     Traverse();
                  }
                  break;

        case 5:  exit(0);
        default:
                  cout<< "\n\n\t Invalid Choice: \n";
       } // end of  switch block

    }  // end of while loop

 }  // end of of main() function

void Push(int item)
{



}
```

```
int POP( )
{


}

int Peak( )
{


}

bool IsEmpty( )
{                                                    }

bool IsFull( )
{                                                              }

void Traverse( )
 {


 }
```

Implementing a stack with linked Structure:

Recall lecture in which we have used pointers with structures and we also have learned that how to access the members of the structure using pointers. Using a "linked structure" is one way to implement a stack so that it can handle *essentially* any number of elements. Technically there is no need to check "Overflow" of stack. This approach is used when the numbers of elements which are to be pushed / popped on/from stack are not determined.

Aim: Write a program to create a Stack and different functionality related to it (i.e. Push( ), Pop( ), Peak( ), Traverse( )). Implement it using **linked Structure**.

Description: Here **NODE** is a structure has two members (**Info** and **Next**), where **Info** is used to store the data and **Next** is a pointer used to hold the address of next/lower node. **TOP** is pointer which points to the top most element of the stack. The initial value of the **TOP** is **NULL. ITEM** is the value to be inserted. **NewNode** is pointer variable which holds the address of newly created node.

Format of struct :

```
struct NODE
{  int  Info;
   struct NODE *Next;
};
```

Algorithm to PUSH item on Stack using Linked structure:

Algorithm: PUSH(**ITEM**), [Here the initial value of **TOP** is **NULL**]

1. Create dynamic **NODE** to store (**Info** and **Next**) for top node and store its address into **NewNode** pointer
   [Insert ITEM in new TOP position.]
2. Set NewNode **->** Info = ITEM and NewNode **->** Next = TOP
3. Set TOP = NewNode [Adjust new TOP address]
4. End.

Algorithm to POP item from Stack using Linked structure:

Algorithm: POP( )
This procedure deletes the **TOP** element of **STACK** and assigns its value to the variable **ITEM**.

1. [Check for empty stack]
   If TOP = NULL, then Print: UNDERFLOW/ Stack is empty, and End.
2. Set ITEM = TOP **->** Info [Assign TOP element to ITEM.]
3. Set T = TOP [Load the address of current TOP into T]
4. TOP = TOP **->** Next. [Reset TOP by shifting it to lower/next node.]
5. Release T [Send back the memory occupied by deleted node.]
6. Return ITEM and End

Algorithm to produce PEAK item from Stack using Linked structure:

Algorithm: PEAK( )
This procedure Produce the **TOP** element of **STACK** without deleting it and assigns its value to the variable **ITEM**.

1. [Check for empty stack]
   If TOP = NULL, then Print: UNDERFLOW/ Stack is empty, and End.
2. Set ITEM = TOP **->** Info [Assign TOP element to ITEM.]
3. Return ITEM and End

Algorithm to TRAVERSE Stack using Linked structure:

Algorithm: TRAVERSE ( )
This function produces all elements of the **STACK** without deleting them.

1. [Check for empty stack]
   If TOP = NULL, then Print: UNDERFLOW/ Stack is empty, and End.
2. Set T = TOP [Load the address of current TOP into T]
3. Repeat while (T not NULL)
   a) Print: T **->** Info
   b) T = T **->** Next.
   [End of Loop of step-3.]
4. End

Source code:

```
// A Program that exercise the operations on Stack
// Implementing POINTERS (Linked Structures) (Dynamic Binding)
// This program provides you the concepts that how STACK is
//  implemented using Pointer/Linked Structures

#include <iostream.h>
#include <process.h>

struct NODE {
                int info;
                struct NODE *next;
            };

struct NODE *TOP = NULL;

void PUSH (int item)
{









}

int POP ()




 }


int PEAK ()
{                                                    }


void TRAVERSE ()
{


}

bool IsEmpty()
{                                                          }
```

```
int main ()
{ struct NODE *T;

  int item, choice;
  cout<< "\n\n\n\n\n";
    cout<< "    ******* STACK OPERATIONS ********* \n\n";
    cout<< "  1- Push item \n  2- Pop Item \n";
    cout<< "  3- Peak (Top Item) \n  4- Traverse / Display Stack Items \n";
    cout<< "  5- Exit.";
    cout<< " \n\n\t Your choice ---> ";
    cin>> choice;
        switch(choice)
        { case 1:
                 cout<<"\nPut a value: ";
                        cin>>item;
                        PUSH(item);
                        break;
                case 2:
                          if(IsEmpty()) {cout<<"\n\n Stack is Empty\n";
                                        break;
                                       }
                         item = POP();
                         cout<< item <<"\n\n has been deleted \n";
                         break;

                case 3:
                          if(IsEmpty()) {cout<<"\n\n Stack is Empty\n";
                                        break;
                                       }
                         cout<< PEAK () <<"\n\n is on the Peak \n";
                         break;

                case 4:
                         if(IsEmpty()) {cout<<"\n\n Stack is Empty\n";
                               break;
                                       }
                               TRAVERSE();
                               break;
                case 5:
                               exit(0);
        } // end of switch block
  }   // end of loop
  return 0;
} // end of main function
```

An Application of Stack

Transforming Infix Expression into Postfix Expression:

The following algorithm transforms the infix expression **Q** into its equivalent postfix expression **P.** It uses a stack to temporary hold the operators and left parenthesis. The postfix expression will be constructed from left to right using operands from **Q** and operators popped from STACK.

Algorithm: Infix_to_PostFix(Q, P)

Suppose **Q** is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression **P**.

1. Push "(" onto STACK, and add ")" to the end of **Q**.
2. Scan Q from left to right and repeat Steps **3** to **6** for each element of **Q** until the STACK is empty:
3. If an operand is encountered, add it to **P**.
4. If a left parenthesis is encountered, Push it onto STACK.
5. If an operator © is encountered, then:
   a) Repeatedly Pop from STACK and add to **P** each operator (on the top of STACK) which has the same or higher precedence/priority than ©
   b) Push © to STACK.

   [End of If structure.]
6. If a right parenthesis is encountered, then:
   a) Repeatedly Pop from STACK and add to **P** each operator (on the top of STACK) until a left parenthesis is encountered.
   b) Pop the left parenthesis. [Do not add the left parenthesis to P.]
   [End of If structure.]

   [End of Step 2 loop.]
7. End.

Source code:

```
#include <conio.h>
#include <iostream.h>
#include <process.h>
#include <string.h>

// global declation
int const null = -1;
int const SIZE = 100;

   int Top = null; //  Top of Stack
   char Q[SIZE], P[SIZE], Stack[SIZE]; // Q is infix and P is postfix expression array
   int n = 0;    // used to count item inserted in P and index for P

void  get_Infix ( )
       {
              cout<<"Put an arithematic INFIX _Expression\n\n\t\t";
              cin.getline(Q,99); // reads an infix expression into Q as string
       }
```

```
// following function will do initial work with Q and stack
void   step_One( )
        {                       // This function add ) at the and of Q
                               // This statement will push first '(' on Stack
        }

void convert()
        {
```

```
} // closing of the convert function

void display( )
{
        P[n]='\0'; // this statement will put string terminator at the
                   // end of P which is Postfix expression
        cout<<"\n\nPOSTFIX EXPRESION IS \n\n\t\t"<<P<<endl;
} // closing of the convert function


void Push(char item)
 {


 }

char Pop( )
 {                                                    }

bool IsEmpty( ) {                                                          }


int main()
{
        get_Infix( );
        step_One( );
        convert( );
        display( );
        return 0;

} //END OF MAIN FUNCTION
```

Q U E U E

A queue is a linear list of elements in which deletion can take place only at one end, called ***front***, and insertions can take place only at the other end, called ***rear***. It is a ***first-in-first-out (FIFO)*** list. Since the first element in a queue will be the first element out of the queue. There are main two ways to implement a queue:

1. Circular queue using ***array***
2. Linked Structures using (***Pointers***)

Primary queue operations:

Enqueue: insert an element at the rear/end of the queue.

Dequeue: retrieves the item at the front of the queue, and removes/deletes it from the queue.

Aim: Write a program to create a CIRCULAR QUEUE and different functionality (Enqueue, Dequeue and Teraverse) related to it by implement it using one dimensional Array.

Description: Here **QUEUE** is an array with **QSIZE** locations. **FRONT** points to the beginning of array, where deletion will be performed and **REAR** points to the element where after insertion will be performed. **ITEM** is the value to be inserted.

Algorithm: ENQUEUE / Insertion in CIRCULAR QUEUE using array:

Algorithm-1: ENQUEUE(QUEUE, QSIZE, FRONT, REAR, ITEM)

1. [QUEUE already filled?]
   If FRONT = 0 and REAR = QSIZE - 1 or FRONT = REAR + 1
   then Write: OVERFLOW, and End.

2. [Find new value of REAR, where ITEM is to be Inserted.]
   If REAR = -1, then [Queue initially empty.]
   Set FRONT= 0 and REAR = 0
   Else if REAR = QSIZE - 1, then
   Set REAR = 0
   Else
   Set REAR = REAR+1.
   [End of If Structure.]

3. Set QUEUE[REAR] = ITEM. [This insert new element.]

4. End.

Algorithm: DEQUEUE / Deletion from CIRCULAR QUEUE using array:

Algorithm: DEQUEUE(QUEUE, QSIZE, FRONT, REAR, ITEM)
Description: Here **QUEUE** is an array with **QSIZE** locations. **FRONT** and **REAR** points to the front and rear of the **QUEUE** respectively. This procedure deletes an element from **FRONT** and assigns it to the variable **ITEM.**

[QUEUE already empty?]
1. If FRONT = -1, then: Write UNDERFLOW, and End.
2. Set ITEM = QUEUE[FRONT]. [Assign FRONT element to ITEM.]
[Find new value of FRONT.]
3. If FRONT = REAR, then: [There was one element and has been deleted]
Set FRONT= -1, and REAR = -1.
Else if FRONT= QSIZE - 1, then: [Circular, so set Front = 0]
Set FRONT = 0
Else
Set FRONT = FRONT+1.
[End of If structure.]
4. Return ITEM
5. End

Algorithm: TRAVERSE CIRCULAR QUEUE using array:

Algorithm: TRAVERSE(QUEUE, QSIZE, FRONT, REAR, ITEM)
Description: Here **QUEUE** is an array with **QSIZE** locations. **FRONT** and **REAR** points to the front and rear of the **QUEUE** respectively. This procedure Traverse all elements from **FRONT** to **REAR** and print them on screen.

[QUEUE already empty?]
1. If FRONT = -1, then: Write UNDERFLOW, and End.
2. Set F = FRONT [Put FRONT into F. FRONT should not be changed.]
3. Repeat step-4 and 5 While(F ≠ -1)
4. Print: QUEUE[F] [Prints one value of QUEUE.]
5. If F = QSIZE - 1 then: Set F = 0 [Circular, so set F = 0]
Else if F = REAR then: Set F = -1 [Traversing finished]
Else set F = F + 1
[End if Structure]
6. End

Source code:

```
// c-language code to implement Circular QUEUE using array
#include<iostream.h>
#include <process.h>

int const  QSIZE = 10;
//   Global  declarations of variables available to every function
        int QUEUE[QSIZE];
        int FRONT = -1;
        int REAR  = -1;

bool IsEmpty(){                                                    }

bool IsFull()   {

                }
void Enqueue(int ITEM)
{









}

int Dequeue()
{









}

void Traverse()
{









}
```

```
int main()
{
   int choice,ITEM;
         while(1)
               {
                 cout<<"\n\n\n\n QUEUE operation\n\n";
                 cout<<"1-insert value \n 2-deleted value\n";
                 cout<<"3-Traverse QUEUE \n 4-exit\n\n";
                 cout<<"\t\t your choice:";
                 cin>>choice;

               switch(choice)
               {
                     case 1:
                                 cout"\n put a value:";
                                 cin>>ITEM);
                                 Enqueue(ITEM);break;
                     case 2:
                           ITEM=Dequeue();
                           if(ITEM!=-1)cout<<t<<  " deleted \n";
                           break;
                     case 3:
                           cout<<"\n queue state\n";
                           Traverse(); break;

                     case 4:exit(0);
               }  // end of switch
        }  //  end of while loop
               return 0;
        }// end of main( ) function
```

Implementing a QUEUE with linked Structure:

Reference to the lecture in which we have used pointers with structures and we also have learned that how to access the members of the structure using pointers. Using a "linked structure" is one way to implement a QUEUE so that it can handle *essentially* any number of elements.

Aim: Write a program to create a QUEUE and different functionality related to it (i.e. Enqueue( ), Dequeue( ), Traverse( )). Implement it using **linked Structure**.

Description: Here **QUEUE** is a structure has two members (**Info** and **Next**), where **Info** is used to store the data and **Next** is a pointer used to hold the address of next/right node. **FRONT** is a pointer to QUEUE, points to the beginning/first element of the array, where deletion will be performed. **REAR** is a pointer to QUEUE, points to the Last element where after insertion will be performed. **ITEM** is the value need to be inserted. The initial value of the **FRONT** and **REAR** are **NULL.** **NewNode** is pointer variable which holds the address of newly created QUEUE node.

Format of struct :

```
struct QUEUE
{   int   Info;
    struct NODE *Next;
};
```

Algorithm to ENQUEUE item on QUEUE using Linked structure:

Algorithm: ENQUEUE(**ITEM**), [Here the initial value of **FRONT & REAR** are **NULL**]

1. Create dynamic QUEUE **NODE** to store (**Info** and **Next**) for next node and store its address into **NewNode** pointer
   [Insert ITEM in newly created Node for QUEUE.]
2. Set NewNode **->** Info = ITEM and NewNode **->** Next = NULL
3. If REAR = NULL then:
   Set REAR and FRONT both = NewNode [First Node Enqueued]
   Else Set REAR **->** Next = NewNode and Set REAR = NewNode
   [End of If Structure]
4. End.

Algorithm to DEQUEUE an item from QUEUE using Linked structure:

Algorithm: DEQUEUE( )
This procedure deletes the **FRONT** element of **QUEUE** and assigns its value to the variable **ITEM**.
[QUEUE already empty?]

1. If FRONT = NULL, then: Write UNDERFLOW / Empty QUEUE, and End.
2. ITEM = FRONT **->** Info. [Assign FRONT element to ITEM.]
3. If FRONT = REAR then: [Only one node is there in the QUEUE]
   Set REAR and FRONT both = NULL [Node deleted & QUEUE is Empty]
   Else Set FRONT = FRONT **->** Next. [Set Front to point Next Node]

   [End of If Structure]
4. Return ITEM and End.

Algorithm to TRAVERSE QUEUE using Linked structure:

Algorithm: TRAVERSE( )

FRONT and **REAR** are pointers and points to the front and rear of the **QUEUE** respectively. **F** is Pointer to QUEUE & holds the FRONT. This procedure Traverse all elements from **FRONT** to **REAR** and print them on screen.

[QUEUE already empty?]

1. If FRONT = NULL, then: Write UNDERFLOW / Empty QUEUE, and End.
2. Set F = FRONT [Here F is Pointer to QUEUE to hold FRONT.]
3. Repeat While (F != NULL)
   Print: F **->** Info. [Prints one value of QUEUE.]
   F = F **->** Next [Set F to point Next Node]
   [End of loop of Step-3]
4. End.

Source code:

```
// A Program that exercise the operations on QUEUE
// using Linked Structures / POINTER (Dynamic Binding)

#include <conio.h>
#include <iostream.h>

struct QUEUE
 { int Info;
   QUEUE *Next;
 };

QUEUE *REAR=NULL, *FRONT=NULL;

void Enqueue(int);
int  Dequeue(void);
void Traverse(void);

void main(void)
{ int ITEM, choice;
  while( 1 )
    {
      cout<<"    *******  QUEUE UNSING POINTERS  ********* \n";
      cout<<" \n\n\t  ( 1 )  Enqueue  \n\t ( 2 ) Dequeue \n";
          cout<<"\t ( 3 ) Print queue \n\t ( 4 ) Exit.";
      cout<<" \n\n\n\t  Your choice ---> ";
      cin>>choice);
```

```
    switch(choice)
     {
      case 1:   cout<< "\n Enter a number: ";
                cin>>ITEM;
                Enqueue(ITEM);
                break;

      case 2:   ITEM = Dequeue();
                    if(ITEM) cout<<" \n Deleted from Q = "<<ITEM<<endl;
                break;

      case 3:   cout<<" \n \n List  of values on QUEUE \n"<<endl;
                 Traverse();
                break;

      case 4:   exit(0);
                break;

      default:  cout<<"\n\n\t Invalid Choice: \n";
     } // end of switch block

   } // end of while loop

} // end of of main() function

void Enqueue (int ITEM)
{









}

int Dequeue(void)
{




}
```

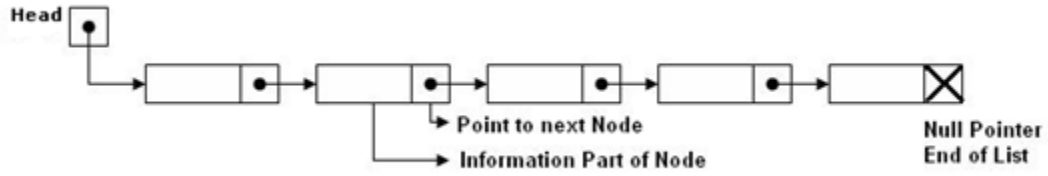
```
void Traverse(void)
{



}
```

LINKED LIST

Linked List:

A linked list or one way list is a linear collection of data elements, called ***nodes***, where the linear order is given by means of ***"pointers"***. Each node is divided into two parts.

- The first part contains the information of the element / node.
- The second part called the link field contains the address of the next node in the list. **Example:**



Operations on Linked List:

a) Traversing Linked List.
b) Searching in Linked List.
c) Insertion in Linked List
d) Deletion from Linked List

Aim: Write a program to create a **Linked List** and different functionality related to it as stated above.

Description: Here **LIST** is a structure has two members (**Info** and **Next**), where **Info** is used to store the data and **Next** is a pointer used to hold the address of next node. **HEAD** is a pointer to **LIST**, points to the beginning/first element of the **LIST. ITEM** is the value needed to insert or delete in/from the **LIST**. The initial value of the **HEAD** is **NULL,** it means that initially **LIST** is empty. **NewNode** is pointer variable which holds the address of newly created **LIST** node.

Format of struct :

```
struct  LIST
{   int   Info;
    struct NODE *Next;
};
```

Traversing the Linked List:

Algorithm: (Traversing a Linked List) Let **LIST** be a linked list in memory. This algorithm traverses **LIST**, applying an operation PROCESS to each Node / element of the **LIST**. **HEAD** points to the first node of the **LIST.** Pointer variable **CURR** point to the node currently being processed.

1. Set CURR = HEAD. [Initializes pointer CURR.]
2. Repeat Steps 3 and 4 while CURR ≠ NULL.
3. Apply PROCESS to CURR -> INFO.
4. Set CURR = CURR -> NEXT [CURR now points to the next node.]
   [End of Step-2 loop.]
5. End.

Inserting a new node in list:

The following algorithm inserts an **ITEM** into **LIST**.

Algorithm: INSERT(ITEM)

NOTE: Here it is needed that all the time, **LIST** should remains in the sorted order using **Info** part of the Node. To Insert / Enqueue a new node in the **LIST**, first **LIST** will be searched (using **Info** of Nodes) to find its LOCATION (where to insert new Node). This exercise makes it possible that every new node has the probability to insert at the beginning, in the middle or at the end of **LIST.**
CURR is a pointer points to the node currently being processed and **PREV** is a pointer pointing to previous node of the current node.

[This algorithm adds NewNodes at any position (Top, in Middle OR at End) in the List]

1. Create a **NewNode** node for **LIST** in memory
2. Set **NewNode ->** Info =ITEM. [Copies new data into INFO of new node.]
3. Set **NewNode ->** Next = NULL. [Copies NULL in NEXT of new node.].

4. If **HEAD** = NULL then: [Condition to create the Linked List.]
   HEAD=**NewNode** and return. [Add first node in list]

[Add on top of existing list]

6. If **NewNode->** Info < **HEAD** ->Info then:
   Set **NewNode-**>NEXT = **HEAD** and **HEAD = NewNode** and return

[Search the LOCATION in Middle or at end.]
[Here CURR is the position where to insert and PREV points to previous node of the CURR node]

7. Set Prev = NULL and Set Curr = NULL;

8. Repeat for CURR =HEAD until CURR ≠ NULL after iteration CURR = CURR ->Next
   c) if(NewNode->Info <= CURR ->Info) then: break the loop
   d) Set PREV = CURR;
   [end of loop of step-7]

[Insert after PREV node (in middle or at end) of the list]

9. Set **NewNode**->NEXT = PREV ->NEXT and
10. Set PREV ->NEXT= **NewNode**.

11. Exit.

Delete a node from list:

The following algorithm deletes a node from any position in the LIST.

Algorithm: DELETE( )

LIST is a linked list in the memory. This algorithm gets **ITEM** from user and deletes the node where **ITEM** first appear in the **LIST**, otherwise it writes "NOT FOUND"

1. if **HEAD** =NULL then write: "Empty List" and return [Check for Empty List]
2. Get value for **ITEM** to delete it from **LIST**
3. if **ITEM** = **HEAD** -> Info then: [Top node is to delete]
Set **HEAD** = **HEAD** -> Next and return

[Search the ITEM in Middle or at end to delete]
[Here CURR will be the position to delete and PREV pointer points to previous node of the CURR node]
4. Set Prev = NULL and Set Curr = NULL;

5. Repeat for CURR = HEAD until CURR not = NULL
and after every iteration update CURR = CURR ->Next

a) if(NewNode->Info = CURR ->Info) then: break the loop
b) Set PREV = CURR;
[end of loop of step-5]

6. if(CURR = NULL) then write : Item not found in the list and return

[delete the current node from the list]
Set PPRE ->NEXT = CURR ->NEXT
7. End.

Source code:

```
// A Program that exercise the operations on Liked List
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#include<iostream.h>
#include <process.h>

struct LIST
{
        int Info;
        struct LIST *Next;
};

struct LIST *HEAD = NULL;

struct LIST *PREV, *CURR;
```

```
void AddNode(int ITEM)
{









} // end of AddNode function

void DeleteNode()
{









}// end of DeleteNode function
```

```
void Traverse()
{



} //  end of Traverse function

int main()
{ int inf, ch;
  while(1)
  { cout<< " \n\n\n\n Linked List Operations\n\n";
    cout<< "  1- Add Node  \n  2- Delete Node \n";
    cout<< "  3- Traverse List \n  4- exit\n";
    cout<< "\n\n  Your Choice: "; cin>>ch;
         switch(ch)
         { case 1: cout<< "\n  Put info/value to Add: ";
                   cin>>inf);
                   AddNode(inf);
                   break;
           case 2:  DeleteNode();  break;
           case 3:  cout<< "\n Linked List Values:\n";
     Traverse(); break;
           case 4: exit(0);
         }  // end of switch
  } //  end of while loop
 return 0;
} // end of main ( ) function
```

BINARY TREE

There are many operations that we often want to perform on trees. One notion that arises frequently is the idea of traversing a tree or visiting each node in the tree exactly once. A full traversal produces a linear order for the information in a tree. A precise way of describing this traversal is to write it as a **recursive procedure**. There are three common traversals for binary trees:

- Preorder
- Inorder
- Postorder

Aim: Write a program to create a **Binary Search Tree (BST)** and different traversing procedure related to it as stated above.

Description: Here **TREE** is a structure has three members (**Info**, **Left** and **Right**). Here **Info** is used to store the data of the node. **Left** and **Right** are pointers used to hold the address of left and right child respectively. If a nodes does not has (Left / Right) child then its relevant **Left** or/and **Right** pointer holds NULL value. **ROOT** is a pointer to **TREE**, points to the root node of the **TREE. ITEM** is the value needed to be insert or delete in/from the **TREE**. The initial value of the **ROOT** is **NULL,** it means that initially **TREE** is empty. **NewNode** is pointer variable which holds the address of newly created **TREE** node.

Format of struct :

```
struct TREE
    {       int     Info;
            struct TREE *Left;
            struct TREE *Right;
    };
```

Preorder Traversal:

Algorithm: PREORDER (pRoot)

First time this function is called by passing original ***ROOT*** into ***pRoot***. Here ***pRoot*** is pointers pointing to current root. This algorithm does a preorder traversal of **TREE**, applying by **recursively** calling same function and updating ***proot*** to traverse in a way i.e. (**NLR**) Node, Left, Right.

1. If (pRoot NOT = NULL) then: [does child exist ?]
   - **a)** Apply PROCESS to pRoot-> info. [e.g. Write: pRoot -> info]
     [recursive call by passing address of left child to update **pRoot**]
   - **b)** PREORDER (pRoot -> Left)
     [recursive call by passing address of right child to update **pRoot**]
   - **c)** PREORDER(pRoot -> Right)

   [End of If structure.]
2. End.

Inorder Traversal:

Algorithm: INORDER (pRoot)

First time this function is called by passing original ***ROOT*** into ***pRoot***. Here ***pRoot*** is pointers pointing to current root. This algorithm does a Inorder traversal of **TREE**, applying by **recursively** calling same function and updating ***proot*** to traverse in a way i.e. (**LNR**) Left, Node, Right.

1. If (pRoot NOT = NULL) then: [does child exist ?]
[recursive call by passing address of left child to update **pRoot**]
a) INORDER (pRoot -> Left)
b) Apply PROCESS to pRoot-> info. [e.g. Write: pRoot -> info]
[recursive call by passing address of right child to update **pRoot**]
c) INORDER (pRoot -> Right)
[End of If structure.]

2. End.

Postorder Traversal:

Algorithm: POSTORDER (pRoot)

First time this function is called by passing original ***ROOT*** into ***pRoot***. Here ***pRoot*** is pointers pointing to current root. This algorithm does a PostOrder traversal of **TREE**, applying by **recursively** calling same function and updating ***proot*** to traverse in a way i.e. (**LRN**) Left, Right, Node.

1. If (pRoot NOT = NULL) then: [does child exist ?]
[recursive call by passing address of left child to update **pRoot**]
a) POSTORDER (pRoot -> Left)
[recursive call by passing address of right child to update **pRoot**]
b) POSTORDER (pRoot -> Right)
c) Apply PROCESS to pRoot-> info. [e.g. Write: pRoot -> info]
[End of If structure.]

2. End.

Source code:

```
/*  This program is about to implement different operations on Binary
      Search Tree. */
#include <iostream.h>
#include <stdlib.h>
#include <process.h>

struct TREE
{
        int      Info;
        struct TREE *Left;
        struct TREE *Right;
};

 struct TREE *ROOT = NULL; // initially  ROOT is NULL

void AttachNode( struct TREE *pRoot, struct TREE *NewNode )
{









}
void Insert(int ITEM)
{




}
```

```
void Pre_Order(struct TREE *pRoot)
{




}

void Post_Order(struct TREE *pRoot)
{




}

void In_Order(struct TREE *pRoot)
{




}

void DisplayDescending(struct TREE *pRoot)
{




}
```

```
void DeleteTree( struct TREE *pRoot)  // This function deletes all nodes in the tree
{









}

int main( void )
{
        int choice, ITEM;
        while( 1 )
        {       cout<<"\n\n\n   Binary Search Tree Functions\n\n";
                cout<<"\n1. Insert a New Node";
                cout<<"\n2. Remove Existing Node";
                cout<<"\n3. In-Order Traverse  (Ascending Order)";
                cout<<"\n4. Pre-Order Traverse ";
                cout<<"\n5. Post-Order Traverse ";
                cout<<"\n6. Display in Descending Order   (Reverse)";
                cout<<"\n7. Exit";
                cout<<"\nEnter you choice: ";
                cin>> choice;

                switch(choice)
                {
                case 1:
                        cout<<"\n\n put a number: "; cin>>item;
                        Insert(item);
                        break;

                case 2:
//                      Remove();  // This function is not defined.
                        break;   //  Students shall write this function as home work.

                case 3:
                        cout<<"\n\n\n In-Order Traverse (ASCENDING ORDER)\n";
                        In_Order(ROOT);
                        cout<<"\n\n";
                        break;
```

```
                case 4:
                        cout<<"\n\n\n Pre-Order Traverse \n";
                        Pre_Order(ROOT);
                        cout<<"\n\n";
                        break;
                case 5:
                        cout<<"\n\n\n Post-Order Traverse \n";
                        Post_Order(ROOT);
                        cout<<"\n\n";
                        break;

                case 6:
                        cout<<"\n\n\nDESCENDING ORDER  (Reverse )\n";
                        DisplayDescending(ROOT);
                        cout<<"\n\n";
                        break;

                case 7:
                          DeleteTree(ROOT);
                        exit(0);

                default:
                        cout<<"\n\nInvalid Input");

                } // end of switch
        } // end of while loop
} // end of main( ) function
```

S O R T I N G

Sorting:

Sorting and searching are fundamental operations in computer science. Sorting refers to the operation of arranging data in some given order. Such as increasing/ascending or decreasing/descending order, with numeric data or alphabetically. There are so many ways / techniques to do sorting. Following are some techniques:

AIM: Write a program to **Sort** values in Ascending / Increasing Order using **Bubble Sort** Technique in Linear Array:

Description: Sorting refers to the operation of re-arrangements of values in Ascending or Descending order of the Linear Array '**A**'. Here '**A**' is Linear Array and **N** is the number of values stored in **A.**

Algorithm: (Bubble Sort) BUBBLE (A, N)
Here ***A*** is an Array with ***N*** elements stored in it. This algorithm sorts the elements in ***A***.

1. Repeat Steps 2 to 4 for Pass = 1 to N – 1
2. Set Swapped = 0 and K = 0
3. Repeat while K < (N – Pass)
   - **(a)** if ***A***[K] > ***A***[K + 1] then
     Interchange ***A***[K] and ***A***[K + 1] and
     Set Swapped = 1
     [End of if structure.]
   - **(b)** Set K = K + 1

   [End of inner loop of step-3.]
4. if Swapped = 0 then break the outer loop of step-1
   [End of outer loop of step-1.]
5. End

Source code:

```
/* This program sorts the array elements in the ascending order using bubble
sort  method */

#include <iostream.h>
int const  N = 6;

void BubbleSort(int [ ], int); // function prototyping

int main()
{
int A[N]={77,42,35,12,101,6}; //You can also get values from user for the array.
int i;
cout<< "The elements of the array before sorting\n";
for (i=0; i< N ; i++) cout<< A[i]<<",  ";
BubbleSort(A, N);
cout<< "\n\nThe elements of the array after sorting\n";
for (i=0; i< N; i++) cout<<A[i]<< ",  ";
return 0;
}

void BubbleSort(int A[N],  int N)
{









}
```

Insertion Sort:

AIM: Write a program to **Sort** values in Ascending / Increasing Order using **Insertion Sort** Technique in Linear Array:

Description: Here '**A**' is Linear Array and **N** is the number of values stored in **A.**

Algorithm: (INSERTION SORT) INSERTION (A, N)
[Where **A** is an array and **N** is the number of values in the array]

1. Repeat steps 2 to 4 for K=1,2,3, ... N-1:
2. Set TEMP = A[K] and i =K-1.
3. Repeat while i >= 0 and TEMP < A[i]
   a) Set A[i+1] = A[i]. [Moves element forward.]
   b) Set i = i -1.
   [End of loop.]
4. Set A[i+1] =TEMP. [Insert element in proper place.]
   [End of Step 2 loop.]
5. Return.

Source code:

```
//      Programe of Implementation of    Insertion SORT
#include<iostream.h>
#include<conio.h>

const int SIZE = 100;
void InsertionSort( int x[],int );

void main(void)
{       int i,N;
        int     A[SIZE];
        cout<"\nData Collection for Insertion Sort\n";
        cout<< "\nHow many value are to process: ";
        cin>> N;
        cout<<"\nINPUT "<<N<<" values:\n";
        for( i=0; i< N ; i++ ) cin>>A[i];
        InsertionSort( A,N);
        cout<< "\nSORTED ARRAY\n";
        for( i=0; i < N; i++ ) A[i]<<'\t';
}

void InsertionSort( int A[100], int N )
{




}
```

Selection Sort:

AIM: Write a program to **Sort** values in Ascending / Increasing Order using **Selection Sort** Technique in Linear Array:

Description: Here '**A**' is Linear Array and **N** is the number of values stored in **A.**

Algorithm: (SELECTION SORT) SELECTION (A, N)

1. Repeat steps 2 and 3 for K=0,1,2, ... N-2:
2. Call MIN(A, K, N, LOC).
   [Interchange A[K] and A[LOC].]
3. Set TEMP = A[K], A[K] = A[LOC] and A[LOC] = TEMP.
   [End of step 1 loop.]
4. Exit.

Algorithm: MIN(A, K, N, LOC)

[An Array **A** is in memory. This procedure finds the location Loc of the smallest element among A[K], A[K+1], ...A[N-1].]

1. Set MIN = A[K] and LOC = K. [Initializes pointers.]
2. Repeat for J=K+1, K+2, ... N:
   If MIN > A[J], then: MIN = A[J] and LOC = J.
3. Return LOC.

Source code:

```
// Programe of Implementation of    Selection   SORT

#include<iostream.h>
#include<conio.h>

int const SIZE = 100;

// function prototyping
void SelectionSort( int [], int);
int  MIN(int [], int);

void main(void)
{       int i,N ;
        int      A[SIZE];
        cout<< "\n Selection Sort\n";
        cout<< "How many values are to process: "; cin>>N;
        cout<< "\PUT "<<N<< " Random values\n";
        for( i=0; i< N; i++ ) cin>> A[i] );
        SelectionSort(A, N);
        cout<<"\nSORTED ARRAY\n";
        for( i=0; i< N; i++ ) cout<< A[i]<<"\t";
 }

void SelectionSort( int A[SIZE], int N )
{









}


int  MIN(int A[SIZE], int K)









  }
```

Merge Sort:

AIM: Write a program to **Sort** values in Ascending / Increasing Order using **Merge Sort** Technique in Linear Array:

Description: Here '**A**' is Linear Array and **N** is the number of values stored in **A.** Here, recursive function is going to use for MERGE SORT. For detail about this algorithm, see the lecture notes written by Shahid Iqbal Lone, Lecturer Computer College, Qassim University K.S.A.

Algorithm: MergeSort (A, BEG, END)
[Here **A** is an unsorted array. **BEG** is the lower bound and **END** is the upper bound.]

1. If (BEG < END) Then
   - **a)** Set MID = (BEG + END) / 2
   - **b)** Call MergeSort (A, BEG, MID)
   - **c)** Call MergeSort (A, MID + 1, END)
   - **d)** Call MERGE (A, BEG, MID, END)

   [End of If of step-1]
2. Return

Algorithm: MERGE (A, BEG, MID, END)
[Here **A** is an unsorted array. **BEG** is the lower bound, **END** is the upper bound and **MID** is the middle value of array. **B** is an empty array.]

1. Repeat for I = BEG to END
   Set B[I] = A[I] [Assign array A to B]
   [End of For Loop]
2. Set I = BEG, J = MID + 1, K = BEG
3. Repeat step-4 While (I <= MID) and (J <= END)
4. If (B[I] <= B[J]) Then [Assign smaller value to A]
   - **a)** Set A[K] = B[I]
   - **b)** Set I = I + 1 and K = K + 1

   Else
   - **a)** Set A[K] = B[J]
   - **b)** Set J = J + 1 and K = K + 1

   [End of If]
   [End of While Loop of step-3]
5. If (I <= MID) Then [Check whether first half has finished or not]
   - **a)** Repeat While (I <= MID)
     - **i.** Set A[K] = B[I]
     - **ii.** Set I = I + 1 and K = K + 1

     [End of While Loop step-5(**a**)]

   Else
   - **b)** Repeat While (J <= END)
     - **i.** Set A[K] = B[J]
     - **ii.** Set J = J + 1 and K = K + 1

     [End of While Loop step-5(**b**)]

   [End of If of step-5]
6. End

Source code:

// C++ Code (MERGE SORT)

```
#include <iostream.h>

// function prototyping
void MergeSort(int [], int, int);
void MERGE(int [], int, int, int);

int main()
 {    int A[50], N, I;
      cout<<"\nEnter how many values for the array: ";
      cin>>N;
      cout<<"\nEnter "<<N<<" elements  for array:\n";
      for(I=0; I<N; I++) cin>> A[I];
      MergeSort(A, 0, N-1);
      cout<<"\n\nAfter sorting:\n";
      for(I=0; I<N; I++)
      cout<<A[I]<< "\t";
      return 0;
 }

void MergeSort(int A[], int BEG, int END)
 {









}

void MERGE(int A[], int BEG, int MID, int END)
{
```

```
} // end of function
```

Quick Sort:

AIM: Write a program to **Sort** values in Ascending / Increasing Order using **Quick Sort** Technique in Linear Array:

Description: Here '**A**' is Linear Array and **N** is the number of values stored in **A.** Here, recursive function is going to use for QUICK SORT. For detail about this algorithm, see the lecture notes written by Shahid Iqbal Lone, Lecturer Computer College, Qassim University K.S.A.

Quick sort is an algorithm of the divide-and-conquer type. That is, the problem of sorting a set is reduced to the problem of sorting two smaller sets.

Algorithm: QUICKSORT (A, LEFT, RIGHT)

1. If LEFT ≥ RIGHT, then: Return.
2. Set MIDDLE = PARTITION (A, LEFT, RIGHT).
3. Call QUICKSORT (A, LEFT, MIDDLE-1).
4. Call QUICKSORT (A, MIDDLE+1, RIGHT).
5. Return.

Algorithm: PARTITION (A, LEFT, RIGHT)

1. Set X = A[LEFT].
2. Set I = LEFT.
3. Repeat for j = LEFT+1,LEFT+2, ... RIGHT
   If A[j] < X, then:
   Set i = i+1.
   SWAP (A[i], A[j]). [Interchange A[i] and A[j].]
4. SWAP (A[i], A[LEFT]. [Interchange A[i] and A[LEFT].]
5. Return i.

Source code:

```
// C++ Code (QUICK SORT):
#include <iostream.h>
#include <conio.h>

// Global Declarations
int const MAX=100;
int A[MAX];
int N;

// function Definition before main( )

void get()
{       cout<<"\n\n How many values you want to sort --> ";
        cin>>N;
        if(N>MAX)
          {cout<<"\n\n \t Maximum values Limits "<<MAX<<" so try again...";
                get(); // recursive call
          }
        cout<<"\n\t\t Put "<<N<<" random values..\n\n";
        for(int i=0 ; i < N ; i++)
              {cin>>A[i];}
}


int  partition(int LEFT, int RIGHT)
{









}


void quick_sort(int LEFT, int RIGHT)
{



}
```

```
void  display()
{
        cout<<"\n\n\n\n\t\t\t\t Sorted List\n\n";
        cout<<"\t\t\t\t(QUICK SORT)\n\n\n";
        for(int i=0 ; i < N ; i++)
              {cout<<A[i]<<"\t";}
}
void main()
{
       get();
       quick_sort(0,N-1); // Here  0  is  Lower Bound  and  N-1  is Upper Bound
       display();

}
```

Radix Sort:

AIM: Write a program to **Sort** values in Ascending / Increasing Order using **Radix Sort** Technique in Linear Array:

Description: Here '**A**' is Linear Array and **N** is the number of values stored in **A** and **R** is the Radix/Base (**10**) of Number System . Here, recursive function is going to use for RADIX SORT. For detail/explanation about this algorithm, see the lecture notes written by Shahid Iqbal Lone, Lecturer Computer College, Qassim University K.S.A.

Algorithm: RADIXSORT (A, N, R)

1. If N=1, then: Return.
2. Set D=R/10, P=-1.
3. Repeat for i=0,1,2 . . . . (N-1)
   Repeat for j=0,1,2 . . . 9
   Set C [i] [j] := -1.
4. Repeat for i=0,1,2 . . . . (N-1)
   i. Set X=A [i] % R.
   ii. Set m=X/D.
   iii. If m>0, then Set Z=1.
   iv. Set C [i] [m] = A [i].
5. Repeat for j=0,1,2 9
   i. Repeat for j=0,1,2 . . . . (N-1)
      1. If C [i] [j] ≠ -1,then:
         a. Set P=P+1.
         b. Set A [P]=C [i] [j].
6. If Z=1, then
   RADIXSORT (A, N, R*10).
7. Return.

Source code:

// C++ Code (RADIX SORT):

```
#include <iostream.h>
#include <conio.h>

int const MAX = 100;
int A[MAX];
int C[MAX][10]; // Two dimensional array, where each row has ten pockets
int N;

void get_Values()
{       cout<<"\n\tHow many values you want to sort -->";
         cin>>N;
         if(N>MAX)
                {cout<<"\n\n \t Maximum values Limits "<<MAX<<" so try again...";
                  get_Values(); // recursive call
                }
         int i=0;
         cout<<"\n Put "<<N<<"  Values..\n";
         for(i=0 ; i< N ; i++)   cin>>A[i];
}

void  RadixSort (int  R)
{









} // end of RadixSort( ) function
```

```
void display()
{
        cout<<"\n\n\t\t\t Sorted List\n\n";
        for(int i =0; i < N ; i++)    cout<<A[i]<<"\t";
}


void main()
       {

        get_Values();
        if(N>1) RadixSort (10); // Here 10 is the Radix of Decimal Numbersl
        display();
        cout<<endl<<endl;
       }
```