Laboratory Manual

Data Structure
(CS-303)

For

Second Year Students
Department: Computer Science & Engineering

INSTITUTE OF TECHNOLOGY & MANAGEMENT
GWALIOR • MP • INDIA

Department of Computer Science and Engineering

Vision of CSE Department:

The department envisions to nurture students to become technologically proficient, research competent and socially accountable for the welfare of the society.

Mission of the CSE Department:

I. To provide high quality education through effective teaching-learning process emphasizing active participation of students.

II. To build scientifically strong engineers to cater to the needs of industry, higher studies, research and startups.

III. To awaken young minds ingrained with ethical values and professional behaviors for the betterment of the society.

Program Educational Objectives:

Graduates will be able to

I. Our engineers will demonstrate application of comprehensive technical knowledge for innovation and entrepreneurship.

II. Our graduates will employ capabilities of solving complex engineering problems to succeed in research and/or higher studies.

III. Our graduates will exhibit team-work and leadership qualities to meet stakeholder business objectives in their careers.

IV. Our graduates will evolve in ethical and professional practices and enhance socioeconomic contributions to the society.

INSTITUTE OF TECHNOLOGY & MANAGEMENT
GWALIOR • MP • INDIA

Program Outcomes (POs):

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering Fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **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.
3. **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.
4. **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.
5. **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.
6. **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.
7. **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.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **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.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **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.

INSTITUTE OF TECHNOLOGY & MANAGEMENT
GWALIOR • MP • INDIA

CourseOutcomes

Data Structure (CS-303)

CO1:	Ability to Define, understand concepts of different categories of Data Structures.
CO2 :	Identify different parameters to analyze the performance of an algorithm. .
CO3 :	Design algorithms to perform operations with Linear and Nonlinear Data Structures
CO4 :	Compare and contrast different implementations of Data Structures.
CO5 :	Apply appropriate Data Structure to solve and implement various real time problems

Course	Course Outcomes	CO Attainment	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	Ability to Define, understand concepts of different categories of data Structures.		3				1								2		
CO2	Identify different parameters to analyze the performance of an algorithm.		2				2								1	2	
CO3	Design algorithms to perform operations with Linear and Nonlinear data structures										1					1	2
CO4	Compare and contrast different implementations of data structures.				1		1					1			2		1
CO5	Apply appropriate data structure to solve and implement various real time problems											1		1	1		1

INSTITUTE OF TECHNOLOGY & MANAGEMENT
GWALIOR • MP • INDIA

List of Program

S. No.	List	Course Outcome	Page No.
	INTRODUCTION TO DATA STRUCTURE		1
1	Write a program to demonstrate the working of traversing, insertion at specified location and deletion in array.	CO1, CO2, CO4	2-5
2	WAP to search in array by using linear search.	CO3,CO4	6
3	WAP to search in array by using binary search.	CO3,CO4	7-10
4	Write a program to demonstrate the working of bubble sort.	CO3,CO4	11-14
5	Write a menu driven program to implement following operation in a singly link list. a. Insertion a node of the front of link list. b. Insertion a node of the last of link list c. Insert a node such that the list is in ascending order.	CO4	15-21
6	Write a menu driven program to implement following operation in a singly link list. a. Delete a node of the front of link list. b. Delete a node of the last of link list c. Delete a node after specific position.	CO4	22-29
7	WAP to add two polynomial in link list.	CO5	30-33
8	WAP to demonstrate stack operations a. push b. pop	CO3,CO4	34-36
9.	WAP to convert infix expression in to postfix expression by using stack.	CO5	37-40
10.	WAP to implement QUEUE using array that perform following operations a. Insert b. Delete c. Display	CO4	41-45
11.	WAP to implement tree traversing in BST.	CO3,CO4	46-48
12.	WAP to implement BST.	CO3,CO4	49-51
13.	WAP to demonstrate working of selection sort.	CO3,CO4	52-54
14.	WAP to demonstrate working of insertion sort..	CO3,CO4	55-57

15.	WAP to demonstrate working Quick sort.	CO3,CO4	58-60
16.	WAP to demonstrate working of Merge sort.	CO3,CO4	61-62

INTRODUCTION TO DATA STRUCTURE

A **data structure** is a special way of organizing and storing data in a computer so that it can be used efficiently. Array, Linked List, Stack, Queue, Tree, Graph etc are all data structures that stores the data in a special way so that we can **access and use the data efficiently**. Each of these mentioned **data structures** has a different special way of organizing data so we choose the data structure based on the requirement, we will cover each of these data structures in a separate tutorials.

Data Structure plays a very important role in various algorithms, it allows the programmer to handle the data efficiently. In layman's terms you can say that the data structure allows faster data storing and retrieval.

LAB REQUIREMENTS

For Data Structure Programming

- Turbo C++
- 4 GB Ram

This Compiler has no special hardware requirements as such. Any System with a minimum 256 MB RAM and any normal processor can use for this lab.

Program-1

Write a program to demonstrate the working of traversing, insertion at specified location and deletion in array.

Algorithm:

Traversing:

Step 01: Start.

Step 02: [Initialize counter variable.] Set i = LB .

Step 03: Repeat for i = LB to UB .

Step 04: Apply process to arr[i] .

Step 05: [End of loop.]

Step 06: Stop.

Insertion:

[Initialize Counter] Set J: = N.

Repeat steps 3 and 4 while J ≥ K.

[Move Jth element downward] Set LA [J+1]: = LA [J]

[Decrease Counter] Set J: = J-1. [End of step 2 loop]

[Insert element] Set LA [K]: = ITEM.

[Reset N] Set N: = N+1.

Exit.

Deletion:

Step 01: Start.

Step 02: [Initialize counter variable.] ...

Step 03: Repeat Step 04 and 05 for i = pos - 1 to i < size.

Step 04: [Move i^{th} element backward (left).] ...

Step 05: [Increase counter.] ...

Step 06: [End of step 03 loop.]

Step 07: [Reset size of the array.] ...

Step 08: Stop.

Implementation:

Traversing:

```
#include <stdio.h>
main() {
   int LA[] = {1,3,5,7,8};
   int item = 10, k = 3, n = 5;
   int i = 0, j = n;
   printf("The original array elements are :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

OUTPUT:

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
```

Insertion:

```
#include <stdio.h>
main() {
   int LA[] = {1,3,5,7,8};
   int item = 10, k = 3, n = 5;
   int i = 0, j = n;
   printf("The original array elements are :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
   n = n + 1;
   while( j >= k) {
      LA[j+1] = LA[j];
      j = j - 1;
   }
   LA[k] = item;
   printf("The array elements after insertion :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

OUTPUT:

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after insertion :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
LA[5] = 8
```

Deletion:

```
#include <stdio.h>
void main() {
   int LA[] = {1,3,5,7,8};
   int k = 3, n = 5;
   int i, j;
  printf("The original array elements are :\n");
 for(i = 0; i<n; i++) {
 printf("LA[%d] = %d \n", i, LA[i]);
   }
    j = k;
   while( j < n) {
     LA[j-1] = LA[j];
     j = j + 1;
   }
 n = n -1;
 printf("The array elements after deletion :\n");
 for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

Output:

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
```

The array elements after deletion :
LA[0] = 1
LA[1] = 3
LA[2] = 7
LA[3] = 8

Program -2

WAP to search in array by using linear search.

Algorithm:

Step 1: Select the first element as the current element.
Step 2: Compare the current element with the target element. If matches, then go to *step 5*.
Step 3: If there is a next element, then set current element to next element and go to *Step 2*.
Step 4: Target element not found. Go to *Step 6.*
Step 5: Target element found and return location.
Step 6: Exit process.

Implementation:

```
#include <stdio.h>
 int search(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

// Driver code
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function call
    int result = search(arr, n, x);
    (result == -1)
        ? printf("Element is not present in array")
        : printf("Element is present at index %d", result);
    return 0;
}
```

Output

Element is present at index 3

Program -3

WAP to search in array by using binary search.

Algorithm

```
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
```

Implementation:

```
#include <stdio.h>
#define MAX 20
// array of items on which linear search will be conducted.
int intArray[MAX] = {1,2,3,4,6,7,9,11,12,14,15,16,17,19,33,34,43,45,55,66};
void printline(int count) {
   int i;
   for(i = 0;i <count-1;i++) {
      printf("=");
```

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

int find(int data) {
   int lowerBound = 0;
   int upperBound = MAX -1;
   int midPoint = -1;
   int comparisons = 0;
   int index = -1;

   while(lowerBound <= upperBound) {
      printf("Comparison %d\n" , (comparisons +1) );
      printf("lowerBound : %d, intArray[%d] = %d\n",lowerBound,lowerBound,
         intArray[lowerBound]);
      printf("upperBound : %d, intArray[%d] = %d\n",upperBound,upperBound,
         intArray[upperBound]);
      comparisons++;

      // compute the mid point
      // midPoint = (lowerBound + upperBound) / 2;
      midPoint = lowerBound + (upperBound - lowerBound) / 2;

      // data found
      if(intArray[midPoint] == data) {
         index = midPoint;
         break;
      } else {
         // if data is larger
         if(intArray[midPoint] < data) {
            // data is in upper half
            lowerBound = midPoint + 1;
         }
         // data is smaller
         else {
            // data is in lower half
```

```
            upperBound = midPoint -1;
         }
      }
   }
   printf("Total comparisons made: %d" , comparisons);
   return index;
}

void display() {
   int i;
   printf("[");

   // navigate through all items
   for(i = 0;i<MAX;i++) {
      printf("%d ",intArray[i]);
   }

   printf("]\n");
}

void main() {
   printf("Input Array: ");
   display();
   printline(50);

   //find location of 1
   int location = find(55);

   // if element was found
   if(location != -1)
      printf("\nElement found at location: %d" ,(location+1));
   else
      printf("\nElement not found.");
}
```

Output

Input Array: [1 2 3 4 6 7 9 11 12 14 15 16 17 19 33 34 43 45 55 66]
Comparison 1
lowerBound : 0, intArray[0] = 1
upperBound : 19, intArray[19] = 66
Comparison 2
lowerBound : 10, intArray[10] = 15
upperBound : 19, intArray[19] = 66
Comparison 3
lowerBound : 15, intArray[15] = 34
upperBound : 19, intArray[19] = 66
Comparison 4
lowerBound : 18, intArray[18] = 55
upperBound : 19, intArray[19] = 66
Total comparisons made: 4
Element found at location: 19

Program-4

Write a program to demonstrate the working of bubble sort.

Algorithm

```
begin BubbleSort(list)
   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for
      return list
  end BubbleSort
```

Implementation:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 10
int list[MAX] = {1,8,4,6,0,3,5,2,7,9};
void display() {
   int i;
   printf("[");
      // navigate through all items
   for(i = 0; i < MAX; i++) {
      printf("%d ",list[i]);
   }
      printf("]\n");
}

void bubbleSort() {
   int temp;
   int i,j;

   bool swapped = false;

   // loop through all numbers
   for(i = 0; i < MAX-1; i++) {
      swapped = false;
```

```
  // loop through numbers falling ahead
     for(j = 0; j < MAX-1-i; j++) {
       printf("     Items compared: [ %d, %d ] ", list[j],list[j+1]);

       // check if next number is lesser than current no
       //   swap the numbers.
       //  (Bubble up the highest number)

       if(list[j] > list[j+1]) {
         temp = list[j];
         list[j] = list[j+1];
         list[j+1] = temp;

         swapped = true;
         printf(" => swapped [%d, %d]\n",list[j],list[j+1]);
       } else {
         printf(" => not swapped\n");
       }

     }

     // if no number was swapped that means
     //   array is sorted now, break the loop.
     if(!swapped) {
       break;
     }

     printf("Iteration %d#: ",(i+1));
     display();
   }

}
void main() {
   printf("Input Array: ");
   display();
   printf("\n");

   bubbleSort();
   printf("\nOutput Array: ");
   display();
}
```

Output:

```
Input Array: [1 8 4 6 0 3 5 2 7 9 ]
    Items compared: [ 1, 8 ]  => not swapped
    Items compared: [ 8, 4 ]  => swapped [4, 8]
    Items compared: [ 8, 6 ]  => swapped [6, 8]
    Items compared: [ 8, 0 ]  => swapped [0, 8]
    Items compared: [ 8, 3 ]  => swapped [3, 8]
    Items compared: [ 8, 5 ]  => swapped [5, 8]
    Items compared: [ 8, 2 ]  => swapped [2, 8]
    Items compared: [ 8, 7 ]  => swapped [7, 8]
    Items compared: [ 8, 9 ]  => not swapped

Iteration 1#: [1 4 6 0 3 5 2 7 8 9 ]
    Items compared: [ 1, 4 ]  => not swapped
    Items compared: [ 4, 6 ]  => not swapped
    Items compared: [ 6, 0 ]  => swapped [0, 6]
    Items compared: [ 6, 3 ]  => swapped [3, 6]
    Items compared: [ 6, 5 ]  => swapped [5, 6]
    Items compared: [ 6, 2 ]  => swapped [2, 6]
    Items compared: [ 6, 7 ]  => not swapped
    Items compared: [ 7, 8 ]  => not swapped

Iteration 2#: [1 4 0 3 5 2 6 7 8 9 ]
    Items compared: [ 1, 4 ]  => not swapped
    Items compared: [ 4, 0 ]  => swapped [0, 4]
    Items compared: [ 4, 3 ]  => swapped [3, 4]
    Items compared: [ 4, 5 ]  => not swapped
    Items compared: [ 5, 2 ]  => swapped [2, 5]
    Items compared: [ 5, 6 ]  => not swapped
    Items compared: [ 6, 7 ]  => not swapped

Iteration 3#: [1 0 3 4 2 5 6 7 8 9 ]
    Items compared: [ 1, 0 ]  => swapped [0, 1]
    Items compared: [ 1, 3 ]  => not swapped
    Items compared: [ 3, 4 ]  => not swapped
    Items compared: [ 4, 2 ]  => swapped [2, 4]
    Items compared: [ 4, 5 ]  => not swapped
    Items compared: [ 5, 6 ]  => not swapped

Iteration 4#: [0 1 3 2 4 5 6 7 8 9 ]
    Items compared: [ 0, 1 ]  => not swapped
    Items compared: [ 1, 3 ]  => not swapped
    Items compared: [ 3, 2 ]  => swapped [2, 3]
    Items compared: [ 3, 4 ]  => not swapped
```

Items compared: [4, 5] => not swapped

Iteration 5#: [0 1 2 3 4 5 6 7 8 9]
Items compared: [0, 1] => not swapped
Items compared: [1, 2] => not swapped
Items compared: [2, 3] => not swapped
Items compared: [3, 4] => not swapped

Output Array: [0 1 2 3 4 5 6 7 8 9]

Program -5

Write a menu driven program to implement following operation in a singly link list.

a. Insertion a node of the front of link list.
b. Insertion a node of the last of link list
c. Insert a node such that the list is in ascending order.

Algorithm

Insertion a node of the front of link list

Step 1: IF PTR = NULL.
Step 2: SET NEW_NODE = PTR.
Step 3: SET PTR = PTR → NEXT.
Step 4: SET NEW_NODE → DATA = VAL.
Step 5: SET NEW_NODE → NEXT = HEAD.
Step 6: SET HEAD = NEW_NODE.
Step 7: EXIT.

Insertion a node of the last of link list

Step 1: **IF** AVAIL = **NULL**
Write OVERFLOW
Go to Step 10
[**END OF IF**]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = **NULL**
Step 6: SET PTR = HEAD
Step 7: Repeat Step 8 **while** PTR -> NEXT != **NULL**
Step 8: SET PTR = PTR -> NEXT
[**END OF** LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT

Insert a node such that the list is in ascending order

If Linked list is empty then make the node as
head and return it.
2) If the value of the node to be inserted is smaller
than the value of the head node, then insert the node
at the start and make it head.
3) In a loop, find the appropriate node after
which the input node (let 9) is to be inserted.
To find the appropriate node start from the head,
keep moving until you reach a node GN (10 in

the below diagram) who's value is greater than the input node. The node just before GN is the appropriate node (7).
4) Insert the node (9) after the appropriate node (7) found in step 3.

Implementation

```
#include<stdio.h>
#include<conio.h>
#include<process.h>

struct node
{
   int data;
   struct node *next;
}*start=NULL,*q,*t;

int main()
{
   int ch;
   void insert_beg();
   void insert_end();
   int insert_pos();
   void display();
   void delete_beg();
   void delete_end();
   int delete_pos();

   while(1)
   {
      printf("\n\n---- Singly Linked List(SLL) Menu ----");
      printf("\n1.Insert\n2.Display\n3.Delete\n4.Exit\n\n");
      printf("Enter your choice(1-4):");
      scanf("%d",&ch);

      switch(ch)
      {
         case 1:
               printf("\n---- Insert Menu ----");
               printf("\n1.Insert at beginning\n2.Insert at end\n3.Insert at specified position\n4.Exit");
               printf("\n\nEnter your choice(1-4):");
               scanf("%d",&ch);

               switch(ch)
               {
```

```
    case 1: insert_beg();
                    break;
                case 2: insert_end();
                    break;
                case 3: insert_pos();
                    break;
                case 4: exit(0);
                default: printf("Wrong Choice!!");
            }
            break;

        case 2: display();
            break;

        case 3: printf("\n---- Delete Menu ----");
            printf("\n1.Delete from beginning\n2.Delete from end\n3.Delete from specified
position\n4.Exit");
            printf("\n\nEnter your choice(1-4):");
            scanf("%d",&ch);

            switch(ch)
            {
                case 1: delete_beg();
                    break;
                case 2: delete_end();
                    break;
                case 3: delete_pos();
                    break;
                case 4: exit(0);
                default: printf("Wrong Choice!!");
            }
            break;
        case 4: exit(0);
            default: printf("Wrong Choice!!");
      }
   }
   return 0;
}

void insert_beg()
{
   int num;
   t=(struct node*)malloc(sizeof(struct node));
   printf("Enter data:");
   scanf("%d",&num);
```

```
  t->data=num;
   if(start==NULL)        //If list is empty
   {
      t->next=NULL;
      start=t;
   }
   else
   {
      t->next=start;
      start=t;
   }
}

void insert_end()
{
   int num;
   t=(struct node*)malloc(sizeof(struct node));
   printf("Enter data:");
   scanf("%d",&num);
   t->data=num;
   t->next=NULL;

   if(start==NULL)        //If list is empty
   {
      start=t;
   }
   else
   {
      q=start;
      while(q->next!=NULL)
      q=q->next;
      q->next=t;
   }
}

int insert_pos()
{
   int pos,i,num;
   if(start==NULL)
   {
      printf("List is empty!!");
      return 0;
   }

   t=(struct node*)malloc(sizeof(struct node));
```

```
 printf("Enter data:");
    scanf("%d",&num);
    printf("Enter position to insert:");
    scanf("%d",&pos);
    t->data=num;

    q=start;
    for(i=1;i<pos-1;i++)
    {
       if(q->next==NULL)
       {
          printf("There are less elements!!");
          return 0;
       }

       q=q->next;
    }

    t->next=q->next;
    q->next=t;
    return 0;
}

void display()
{
    if(start==NULL)
    {
       printf("List is empty!!");
    }
    else
    {
       q=start;
       printf("The linked list is:\n");
       while(q!=NULL)
       {
          printf("%d->",q->data);
          q=q->next;
       }
    }
}

void delete_beg()
{
    if(start==NULL)
    {
```

```
 printf("The list is empty!!");
    }
    else
    {
        q=start;
        start=start->next;
        printf("Deleted element is %d",q->data);
        free(q);
    }
}

void delete_end()
{
    if(start==NULL)
    {
        printf("The list is empty!!");
    }
    else
    {
        q=start;
        while(q->next->next!=NULL)
        q=q->next;

        t=q->next;
        q->next=NULL;
        printf("Deleted element is %d",t->data);
        free(t);
    }
}

int delete_pos()
{
    int pos,i;

    if(start==NULL)
    {
        printf("List is empty!!");
        return 0;
    }

    printf("Enter position to delete:");
    scanf("%d",&pos);

    q=start;
    for(i=1;i<pos-1;i++)
```

```
    {
        if(q->next==NULL)
        {
            printf("There are less elements!!");
            return 0;
        }
        q=q->next;
    }

    t=q->next;
    q->next=t->next;
    printf("Deleted element is %d",t->data);
    free(t);

    return 0;
}
```

Output

—- Singly Linked List(SLL) Menu —-
1.Insert
2.Display
3.Delete
4.ExitEnter your choice(1-4):1—- Insert Menu —-
1.Insert at beginning
2.Insert at end
3.Insert at specified position
4.ExitEnter your choice(1-4):1
Enter data:4—- Singly Linked List(SLL) Menu —-
1.Insert
2.Display
3.Delete
4.ExitEnter your choice(1-4):2
The linked list is:
4->
—- Singly Linked List(SLL) Menu —-
1.Insert
2.Display
3.Delete
4.Exit

Enter your choice(1-4):4

Program -6

Aim: Write a menu driven program to implement following operation in a singly link list.
a. Delete a node of the front of link list.
b. Delete a node of the last of link list
c. Delete a node after specific position.

Algorithm

Delete a node of the front of link list.
Step 1:
IF HEAD = NULL
Write UNDERFLOW
Go to Step 5
Step 2: SET PTR = HEAD
Step 3: SET HEAD = HEAD -> NEXT
Step 4: FREE PTR
Step 5: EXIT

Implementation

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

void create(int);
void deleteNode();
struct node
{
   int data;
   struct node *next;
};
struct node *head;
void main ()
{
   int choice,item;
   do
   {
      printf("\n1.Insert Node\n2.Delete Node\n3.Exit\n4.Enter your choice:\n");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         printf("\nEnter the element:\n");
         scanf("%d",&item);
         create(item);
         break;
```

```
            case 2:
            deleteNode();
            break;
            case 3:
            exit(0);
            break;
            default:
            printf("\nPlease enter a valid choice:\n");
        }

    }while(choice != 3);
}
void create(int item)
    {
        struct node *ptr = (struct node *)malloc(sizeof(struct node *));
        if(ptr == NULL)
        {
            printf("\nOVERFLOW\n");
        }
        else
        {
            ptr->data = item;
            ptr->next = head;
            head = ptr;
            printf("\nNode inserted successfully!!\n");
        }

    }
void deleteNode()
    {
        struct node *ptr;
        if(head == NULL)
        {
            printf("\nKindly Check. The list is empty.");
        }
        else
        {
            ptr = head;
            head = ptr->next;
            free(ptr);
            printf("\nNode successfully deleted from the beginning of the list!!");
        }
    }
```

Output

```
1.Insert Node
2.Delete Node
3.Exit
4.Enter your choice:
1

Enter the element:
2

Node inserted successfully!!

1.Insert Node
2.Delete Node
3.Exit
4.Enter your choice:
1

Enter the element:
3

Node inserted successfully!!

1.Insert Node
2.Delete Node
3.Exit
4.Enter your choice:
2

Node successfully deleted from the beginning of the list!!
1.Insert Node
2.Delete Node
3.Exit
4.Enter your choice:
3
```

Delete a node of the last of link list

Algorithm

Step 1:if head = null
Write underflow
Goto step 10
End of if
Step 2: set temp = head
Step 3: set i = 0
Step 4: repeat step 5 to 8 until i
Step 5: temp1 = temp
Step 6: temp = temp → next
Step 7:if temp = null
Write "desired node not present"
Goto step 12
End of if
Step 8:i = i+1
End of loop
Step 9: temp1 → next = temp → next
Step 10: free temp
Step 11: exit

Implementation

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

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

struct node *head;

void insertAtLast ();
void deleteAtRandom();
void show();
void main ()
{
   int choice =0;
   while(choice != 9)
   {

      printf("\n1.Insert Node\n2.Delete node after specified location\n3.Show\n4.Exit\n");
      printf("\nEnter your choice:\n");
      scanf("\n%d",&choice);
```

```
 switch(choice)
       {
          case 1:
          insertAtLast();
          break;
          case 2:
          deleteAtRandom();
          break;
          case 3:
          show();
          break;
          case 4:
          exit(0);
          break;
          default:
          printf("Enter a valid choice.");
       }
    }
}

void insertAtLast()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
       printf("\nOVERFLOW");
    }
    else
    {
       printf("\nEnter value:\n");
       scanf("%d",&item);
       ptr->data = item;
       if(head == NULL)
       {
          ptr -> next = NULL;
          head = ptr;
          printf("\nNode successfully inserted!!");
       }
       else
       {
          temp = head;
          while (temp -> next != NULL)
          {
             temp = temp -> next;
```

```
  }
        temp->next = ptr;
        ptr->next = NULL;
        printf("\nNode successfully inserted!!");

     }
   }
}

void deleteAtRandom()
{
   struct node *ptr,*ptr1;
   int loc,i;
   printf("\nEnter the location of the node after which you want to perform deletion: \n");
   scanf("%d",&loc);
   ptr=head;
   for(i=0;i<loc;i++)
   {
      ptr1 = ptr;
      ptr = ptr->next;

      if(ptr == NULL)
      {
         printf("\nDeletion not possible!!\n");
         return;
      }
   }
   ptr1 ->next = ptr ->next;
   free(ptr);
   printf("\nDeleted node %d ",loc+1);
}

void show()
{
   struct node *ptr;
   ptr = head;
   if(ptr == NULL)
   {
      printf("Nothing to print");
   }
   else
   {
      printf("\nPrinting list . . . . .\n");
      while (ptr!=NULL)
      {
         printf("\n%d",ptr->data);
```

```
    ptr = ptr -> next;
      }
    }
}
```

Output:

```
1.Insert Node
2.Delete node after specified location
3.Show
4.Exit

Enter your choice:
1

Enter value:
2

Node successfully inserted!!
1.Insert Node
2.Delete node after specified location
3.Show
4.Exit

Enter your choice:
1

Enter value:
4

Node successfully inserted!!
1.Insert Node
2.Delete node after specified location
3.Show
4.Exit

Enter your choice:
1

Enter value:
6

Node successfully inserted!!
1.Insert Node
2.Delete node after specified location
```

3.Show
4.Exit

Enter your choice:
3

Printing list . . . . .

2
4
6
1.Insert Node
2.Delete node after specified location
3.Show
4.Exit

Enter your choice:
2

Enter the location of the node after which you want to perform deletion:
1

Deleted node 2
1.Insert Node
2.Delete node after specified location
3.Show
4.Exit

Enter your choice:
3

Printing list . . . . .

2
6
1.Insert Node
2.Delete node after specified location
3.Show
4.Exit

Enter your choice:
4

Program -7

WAP to add two polynomial in link list.

Algorithm

- Create a new linked list, newHead to store the resultant list.
- Traverse both lists until one of them is null.
- If any list is null insert the remaining node of another list in the resultant list.
- Otherwise compare the degree of both nodes, a (first list node) and b (second list node). Here three cases are possible:

1) If the degree of a and b is equal, we insert a new node in the resultant list with the coefficient equal to the sum of coefficients of a and b and the same degree.
2) If the degree of a is greater than b, we insert a new node in the resultant list with the coefficient and degree equal to that of a.
3) If the degree of b is greater than a, we insert a new node in the resultant list with the coefficient and degree equal to that of b.

Implementation

```
#include<stdio.h>

struct Node {
   int coeff;
   int pow;
   struct Node* next;
};

// Function to create new node
void create_node(int x, int y, struct Node** temp)
{
   struct Node *r, *z;
   z = *temp;
   if (z == NULL) {
      r = (struct Node*)malloc(sizeof(struct Node));
      r->coeff = x;
      r->pow = y;
      *temp = r;
      r->next = (struct Node*)malloc(sizeof(struct Node));
      r = r->next;
      r->next = NULL;
   }
   else {
      r->coeff = x;
      r->pow = y;
```

```
        r->next = (struct Node*)malloc(sizeof(struct Node));
        r = r->next;
        r->next = NULL;
    }
}

// Function Adding two polynomial numbers
void polyadd(struct Node* poly1, struct Node* poly2,
             struct Node* poly)
{
    while (poly1->next && poly2->next) {
        if (poly1->pow > poly2->pow) {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff;
            poly1 = poly1->next;
        }

        else if (poly1->pow < poly2->pow) {
            poly->pow = poly2->pow;
            poly->coeff = poly2->coeff;
            poly2 = poly2->next;
        }
        else {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff + poly2->coeff;
            poly1 = poly1->next;
            poly2 = poly2->next;
        }

        // Dynamically create new node
        poly->next
            = (struct Node*)malloc(sizeof(struct Node));
        poly = poly->next;
        poly->next = NULL;
    }
    while (poly1->next || poly2->next) {
        if (poly1->next) {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff;
            poly1 = poly1->next;
        }
        if (poly2->next) {
            poly->pow = poly2->pow;
            poly->coeff = poly2->coeff;
            poly2 = poly2->next;
        }
```

```
  poly->next
          = (struct Node*)malloc(sizeof(struct Node));
       poly = poly->next;
       poly->next = NULL;
    }
}

// Display Linked list
void show(struct Node* node)
{
    while (node->next != NULL) {
       printf("%dx^%d", node->coeff, node->pow);
       node = node->next;
       if (node->coeff >= 0) {
          if (node->next != NULL)
             printf("+");
       }
    }
}

// Driver code
int main()
{
    struct Node *poly1 = NULL, *poly2 = NULL, *poly = NULL;

    // Create first list of 5x^2 + 4x^1 + 2x^0
    create_node(5, 2, &poly1);
    create_node(4, 1, &poly1);
    create_node(2, 0, &poly1);

    // Create second list of -5x^1 - 5x^0
    create_node(-5, 1, &poly2);
    create_node(-5, 0, &poly2);

    printf("1st Number: ");
    show(poly1);

    printf("\n2nd Number: ");
    show(poly2);

    poly = (struct Node*)malloc(sizeof(struct Node));

    // Function add two polynomial numbers
    polyadd(poly1, poly2, poly);

    // Display resultant List
```

```
  printf("\nAdded polynomial: ");
    show(poly);

    return 0;
}
```

Output

```
Polynomial A
 3x^4 + 9x^3 + 4x^2 + 7x^1 + 3
 Polynomial B
 7x^3 + 1x^2 + 6x^1 + 4
 Result
 3x^4 + 16x^3 + 5x^2 + 13x^1 + 7
```

Program -8

WAP to demonstrate stack operations
a. push
b. pop

Algorithm

Push

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.

Pop

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.

Implementation

```
#include <stdio.h>
int MAXSIZE = 8;
int stack[8];
int top = -1;
int isempty() {
  if(top == -1)
    return 1;
  else
    return 0;
}
  int isfull() {

  if(top == MAXSIZE)
    return 1;
  else
    return 0;
}

int peek() {
  return stack[top];
}

int pop() {
  int data;
```

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

int push(int data) {

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

int main() {
   // push items on to the stack
   push(3);
   push(5);
   push(9);
   push(1);
   push(12);
   push(15);

   printf("Element at top of the stack: %d\n" ,peek());
   printf("Elements: \n");

   // print stack data
   while(!isempty()) {
      int data = pop();
      printf("%d\n",data);
   }

   printf("Stack full: %s\n" , isfull()?"true":"false");
   printf("Stack empty: %s\n" , isempty()?"true":"false");

   return 0;
}
```

Output

Element at top of the stack: 15
Elements:
15
12
1
9
5
3
Stack full: false
Stack empty: true

Program -9

WAP to convert infix expression in to postfix expression by using stack

Algorithm

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
1 If the precedence and associativity of the scanned operator is greater than the precedence and associativity of the operator in the stack (or the stack is empty or the stack contains a '('), push it.
2 '^' operator is right associative and other operators like '+','-','*' and '/' are left associative. Check especially for a condition when both top of the operator stack and scanned operator are '^'. In this condition the precedence of scanned operator is higher due to it's right associativity. So it will be pushed in the operator stack. In all the other cases when the top of operator stack is same as scanned operator we will pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.
3 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and output it until a '('is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

Implementation

```
// C program to convert infix expression to postfix
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Stack type
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// Stack Operations
struct Stack* createStack( unsigned capacity )
{
    struct Stack* stack = (struct Stack*)
```

```
  malloc(sizeof(struct Stack));

    if (!stack)
        return NULL;

    stack->top = -1;
    stack->capacity = capacity;

    stack->array = (int*) malloc(stack->capacity *
                                 sizeof(int));

    return stack;
}
int isEmpty(struct Stack* stack)
{
    return stack->top == -1 ;
}
char peek(struct Stack* stack)
{
    return stack->array[stack->top];
}
char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--] ;
    return '$';
}
void push(struct Stack* stack, char op)
{
    stack->array[++stack->top] = op;
}


// A utility function to check if
// the given character is operand
int isOperand(char ch)
{
    return (ch >= 'a' && ch <= 'z') ||
           (ch >= 'A' && ch <= 'Z');
}

// A utility function to return
// precedence of a given operator
// Higher returned value means
// higher precedence
```

```
int Prec(char ch)
{
    switch (ch)
    {
    case '+':
    case '-':
        return 1;

    case '*':
    case '/':
        return 2;

    case '^':
        return 3;
    }
    return -1;
}


// The main function that
// converts given infix expression
// to postfix expression.
int infixToPostfix(char* exp)
{
    int i, k;

    // Create a stack of capacity
    // equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    if(!stack) // See if stack was created successfully
        return -1 ;

    for (i = 0, k = -1; exp[i]; ++i)
    {

        // If the scanned character is
        // an operand, add it to output.
        if (isOperand(exp[i]))
            exp[++k] = exp[i];

        // If the scanned character is an
        // ‘(‘, push it to the stack.
        else if (exp[i] == '(')
            push(stack, exp[i]);
```

```
        // If the scanned character is an ‘)’,
 // pop and output from the stack
        // until an ‘(‘ is encountered.
        else if (exp[i] == ')')
        {
            while (!isEmpty(stack) && peek(stack) != '(')
                exp[++k] = pop(stack);
            if (!isEmpty(stack) && peek(stack) != '(')
                return -1; // invalid expression
            else
                pop(stack);
        }
        else // an operator is encountered
        {
            while (!isEmpty(stack) &&
                 Prec(exp[i]) <= Prec(peek(stack)))
                exp[++k] = pop(stack);
            push(stack, exp[i]);
        }

    }

    // pop all the operators from the stack
    while (!isEmpty(stack))
        exp[++k] = pop(stack );

    exp[++k] = '\0';
    printf( "%s", exp );
}

// Driver program to test above functions
int main()
{
    char exp[] = "a+b*(c^d-e)^(f+g*h)-i";
    infixToPostfix(exp);
    return 0;
}
```

Output

```
abcd^e-fgh*+^*+i-
```

Program -10

WAP to implement QUEUE using array that perform following operations

a. Insert
b. Delete
c. Display

Algorithm

Insert

Step 1: IF REAR = MAX - 1

Write OVERFLOW

Go to step

[END OF IF]

Step 2: IF FRONT = -1 and REAR = -1

SET FRONT = REAR = 0

ELSE

SET REAR = REAR + 1

[END OF IF]

Step 3: Set QUEUE[REAR] = NUM

Step 4: EXIT

Delete

Step 1: IF FRONT = -1 or FRONT > REAR

Write UNDERFLOW

ELSE

SET VAL = QUEUE[FRONT]

SET FRONT = FRONT + 1

[END OF IF]

Step 2: EXIT

Implementation

```
#include<stdio.h>
#include<stdlib.h>
#define maxsize 5
void insert();
void delete();
void display();
int front = -1, rear = -1;
int queue[maxsize];
void main ()
{
   int choice;
   while(choice != 4)
   {
      printf("\n*************************Main Menu*****************************\n");
      printf("\n=================================================================
====\n");
      printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
      printf("\nEnter your choice ?");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         insert();
         break;
         case 2:
         delete();
         break;
         case 3:
         display();
         break;
         case 4:
         exit(0);
         break;
         default:
         printf("\nEnter valid choice??\n");
      }
   }
}
void insert()
{
   int item;
   printf("\nEnter the element\n");
```

```
    scanf("\n%d",&item);
    if(rear == maxsize-1)
    {
       printf("\nOVERFLOW\n");
       return;
    }
    if(front == -1 && rear == -1)
    {
       front = 0;
       rear = 0;
    }
    else
    {
       rear = rear+1;
    }
    queue[rear] = item;
    printf("\nValue inserted ");

}
void delete()
{
    int item;
    if (front == -1 || front > rear)
    {
       printf("\nUNDERFLOW\n");
       return;

    }
    else
    {
       item = queue[front];
       if(front == rear)
       {
          front = -1;
          rear = -1 ;
       }
       else
       {
          front = front + 1;
       }
       printf("\nvalue deleted ");
    }
     }
```

```
void display()
{
   int i;
   if(rear == -1)
   {
      printf("\nEmpty queue\n");
   }
   else
   {   printf("\nprinting values .....\n");
      for(i=front;i<=rear;i++)
      {
         printf("\n%d\n",queue[i]);
      }
   }
}
```

Output

***************Main Menu***************

==========

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter the element
123

Value inserted

***************Main Menu***************

==========

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter the element
90

Value inserted

***************Main Menu***************

==========

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?2

value deleted

***************Main Menu***************
==========

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

90

***************Main Menu***************

==========

1.insert an element
2.Delete an element
3.Display the queue
4.Exit
Enter your choice ?4

Program -11

WAP to implement tree traversing in BST

Algorithms

Inorder

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Preorder

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Postorder

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Implementation

```
/ C program for different tree traversals
#include <stdio.h>
#include <stdlib.h>
/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node {
    int data;
    struct node* left;
    struct node* right;
};
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node
        = (struct node*)malloc(sizeof(struct node));
```

```
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return (node);
}

/* Given a binary tree, print its nodes according to the
  "bottom-up" postorder traversal. */
void printPostorder(struct node* node)
{
    if (node == NULL)
        return;
    // first recur on left subtree
    printPostorder(node->left);
    // then recur on right subtree
    printPostorder(node->right);
    // now deal with the node
    printf("%d ", node->data);
}
/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);
    /* then print the data of node */
    printf("%d ", node->data);
    /* now recur on right child */
    printInorder(node->right);
}
/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct node* node)
{
    if (node == NULL)
        return;
    /* first print data of node */
    printf("%d ", node->data);
    /* then recur on left subtree */
    printPreorder(node->left);
    /* now recur on right subtree */
    printPreorder(node->right);
}
```

```
/* Driver program to test above functions*/
int main()
{
    struct node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    printf("\nPreorder traversal of binary tree is \n");
    printPreorder(root);
    printf("\nInorder traversal of binary tree is \n");
    printInorder(root);
    printf("\nPostorder traversal of binary tree is \n");
    printPostorder(root);
    getchar();
    return 0;
}
```

Output:

Preorder traversal of binary tree is

1 2 4 5 3

Inorder traversal of binary tree is

4 2 5 1 3

Postorder traversal of binary tree is

4 5 2 3 1

Program -12

WAP to implement BST

Algorithms

1. Create a new BST node and assign values to it.
2. insert(node, key)
   i) If root == NULL,
      return the new node to the calling function.
   ii) if root=>data < key
      call the insert function with root=>right and assign the return value in root=>right.
      root->right = insert(root=>right,key)
   iii) if root=>data > key
      call the insert function with root->left and assign the return value in root=>left.
      root=>left = insert(root=>left,key)
3. Finally, return the original root pointer to the calling function.

Implementation

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

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

//this function will return the new node with the given value
struct node *getNewNode(int val)
{
   struct node *newNode = malloc(sizeof(struct node));
   newNode->key   = val;
   newNode->left  = NULL;
   newNode->right = NULL;

   return newNode;
}
struct node *insert(struct node *root, int val)
{
   /*
    * It will handle two cases,
```

```
     * 1. if the tree is empty, return new node in root
     * 2. if the tree traversal reaches NULL, it will return the new node
     */
    if(root == NULL)
        return getNewNode(val);
    /*
     * if given val is greater than root->key,
     * we should find the correct place in right subtree and insert the new node
     */
    if(root->key < val)
        root->right = insert(root->right,val);
    /*
     * if given val is smallar than root->key,
     * we should find the correct place in left subtree and insert the new node
     */
    else if(root->key > val)
        root->left = insert(root->left,val);
    /*
     * It will handle two cases
     * (Prevent the duplicate nodes in the tree)
     * 1.if root->key == val it will straight away return the address of the root node
     * 2.After the insertion, it will return the original unchanged root's address
     */
    return root;
}
/*
 * it will print the tree in ascending order
 * we will discuss about it in the upcoming tutorials
 */
void inorder(struct node *root)
{
    if(root == NULL)
        return;
    inorder(root->left);
    printf("%d ",root->key);
    inorder(root->right);
}
```

```
int main()
{
   struct node *root = NULL;
   root = insert(root,100);
   root = insert(root,50);
   root = insert(root,150);
   root = insert(root,50);

   inorder(root);

   return 0;
}
```

Output

50 100 150

Program -13

WAP to demonstrate working of selection sort

Algorithms

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

Implementation

```
#include <stdio.h>

#include <stdbool.h>

#define MAX 7

int intArray[MAX] = {4,6,3,2,1,9,7};

void printline(int count) {
   int i;

   for(i = 0;i < count-1;i++) {
      printf("=");
   }

   printf("=\n");
}

void display() {
   int i;
   printf("[");

   // navigate through all items
   for(i = 0;i < MAX;i++) {
      printf("%d ", intArray[i]);
   }

   printf("]\n");
}

void selectionSort() {
   int indexMin,i,j;

   // loop through all numbers
```

```
for(i = 0; i < MAX-1; i++) {

      // set current element as minimum
      indexMin = i;

      // check the element to be minimum
      for(j = i+1;j < MAX;j++) {
        if(intArray[j] < intArray[indexMin]) {
          indexMin = j;
        }
      }

      if(indexMin != i) {
        printf("Items swapped: [ %d, %d ]\n" , intArray[i], intArray[indexMin]);

        // swap the numbers
        int temp = intArray[indexMin];
        intArray[indexMin] = intArray[i];
        intArray[i] = temp;
      }
printf("Iteration %d#:",(i+1));
      display();
   }
}
void main() {
   printf("Input Array: ");
   display();
   printline(50);
   selectionSort();
   printf("Output Array: ");
   display();
   printline(50);
}
```

Output

```
Input Array: [4 6 3 2 1 9 7 ]
==================================================
Items swapped: [ 4, 1 ]
Iteration 1#:[1 6 3 2 4 9 7 ]
Items swapped: [ 6, 2 ]
Iteration 2#:[1 2 3 6 4 9 7 ]
Iteration 3#:[1 2 3 6 4 9 7 ]
Items swapped: [ 6, 4 ]
Iteration 4#:[1 2 3 4 6 9 7 ]
Iteration 5#:[1 2 3 4 6 9 7 ]
```

Items swapped: [9, 7]
Iteration 6#:[1 2 3 4 6 7 9]
Output Array: [1 2 3 4 6 7 9]

Program-14

WAP to demonstrate working of insertion sort

Algorithms

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

Implementation

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 7
int intArray[MAX] = {4,6,3,2,1,9,7};
void printline(int count) {
   int i;
      for(i = 0;i < count-1;i++) {
      printf("=");
   }
      printf("=\n");
}

void display() {
   int i;
   printf("[");
      // navigate through all items
   for(i = 0;i < MAX;i++) {
      printf("%d ",intArray[i]);
   }
      printf("]\n");
}

void insertionSort() {
   int valueToInsert;
   int holePosition;
   int i;

   // loop through all numbers
   for(i = 1; i < MAX; i++) {
```

```
// select a value to be inserted.
    valueToInsert = intArray[i];

    // select the hole position where number is to be inserted
    holePosition = i;

    // check if previous no. is larger than value to be inserted
    while (holePosition > 0 && intArray[holePosition-1] > valueToInsert) {
      intArray[holePosition] = intArray[holePosition-1];
      holePosition--;
      printf(" item moved : %d\n" , intArray[holePosition]);
    }

    if(holePosition != i) {
      printf(" item inserted : %d, at position : %d\n" , valueToInsert,holePosition);
      // insert the number at hole position
      intArray[holePosition] = valueToInsert;
    }

    printf("Iteration %d#:",i);
    display();
     }
}
void main() {
  printf("Input Array: ");
  display();
  printline(50);
  insertionSort();
  printf("Output Array: ");
  display();
  printline(50);
}
```

Output

```
Input Array: [4 6 3 2 1 9 7 ]
==================================================
Iteration 1#:[4 6 3 2 1 9 7 ]
 item moved : 6
 item moved : 4
 item inserted : 3, at position : 0
Iteration 2#:[3 4 6 2 1 9 7 ]
 item moved : 6
 item moved : 4
 item moved : 3
```

```
 item inserted : 2, at position : 0
Iteration 3#:[2 3 4 6 1 9 7 ]
 item moved : 6
 item moved : 4
 item moved : 3
 item moved : 2
 item inserted : 1, at position : 0
Iteration 4#:[1 2 3 4 6 9 7 ]
Iteration 5#:[1 2 3 4 6 9 7 ]
 item moved : 9
 item inserted : 7, at position : 5
Iteration 6#:[1 2 3 4 6 7 9 ]
Output Array: [1 2 3 4 6 7 9 ]
==================================================
```

Program -15

WAP to demonstrate working Quick sort

Algorithms

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot

Implementation

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 7
int intArray[MAX] = {4,6,3,2,1,9,7};
void printline(int count) {
   int i;
      for(i = 0;i < count-1;i++) {
      printf("=");
   }

   printf("=\n");
}

void display() {
   int i;
   printf("[");

   // navigate through all items
   for(i = 0;i < MAX;i++) {
      printf("%d ",intArray[i]);
   }
      printf("]\n");
}
void swap(int num1, int num2) {
   int temp = intArray[num1];
   intArray[num1] = intArray[num2];
   intArray[num2] = temp;
}
```

```
int partition(int left, int right, int pivot) {
  int leftPointer = left -1;
  int rightPointer = right;

  while(true) {
    while(intArray[++leftPointer] < pivot) {
      //do nothing
    }

    while(rightPointer > 0 && intArray[--rightPointer] > pivot) {
      //do nothing
    }

    if(leftPointer >= rightPointer) {
      break;
    } else {
      printf(" item swapped :%d,%d\n", intArray[leftPointer],intArray[rightPointer]);
      swap(leftPointer,rightPointer);
    }
  }

  printf(" pivot swapped :%d,%d\n", intArray[leftPointer],intArray[right]);
  swap(leftPointer,right);
  printf("Updated Array: ");
  display();
  return leftPointer;
}

void quickSort(int left, int right) {
  if(right-left <= 0) {
    return;
  } else {
    int pivot = intArray[right];
    int partitionPoint = partition(left, right, pivot);
    quickSort(left,partitionPoint-1);
    quickSort(partitionPoint+1,right);
  }
}

int main() {
  printf("Input Array: ");
  display();
  printline(50);
  quickSort(0,MAX-1);
  printf("Output Array: ");
```

```
display();
   printline(50);
}
```

Output

```
Input Array: [4 6 3 2 1 9 7 ]
==================================================
 pivot swapped :9,7
Updated Array: [4 6 3 2 1 7 9 ]
 pivot swapped :4,1
Updated Array: [1 6 3 2 4 7 9 ]
 item swapped :6,2
 pivot swapped :6,4
Updated Array: [1 2 3 4 6 7 9 ]
 pivot swapped :3,3
Updated Array: [1 2 3 4 6 7 9 ]
Output Array: [1 2 3 4 6 7 9 ]
==================================================
```

Program -16

WAP to demonstrate working of Merge sort

Algorithms

Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted order.

Implementation

```
#include <stdio.h>
#define max 10
int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };
int b[10];
void merging(int low, int mid, int high) {
  int l1, l2, i;
  for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
    if(a[l1] <= a[l2])
      b[i] = a[l1++];
    else
      b[i] = a[l2++];
  }

  while(l1 <= mid)
    b[i++] = a[l1++];
  while(l2 <= high)
    b[i++] = a[l2++];
  for(i = low; i <= high; i++)
    a[i] = b[i];
}

void sort(int low, int high) {
  int mid;

  if(low < high) {
    mid = (low + high) / 2;
    sort(low, mid);
    sort(mid+1, high);
    merging(low, mid, high);
  } else {
    return;
  }
}
```

```
int main() {
   int i;

   printf("List before sorting\n");

   for(i = 0; i <= max; i++)
      printf("%d ", a[i]);

   sort(0, max);

   printf("\nList after sorting\n");

   for(i = 0; i <= max; i++)
      printf("%d ", a[i]);
}
```

Output

```
List before sorting
10 14 19 26 27 31 33 35 42 44 0
List after sorting
0 10 14 19 26 27 31 33 35 42 44
```