LAB MANUAL
for
DATA STRUCTURE USING C

4th Semester
Diploma in Computer Engineering

Prepared By Rinki Bhati
Department of Computer Science
Raja Jait Singh Govt. Polytechnic, Neemka Faridabad

GENERAL INSTRUCTIONS

1) Students are instructed to bring their Lab Record to the Lab.
2) Students should come to the Lab in time.
3) Students should be in a proper Uniform.
4) It is mandatory to enter your name in Log-in-Register.
5) Headphones should not be used for any other purpose except for listening to the software.
6) Students are not allowed into the lab without I.D. Cards.
7) After completion of any experiment/activity the student must record it in Lab record and get it signed by the faculty-in-charge.
8) Use of mobile phones during lab hours is strictly prohibited.
9) All students should actively participate in the lab activities.
10) You will not be allowed to copy any software in any format.
11) Marks will be awarded on the basis of the performance in each experiments

OBJECTIVES

1. To introduce students to the basic knowledge of programming fundamentals of C language.
2. To impart writing skill of C programming to the students and solving problems.
3. To impart the concepts like looping, array, functions, pointers, structure.

COURSE OUTCOME

After completing this lab course you will be able to:

- Identify the problem and formulate an algorithm for it.
- Identify the best data structures to solve the problem
- Store data, process data using appropriate data structures
- Sort the data in ascending or descending order.

- Implement trees and various traversing techniques.
- Implement various searching and sorting algorithms and to compare them for checking efficiency

S.NO	PRACTICAL	DATE	SIGN	PAGE	REMARK
1	Sorting of an array				
2	The addition of two matrices using functions.				
3	The multiplication of two matrices.				
4	Push and pop operation in stack				
5	Inserting and deleting elements in queue..				
6	Inserting and deleting elements in circular queue .				
7	Insertion and deletion of elements in linked list.				
8	Insertion and deletion of elements in double linked list.				
9	The factorial of a given number with recursion and without recursion				
10	Fibonacci series with recursion and without recursion.				
11	Program for binary search tree operation.				
12	The selection sort technique.				
13	The bubble sort technique.				
14	The quick sort technique				
15	The merge sort technique.				
16	The binary search procedure to search an element in a given list.				
17	The linear search procedures to search an element in a given list				

INDEX

ABOUT DATA STRUCTURE :➔ Data Structure is a way to store and organize data so that it can be used efficiently. The data structure name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory

Hardware Requirement: Desktop Computer / laptop computer.

Software Requirement: Linux Operating System with GCC / TURBO C in WINDOWS OS / TURBO C++ in WINDOWS OS.

GCC

GCC is a Linux-based C compiler released by the Free Software Foundation which is usually operated via the command line. It often comes distributed freely with a Linux installation, so if you are running UNIX or a Linux variant you will probably have it on your system. You can invoke GCC on a source code file simply by typing:- gcc filename

The default executable output of GCC is "a.out", which can be run by typing "./a.out". It is also possible to specify a name for the executable file at the command line by using the syntax " -o outputfile" , as shown in the following example : -
gcc filename -o outputfile
Again, you can run your program with "./outputfile". (The ./ is there to ensure you run the program for the current working directory.)

Note: If you need to use functions from the math library (generally functions from "math.h" such as sin or sqrt), then you need to explicitly ask it to link with that library with the " –l " flag and the library "m": gcc filename -o outputfile -lm Lab Manual for Programming in C Lab by Er. Suraj Deb Barma

Turbo C/C++

Open Turbo C/C++ from your Desktop or Programs menu. Select "File" from Menu bar and select option "New" and Save C program with filename „.C" extension. To do compiling – Select -> Compile from menu and click-> compile. If the compilation is successful – you will see a "success" message. Else you will see the number of errors. To RUN the program – you may select ->Run from menu and click -> Run Now you will see the output screen.

STRUCTURE OF „C" PROGRAM :

C program is a collection of several instructions where each instruction is written as a separate statement. The C program starts with a main function followed by the opening braces which indicates the start of the function. Then follows the variable and constant declarations which are followed by the statements that include input and output statements. C program may

contain one or more sections as shown below:

INSTRUCTIONS TO STUDENTS FOR PREPARING A LAB REPORT

This Lab Manual is prepared to help the students with their practical understanding and development of programming skills, and may be used as a base reference during the lab/practical classes.

Students have to submit Lab Exercise report of previous lab into corresponding next lab, and can be collected back after the instructor/course co-ordinator after it has been checked and signed. At the end of the semester, students should compile all the Lab Exercise reports into a single report and submit during the end semester sessional examination.

"Sample of Lab report" is shown for LAB Exercise #1 in this manual. For the rest of the labs, the reporting style as provided is to be followed.

The lab report to be submitted during the end semester Sessional Examination should include at least the following topics:-

1. Top Cover page
2. Index
3. Title of the program
5. Algorithm
6. Flowchart (optional)
7. Coding
8. Output (compilation, debugging & testing)

Practical 1

Shorting an Array

Original Array: 5 2 8 7 1

Array after shorting: 1 2 5 7 8

Elements will be short in such a way that smallest element will appear on extreme left which in this case 1. The largest element will appear on extreme right which in class case is 8.

ALGORITHM:

- **STEP 1:** Start
- **STEP 2:** initialize arr[] = {5,2,8,7,1}..
- **STEP 3:** Set temp = 0
- **STEP 4:** length= sizeof9arr)/sizeof(arr[0])
- **STEP 5:** Print "Element of Original Array"
- **STEP 6:** Set i=0. Repeat STEP 7 and STEP 8 until i<length
- **STEP 7:** Print arr[i]
- **STEP 8:** i=i+1.
- **STEP 9:** Set i=0, Repeat STEP 10 until i<n
- **STEP 10:** Set j=i+1. Repeat STEP 11 until j<length
- **STEP 11:** if (arr[i]>arr[j]) then
  then = arr[i]
  arr[i]=arr[j]
  arr[j]=temp
- **STEP 12:** j=j+1.
- **STEP 13:** i=i+1.
- **STEP 14:** print new line
- **STEP 15:** Print "Element of array sorted in ascending order"
- **STEP 16:** Set i=0. Repeat STEP 17 and STEP 18 until i<length
- **STEP 17:** print arr[i]
- **STEP 18:** i=i+1.
- **STEP 19:** Return 0.
- **STEP 20:** END.

PROGRAM:

```c
#include <stdio.h>

int main()
{
    //Initialize array
    int arr[] = {5, 2, 8, 7, 1};
    int temp = 0;

    //Calculate length of array arr
    int length = sizeof(arr)/sizeof(arr[0]);

    //Displaying elements of original array
    printf("Elements of original array: \n");
    for (int i = 0; i < length; i++) {
        printf("%d ", arr[i]);
    }

    //Sort the array in ascending order
    for (int i = 0; i < length; i++) {
        for (int j = i+1; j < length; j++) {
           if(arr[i] > arr[j]) {
               temp = arr[i];
               arr[i] = arr[j];
               arr[j] = temp;
           }
        }
    }

    printf("\n");

    //Displaying elements of array after sorting
    printf("Elements of array sorted in ascending order: \n");
    for (int i = 0; i < length; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Practical 2

1	0	2
0	3	0
4	0	5

Matrix 1

1	0	2
0	3	0
4	0	2

Matrix 2

2	0	4
0	6	0
8	0	18

Added Matric

ALGORITHM:

- **STEP 1:** Input matrix 1 and matrix 2.
- **STEP 2:** If the number of rows and number of columns of matrix 1 and matrix 2 is equal,
- **STEP 3:** for i = 1 to rows [matrix 1]
- **STEP 4:** for j = 1 to columns matrix [1]
- **STEP 5:** input matrix 1 [i,j]
- **STEP 6:** input matrix 2 [i,j]
- **STEP 7:** matrix 3 [i,j] = matrix 1 [i,j] + matrix 2 [i,j];

- **STEP 8:** Display matrix 3 [i,j];

PROGRAM:

```c
#include <stdio.h>
int main()
{
    int r, c, mat1[100][100], mat2[100][100], sum[100][100], i, j;
    printf("Enter the number of rows and columns :\n");
    scanf("%d %d", &r, &c);
    printf("Input Matrix 1 elements :\n");
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j)
        {
            scanf("%d", &mat1[i][j]);
        }
    printf("\n Matrix 1\t\n");
    for (i = 0; i < r; i++)
    {
        for (j = 0; j < c; j++)
        {
            printf("%d", mat1[i][j]);
        }
        printf("\n");
    }
    printf("Input Matrix 2 elements :\n");
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j)
        {
            scanf("%d", &mat2[i][j]);
        }
    printf("\n Matrix 2\n");
    for (i = 0; i < r; i++)
    {
        for (j = 0; j < c; j++)
        {
            printf("%d", mat1[i][j]);
        }
        printf("\n");
    }
    // Adding Two matrices
    printf("\nAdded Matrix\n");
```

Practical 3

3	4	0	7	3*5 + 4*0	3*6 + 4*7	15	46

ALGORITHM:

- **STEP 1:** Start the Program.
- **STEP 2:** Enter the row and column of the first (a) matrix.
- **STEP 3:** Enter the elements of first (a) matrix.
- **STEP 4:** Enter the row and column of the second (b) matrix.
- **STEP 5:** Enter the elements of second (b) matrix.
- **STEP 6:** Set a loop up to row
- **STEP 7:** Set an inner loop up to the column.
- **STEP 8:** Set another loop up to the column.
- **STEP 9:** Multiply the first (a) and second (b) matrix and store the element in the third matrix (c).
- **STEP 10:** Print the final matrix.
- **STEP 11:** Stop the Program.

PROGRAM:

Lab

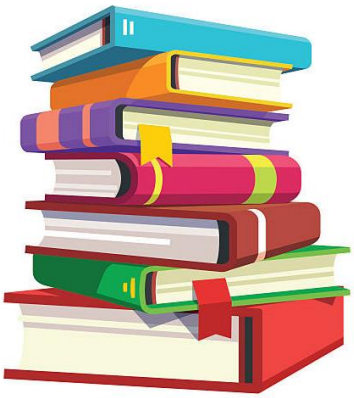
```c
#include <stdio.h>

int main()
{
  int m, n, p, q, c, d, k, sum = 0;
  int first[10][10], second[10][10], multiply[10][10];

  printf("Enter number of rows and columns of first matrix\n");
  scanf("%d%d", &m, &n);
  printf("Enter elements of first matrix\n");
```

Practical 4

Push and pop operation in stack

Stack is an Abstract Data Type (ADT) and it is a linear data structure where insertion or deletion of elements is done from only one side/end which is called top of the stack.
Some Applications of Stack are following:

- ✓ Used in function calls
- ✓ Infix to postfix conversion (and other similar conversions)
- ✓ Parenthesis matching and more…

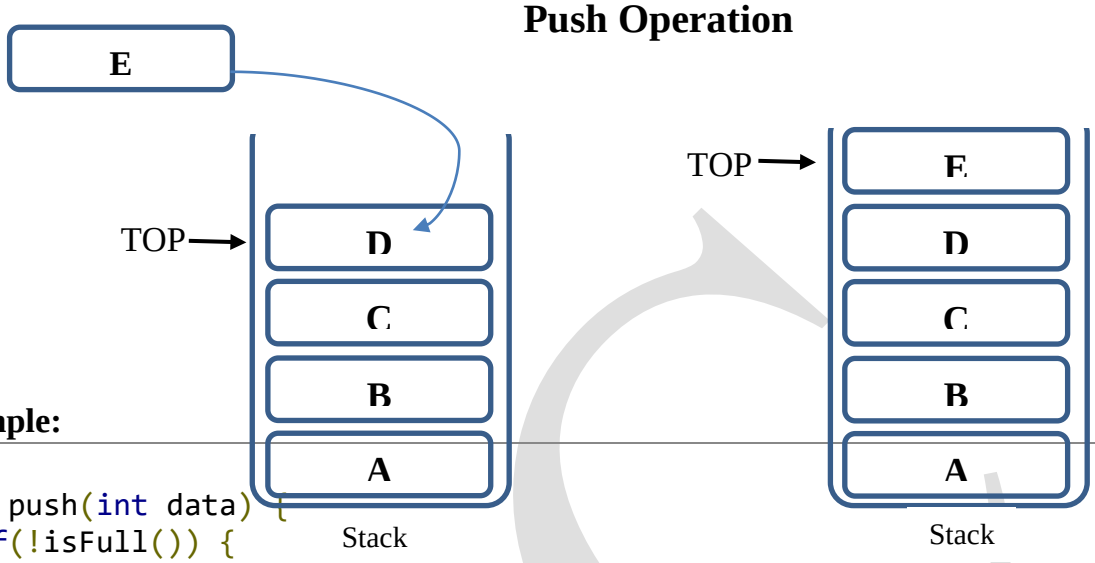


Push Operation: The process of pitting a new data element onto stack is known as Push Operation. Push operation involves a series of steps –

- ➲ Step 1 – Checks if 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.

Push Operation



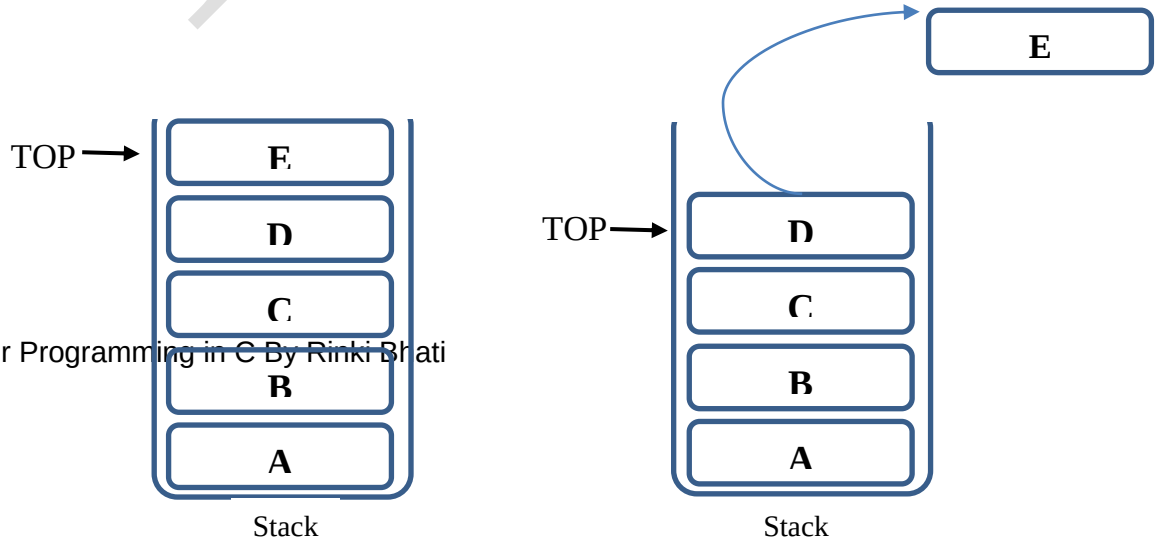
Example:

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

Pop Operation: Accessing the content while removing it from the tack is known as Pop Operation. A Pop operation may involve the following steps –

- ➲ Step 1 – Checks if 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** to pointing.
- ➲ Step 4 – Decreases the value of top by 1.
- ➲ Step 5 – Returns success.

Pop Operation



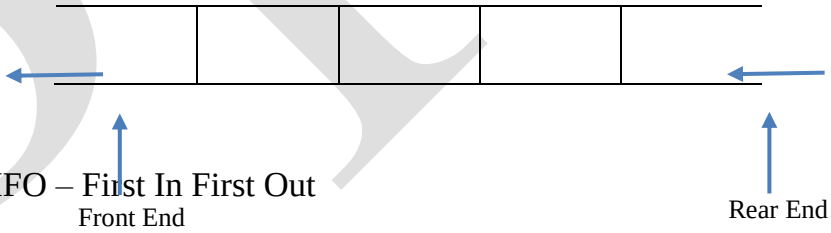
Example:

```
int pop(int data) {

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

Practical 5

Inserting and Deleting Elements in Queue

Queue: is a linear data structure, where the insertion is done at rear end and the deletion is done at the front end.



The order of queue is FIFO – First In First Out

Operations:

- Insert – Inserting an element into a queue.
- Delete – Deleting an element from the queue.

Conditions:

- Queue over Flow – Tring to insert an element into a full queue.
- Queue under Flow – Trying to delete an element from an empty queue.

Program:

```
#include <stdio.h>
#define MAX 50
void insert();
int array[MAX];
int rear = - 1;
int front = - 1;
```

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

```
   else{
      printf("Queue is : \n");
      for (i = front; i <= rear; i++)
         printf("%d ", array[i]);
         printf("\n");
   }
}
void delete(){
   if (front == - 1 || front > rear){
      printf("Queue Underflow \n");
      return ;
   }
   else{
      printf("Element deleted from queue is : %d\n",array[front]);
      front = front + 1;
   }
}
```

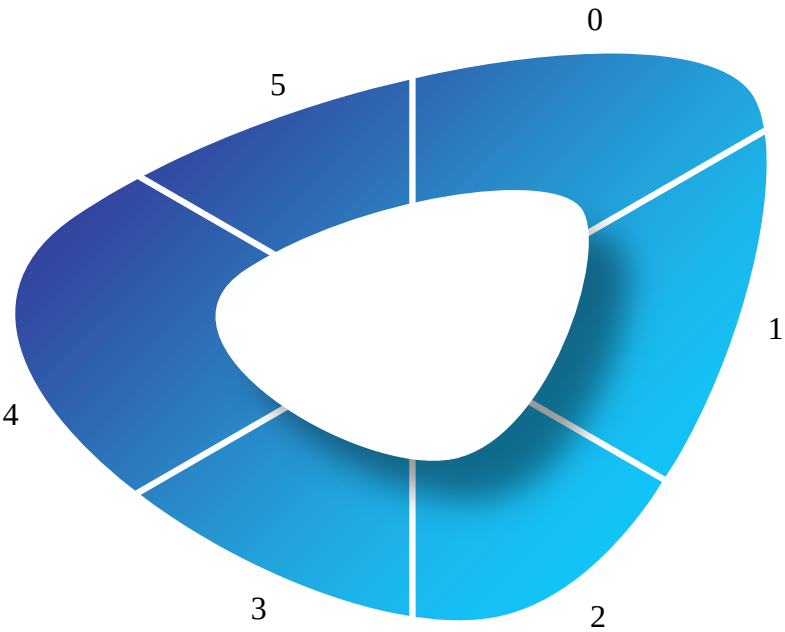
Output:

```
1.Insert element to queue
2.Delete an element from queue
3.Display elements of queue
4.Quit
Enter your choice: 1
Inset the element in queue: 12
1.Insert element to queue
2.Delete an element from queue
3.Display elements of queue
4.Quit
Enter your choice: 1
Inset the element in queue: 23
1.Insert element to queue
2.Delete an element from queue
```

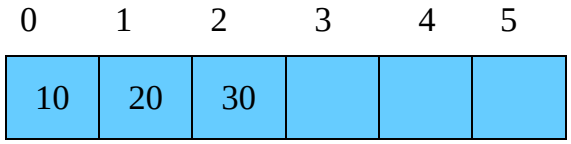
```
3.Display elements of queue
4.Quit
Enter your choice: 1
Inset the element in queue: 34
1.Insert element to queue
2.Delete an element from queue
3.Display elements of queue
4.Quit
Enter your choice: 2
Element deleted from queue is: 12
Queue is:
23 34
1.Insert element to queue
2.Delete an element from queue
3.Display elements of queue
4.Quit
Enter your choice: 2
Element deleted from queue is: 23
Queue is:
34
1.Insert element to queue
2.Delete an element from queue
3.Display elements of queue
4.Quit
Enter your choice: 4
```


Practical 6

Inserting and Deleting Elements in Circular Queue



Circular Queue is a liner data structure in which the operation are performed based on (FIFO First in First Out) principle and the las position is connected back to the first position to make a circle. It is also called '**Ring Buffer**'



Insertion (Queue, Key) –

```
begin
   if front = 0 and rear = n - 1, or front = rear + 1, then queue is
full, and return
   otherwise
   if front = -1, then front = 0 and rear = 0
   else
      if rear = n - 1, then, rear = 0, else rear := rear + 1
   queue[rear] = key
end
```

Delete (Queue) –

```
begin
   if front = -1 then queue is empty, and return
   otherwise
   item := queue[front]
   if front = rear, then front and rear will be -1
   else
      if front = n - 1, then front := 0 else front := front + 1
end
```

Program –

```
#include <iostream>
using namespace std;
int cqueue[5];
int front = -1, rear = -1, n=5;
void insertCQ(int val) {
   if ((front == 0 && rear == n-1) || (front == rear+1)) {
      cout<<"Queue Overflow \n";
      return;
   }
   if (front == -1) {
      front = 0;
      rear = 0;
   }
   else {
      if (rear == n - 1)
         rear = 0;
      else
         rear = rear + 1;
   }
   cqueue[rear] = val ;
}
```

```
void deleteCQ() {
   if (front == -1) {
      cout<<"Queue Underflow\n";
      return ;
   }
   cout<<"Element deleted from queue is : "<<cqueue[front]<<endl;
   if (front == rear) {
      front = -1;
      rear = -1;
   }
   else {
      if (front == n - 1)
         front = 0;
      else
         front = front + 1;
   }
}
void displayCQ() {
   int f = front, r = rear;
   if (front == -1) {
      cout<<"Queue is empty"<<endl;
      return;
   }
   cout<<"Queue elements are :\n";
   if (f <= r) {
      while (f <= r){
         cout<<cqueue[f]<<" ";
         f++;
      }
   }
   else {
      while (f <= n - 1) {
         cout<<cqueue[f]<<" ";
         f++;
      }
      f = 0;
      while (f <= r) {
         cout<<cqueue[f]<<" ";
         f++;
      }
   }
   cout<<endl;
}
int main() {
   int ch, val;
   cout<<"1)Insert\n";
   cout<<"2)Delete\n";
   cout<<"3)Display\n";
   cout<<"4)Exit\n";
   do {
```

```
      cout<<"Enter choice : "<<endl;
      cin>>ch;
      switch(ch) {
         case 1:
            cout<<"Input for insertion: "<<endl;
            cin>>val;
            insertCQ(val);
         break;
         case 2:
            deleteCQ();
         break;
         case 3:
            displayCQ();
         break;
         case 4:
            cout<<"Exit\n";
         break;
            default: cout<<"Incorrect!\n";
      }
   } while(ch != 4);
      return 0;
}
```

Output

```
1)Insert
2)Delete
3)Display
4)Exit
Enter choice :
1
Input for insertion:
10
Enter choice :
1
Input for insertion:
20
Enter choice :
1
Input for insertion:
30
Enter choice :
1
Input for insertion:
40
Enter choice :
1
Input for insertion:
50
Enter choice :
```

```
3
Queue elements are :
10 20 30 40 50
Enter choice :
2
Element deleted from queue is : 10
Enter choice :
2
Element deleted from queue is : 20
Enter choice :
3
Queue elements are :
30 40 50
Enter choice :
4
```

Practical 7

Inserting and Deleting Elements in Linked List

- **Insertion** – Adds a new element
- **Deletion** – Removes the existing elements

Important Points:

- head points to the first node of the liked list
- next pointer of the last is NULL, so if the next current node is NULL, we have reached the end of the linked list.

Program:

```c
// Linked list operations in C

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

// Create a node
struct Node {
  int item;
  struct Node* next;
};

void insertAtBeginning(struct Node** ref, int data) {
  // Allocate memory to a node
  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

  // insert the item
  new_node->item = data;
  new_node->next = (*ref);
```

```
  // Move head to new node
  (*ref) = new_node;
}

// Insert a node after a node
void insertAfter(struct Node* node, int data) {
  if (node == NULL) {
    printf("the given previous node cannot be NULL");
    return;
  }

  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
  new_node->item = data;
  new_node->next = node->next;
  node->next = new_node;
}

void insertAtEnd(struct Node** ref, int data) {
  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
  struct Node* last = *ref;

  new_node->item = data;
  new_node->next = NULL;

  if (*ref == NULL) {
    *ref = new_node;
    return;
  }

  while (last->next != NULL)
    last = last->next;

  last->next = new_node;
  return;
}

void deleteNode(struct Node** ref, int key) {
  struct Node *temp = *ref, *prev;

  if (temp != NULL && temp->item == key) {
    *ref = temp->next;
    free(temp);
    return;
  }
  // Find the key to be deleted
  while (temp != NULL && temp->item != key) {
    prev = temp;
    temp = temp->next;
  }

  // If the key is not present
  if (temp == NULL) return;

  // Remove the node
```

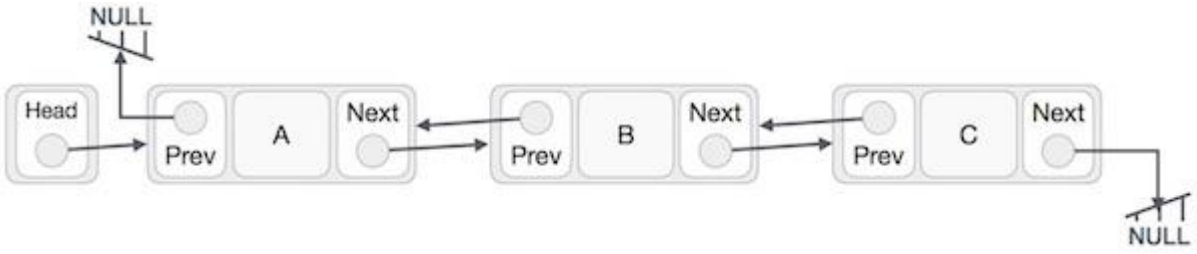
```
  prev->next = temp->next;

  free(temp);
}

// Print the linked list
void printList(struct Node* node) {
  while (node != NULL) {
    printf(" %d ", node->item);
    node = node->next;
  }
}

// Driver program
int main() {
  struct Node* head = NULL;

  insertAtEnd(&head, 1);
  insertAtBeginning(&head, 2);
  insertAtBeginning(&head, 3);
  insertAtEnd(&head, 4);
  insertAfter(head->next, 5);

  printf("Linked list: ");
  printList(head);

  printf("\nAfter deleting an element: ");
  deleteNode(&head, 3);
  printList(head);
}
```

Practical 8

Inserting and Deleting Elements in Doubly Linked List

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List.

Doubly Linked List Representation:



Insertion Operation:

```
//insert link at the first location
void insertFirst(int key, int data) {

   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data = data;

   if(isEmpty()) {
      //make it the last link
      last = link;
   } else {
      //update first prev link
      head->prev = link;
   }

   //point it to old first link
   link->next = head;

   //point first to new first link
   head = link;
}
```

Deletion Operation:

```
//delete first item
struct node* deleteFirst() {

   //save reference to first link
   struct node *tempLink = head;

   //if only one link
   if(head->next == NULL) {
      last = NULL;
   } else {
      head->next->prev = NULL;
   }

   head = head->next;

   //return the deleted link
   return tempLink;
```

Practical 9

The Factorial of a given number with recursion and without recursion

Using Recursion:

```c
#include<stdio.h>
long int multiplyNumbers(int n);
int main() {
    int n;
    printf("Enter a positive integer: ");
    scanf("%d",&n);
    printf("Factorial of %d = %ld", n, multiplyNumbers(n));
    return 0;
}

long int multiplyNumbers(int n) {
    if (n>=1)
        return n*multiplyNumbers(n-1);
    else
        return 1;
}
```

Output:

```
Enter a positive integer: 6
Factorial of 6 = 720
```

Without Recursion:

```c
#include <stdio.h>
int main()
{
        int n, i;
        long fact=1;

        printf(" Enter any number: ");
        scanf("%d", &n);

        for (i=1; i<=n; i++)
        fact = fact*i;
        printf(" Factorial = %ld", fact);

        getch();
}
```

Practical 10

Fibonacii series with recursion and without recursion

- **Without Recursion:**

```c
#include <stdio.h>
int main()
{
    int n1 = 0, n2 = 1, n3, i, number;
    printf("Enter the number of elements:");
    scanf("%d", &number);
    printf("\n%d %d", n1, n2);   //printing 0 and 1
    for (i = 2; i < number; ++i) //loop starts from 2 because 0 and 1 are already
 printed
    {
        n3 = n1 + n2;
        printf(" %d", n3);
        n1 = n2;
        n2 = n3;
    }
    return 0;
}
```

Output:

```
Enter the number of elements:15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

- **With Recursion:**

```c
#include<stdio.h>
void printFibonacci(int n){
    static int n1=0,n2=1,n3;
    if(n>0){
         n3 = n1 + n2;
         n1 = n2;
         n2 = n3;
         printf("%d ",n3);
         printFibonacci(n-1);
    }
}
int main(){
    int n;
    printf("Enter the number of elements: ");
    scanf("%d",&n);
    printf("Fibonacci Series: ");
```

```
    printf("%d %d ",0,1);
    printFibonacci(n-2);//n-2 because 2 numbers are already printed
  return 0;
 }
```

Output:

```
Enter the number of elements:15
8 13 21 34 55 89 144 233 377
```

Practical 11

Program for binary search tree operation

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

The properties that separate a binary search tree from a regular binary tree is

1. All nodes of left subtree are less than the root node
2. All nodes of right subtree are more than the root node
3. Both subtrees of each node are also BSTs i.e. they have the above two properties

```c
// Binary Search Tree operations in C

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

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

// Create a node
struct node *newNode(int item)
{
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Inorder Traversal
```

```c
void inorder(struct node *root)
{
    if (root != NULL)
    {
        // Traverse left
        inorder(root->left);

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

        // Traverse right
        inorder(root->right);
    }
}

// Insert a node
struct node *insert(struct node *node, int key)
{
    // Return a new node if the tree is empty
    if (node == NULL)
        return newNode(key);

    // Traverse to the right place and insert the node
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    return node;
}

// Find the inorder successor
struct node *minValueNode(struct node *node)
{
    struct node *current = node;

    // Find the leftmost leaf
    while (current && current->left != NULL)
        current = current->left;

    return current;
}

// Deleting a node
```

```c
struct node *deleteNode(struct node *root, int key)
{
    // Return if the tree is empty
    if (root == NULL)
        return root;

    // Find the node to be deleted
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    else
    {
        // If the node is with only one child or no child
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }

        // If the node has two children
        struct node *temp = minValueNode(root->right);

        // Place the inorder successor in position of the node to be deleted
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

// Driver code
int main()
{
    struct node *root = NULL;
```

```
    root = insert(root, 8);
    root = insert(root, 3);
    root = insert(root, 1);
    root = insert(root, 6);
    root = insert(root, 7);
    root = insert(root, 10);
    root = insert(root, 14);
    root = insert(root, 4);

    printf("Inorder traversal: ");
    inorder(root);

    printf("\nAfter deleting 10\n");
    root = deleteNode(root, 10);
    printf("Inorder traversal: ");
    inorder(root);
}
```

Practical 12

The Selection Short Technique

```
// Selection sort in C

#include <stdio.h>

// function to swap the the position of two elements
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

void selectionSort(int array[], int size)
{
```

```
    for (int step = 0; step < size - 1; step++)
    {
        int min_idx = step;
        for (int i = step + 1; i < size; i++)
        {

            // To sort in descending order, change > to < in this line.
            // Select the minimum element in each loop.
            if (array[i] < array[min_idx])
                min_idx = i;
        }

        // put min at the correct position
        swap(&array[min_idx], &array[step]);
    }
}

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

// driver code
int main()
{
    int data[] = {20, 12, 10, 15, 2};
    int size = sizeof(data) / sizeof(data[0]);
    selectionSort(data, size);
    printf("Sorted array in Acsending Order:\n");
    printArray(data, size);
}
```

Practical 13

The Bubble Short Technique

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

void swap(int *xp, int *yp)
```

```c
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

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

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

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

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

Practical 14

The Quick Short Technique

Quick sort is a fast sorting algorithm used to sort a list of elements. Quick sort algorithm is invented by **C. A. R.Hoare**.

The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. In quick sort, the partition of the list is performed based on the element called ***pivot***. Here pivot element is one of the elements in the list.

The list is divided into two partitions such that **"all elements to the left of pivot are smaller than the pivot and all elements to the right of pivot are greater than or equal to the pivot"**.

Step by Step Process

In Quick sort algorithm, partitioning of the list is performed using following steps...

- **Step 1 -** Consider the first element of the list as **pivot** (i.e., Element at first position in the list).
- **Step 2 -** Define two variables i and j. Set i and j to first and last elements of the list respectively.
- **Step 3 -** Increment i until list[i] > pivot then stop.
- **Step 4 -** Decrement j until list[j] < pivot then stop.
- **Step 5 -** If i < j then exchange list[i] and list[j].
- **Step 6 -** Repeat steps 3,4 & 5 until i > j.
- **Step 7 -** Exchange the pivot element with list[j] element.

Program:

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

void quickSort(int [10],int,int);

void main(){
  int list[20],size,i;

  printf("Enter size of the list: ");
  scanf("%d",&size);

  printf("Enter %d integer values: ",size);
  for(i = 0; i < size; i++)
    scanf("%d",&list[i]);

  quickSort(list,0,size-1);
```

```
  printf("List after sorting is: ");
  for(i = 0; i < size; i++)
    printf(" %d",list[i]);

  getch();
}

void quickSort(int list[10],int first,int last){
    int pivot,i,j,temp;

     if(first < last){
         pivot = first;
         i = first;
         j = last;

         while(i < j){
             while(list[i] <= list[pivot] && i < last)
                 i++;
             while(list[j] > list[pivot])
                 j--;
             if(i <j){
                  temp = list[i];
                  list[i] = list[j];
                  list[j] = temp;
             }
         }

         temp = list[pivot];
         list[pivot] = list[j];
         list[j] = temp;
         quickSort(list,first,j-1);
         quickSort(list,j+1,last);
    }
}
```

Output:

```
Enter size of the list: 5
Enter 5 integer values: 8 7 12 4 3
List after sorting is:  3 4 7 8 12
```

Practical 15

The merge short technique

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

- **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.

```c
/* C program for Merge Sort */
#include <stdio.h>
#include <stdlib.h>

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

    /* create temp arrays */
```

```
    int L[n1], R[n2];

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

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

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

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

```
}

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

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

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

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

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

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

    mergeSort(arr, 0, arr_size - 1);

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

```
}
```

Output:

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

Practical 16

The binary search procedures to search an element in given list

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

Program

```c
#include <stdio.h>

int binarySearch(int arr[], int size, int element)
{
    int low, mid, high;
    low = 0;
    high = size - 1;
    // Kepp Searching Until Low<= High
    while (low <= high)
    {
```

```
        mid = (low + high) / 2;

        if (arr[mid] == element)
        {
            return mid;
        }

        if (arr[mid] < element)
        {
            low = mid + 1;
        }
        else
        {
            high = mid - 1;
        }
    }
    return -1;
}

int main()
{

    // Sorted Array for Linear Search
    int arr[] = {1, 2, 4, 48, 88, 77, 100, 164, 188, 265, 654, 678, 684, 714, 722
, 850};
    int size = sizeof(arr) / sizeof(int);

    int element = 850;
    int searchIndex = binarySearch(arr, size, element);
    printf("The Element %d was founded at %d \n", element, searchIndex);
    return 0;
}
```

Output:

```
350 was founded at 15
```

Practical 17

The linear search procedures to search an element in a given list

Linear search algorithm finds a given element in a list of elements with **O(n)** time complexity where **n** is total number of elements in the list. This search process starts comparing search element with the first element in the list. If both are matched then result is element found otherwise search element is compared with the next element in the list.

Algorithm

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the first element in the list.

- **Step 3 -** If both are matched, then display "Given element is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then compare search element with the next element in the list.
- **Step 5 -** Repeat steps 3 and 4 until search element is compared with last element in the list.
- **Step 6 -** If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

```c
#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
```

PIC