`

Ramaiah Institute of Technology, Bangalore-560 054

(Autonomous Institute, Affiliated to VTU, Belgaum)

Department of Electronics and Instrumentation Engineering

LAB MANUAL

DATA STRUCTURES USING C LAB

Semester: V

Year: 2024-2025

Sub code: EIL56

Prepared by: Dr. Elavaar Kuzhali.S.

Reviewed by: Dr. M.D.Nandeesh

For private circulation only

Introduction and scope of the course:

Data Structures Using C are used to store data in an organized and efficient manner. The C Programming language has many data structures like an *array, stack, queue, linked list, tree, graph etc.* A programmer selects an appropriate data structure and uses it according to their convenience.

Let us look into some of these data structures:

- Array
- Stack
- Queue
- Linked List
- Trees & Graph
- Searching & Sorting

Data structure is a process through which data is stored and arranged in the disk space of the computer or memory storage, in a way that the data can be easily used and manipulated in the future. It is an effective way of performing various operations related to data management. With a sufficient understanding of data structure data can be organized and stored in a proper manner. Data structures are designed to organize data in order to suit specific purposes so as to access and perform operations in an appropriate manner.

There are basically two main types of data structures:

1. Primitive data structure
2. Abstract data structure

Primitive data structure: The concept to handle, in an efficient way, certain types of data including Boolean & char, float, integer, etc. is called primitive data structure.

Abstract data structure: There are certain complex types of data such as tree, graph, stack & queue, linked list, etc. The concept to handle a large amount of data that are complex and connected is called abstract data structure.

Scope of data structure deals with how big or small your environment is. i.e., There are a lot of programs that still demand something like this, and because storage devices are larger and processes are faster, the constant factors for common code parts are not as important as they were twenty years ago. (Although they still matter a lot.)

As far as we know, in the future, performance considerations are going to be similar to what we have today, but hardware will be slightly faster, programs will talk to more storage devices, and many programs will have more parallel parts.

For Electronics & Instrumentation Engineering students, mastering data structures in C enables them to: Write efficient firmware and embedded code, Optimize memory and processing in resource-constrained environments, Develop advanced algorithms for control, signal processing, and automation, Design and simulate complex systems such as sensor networks, control loops, and embedded systems. So studying data structures helps **you deal with different ways of arranging, processing and storing data**. All codes are made for real time purpose so data structure allow user to provide/use/handle date in different ways.

Currently, heavily-reused code using data structures tends to need these parts:

- **in easy cases**: it usually needs to fall back on large arrays, because processors being manufactured have large caches, meaning that if you load an array and look at all its elements sequentially, you’ll get better performance than if you read a lot of memory locations.

- **in hard cases**: it needs to fall back on data structures based on splitting the data set a lot of times, like binary search trees and heaps.

Data structure and algorithms help in understanding the nature of the problem at a deeper level and thereby a better understanding of the world.

Real-life examples of common data structures and algorithms.

1. Arrays: It is the most used data structure. An array is simply a collection of objects or things. Simple examples can be a collection of books on your table, clothes in a wardrobe and pens in your pen stand. In software programs, the array is used almost everywhere in the program to store any data.

2. Queue: Queue works as first in first out manner. If you ever stand in line whether for getting into a bus or collecting movie tickets where the first one in the line processed first, then you experienced this common data structure. It is used in Transport and operations research where various entities are stored and held to be processed later. In operating systems, it is used for the waiting list in Applications and task scheduling.

3. Stack: Look at the pile of dishes in your kitchen rack. It resembles the stack data structure where the object on the top is taken first. It is used for undo/redo operation in word processors, mathematical expression evaluation, and syntax parsing in any software program.

4. Graph: Think of your friends on facebook. They are connected to you and to others in the form of a graph. You'll also find the application of graph in google maps where the shortest distance from any location is calculated using graph algorithms. In computers, it is used for routing, data organizations and in networks of communication.

5. Tree: A tree data structure can be thought of like the hierarchy of employees in an office. You'll find trees used in many applications. It is used in parsers, file systems, IP routing table, data analysis, and data mining applications.

Semester: V Year: 2024-25

Department:*Electronics & Instrumentation Engineering*	***Course Type:*** *Core*
Course Title: *Data Structures using C Lab*	***Course Code:*** *EIL56*
L-T-P:*0-0-1*	***Credits: 1***
Total Contact Hours:*14 hours*	***Duration of SEE:*** *3 hours*
SEE Marks*: 50*	***CIE Marks:*** *50*

Pre-requisites:

i. Knowledge on Programming Languages.

Course Outcomes:

Students will be able to:

Cos	**Course Outcome Description**	**Blooms Level**
1	Develop programming solutions for real time applications using stack, Queues and linked list	L3
2	Design a hierarchical based programming solution using different tree traversal techniques and graph theory.	L3
3	Apply the knowledge of sorting & searching algorithms for problem solving.	L3

Teaching Methodology:

i. Laboratory instruction classes

ii. Laboratory onsite interaction

Assessment Methods:

- Rubrics for continuous evaluation of laboratory experiments for 30 marks.
- Two Continuous Internal Evaluations (CIEs) for 20 Marks each will be conducted and average will be considered.
- Semester End Examination (SEE) for 50 Marks will be conducted.

Course Outcome to Programme Outcome Mapping:

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	3	3	3		3				2	3			2		3
CO2	3	3	3		3				2	3			2		3
CO3	3	3	3		3				2	3			2		3
EIL 56	3	3	3		3				2	3			2		3

COURSE CONTENT

Program No.	Program Title	Duration
1.	Illustrating arrays, functions for data operations a) Write a C Program to Find 2 Elements in the Array such that Difference between them is Largest b) Write a C Program to calculate the largest two numbers in a given Array c) Write a C Program to perform insert, delete operations on a given Array	02 Hr
2.	Illustrating pointers for data operations, Examining Dynamic memory allocations a) Write a C Program to perform swapping of two numbers using pass by reference b) Write a C Program to concatenate two strings using pointers	02 Hr

	without using built-in string functions c) Write a C Program to calculate the sum of n numbers entered by the user, Dynamically allocate memory to store the n numbers	
3.	Managing Structures in applications a) Create a structure named Complex to represent a complex number with real and imaginary parts. Write a C program to add and multiply two complex numbers b) Write a C Program to create a new structure called Student that contains the following fields: Name (a string of maximum 20 characters), Roll (an integer), and avg_marks (a float). Then, write a main function defines a student structure, initializes a student instance S1, dynamically allocates memory for another student instance S2, and uses pointers to access and print values for both S1 and S2 c) Create a structure named Book to store book details like title, author, and price. Write a C program to input details for three books, find the most expensive and the lowest priced books, and display their information	02 Hr
4.	a) Write a program to illustrate forward and backward surfing in the web browser using stack (Array implementation). Display the appropriate messages in case of exceptions b) Write a program to convert and print a given valid parenthesized infix arithmetic expression to postfix expression. The expression consists of single character operands and binary operators + - * /. Apply the concept of stack data structure to solve this problem. c) A call center phone system has to hold the phone calls from customers and provide service based on the arrival time of the calls. Write a C program to simulate this system using queue data structure. Program should have options to add and remove the phone calls in appropriate order for their service	02 Hr
5.	Write a program to illustrate memory allocation to files based on their size using singly linked list. Program must support the following operations on a singly linked list. a) Program to add and delete nodes at the beginning of the Singly linked list b) Program to add and delete nodes at the end of the Singly linked list	02 Hr
6.	Write a program to illustrate memory allocation to files based on their size using singly linked list. Program must support the following operations on a singly linked list. Program to create and reverse a Singly linked list and display the same Program to search a given element in a given Singly linked list	02 Hr
7.	Write a program to manage integer data using a **doubly linked list**. The program should allow users to perform the following operations:	02 Hr

	1. **Insert at the beginning**: Add an integer value at the beginning of the doubly linked list and **Delete at the beginning**: Remove a node at the beginning of the doubly linked list. 2. **Insert at the end**: Add an integer value at the end of the doubly linked list And **Delete at the end**: Remove a node at the end of the doubly linked list.	
8.	a) Implement a stack using a singly linked list in C. Design functions for push (insert), pop (delete), and display operations. Write a program to simulate the backtracking functionality in a web browser, where the user can visit new web pages (push) or go back to the previous page (pop). Test your program with at least five pages, displaying the stack after each operation. b) Implement a queue using a singly linked list in C. Design functions for enqueue (insert), dequeue (delete), and display operations. Write a program to simulate the customer service system at a bank where customers join the queue for service. Test your program with at least five customers, displaying the queue after each operation.	02 Hr
9.	Write a program to demonstrate round robin processor scheduling using circular queue (array implementation) with suitable inputs. Program should have options to add, remove and display elements of the queue.	02 Hr
10.	a. Construct a binary tree, Write programs for in order, preorder and post order traversal of trees. b. Program to create, traverse and search in a binary search tree	02 Hr
11.	a. Write a program to implement a linear search algorithm to search a specific record in a database containing student roll numbers. b. Write a program to implement the binary search algorithm to efficiently search a sorted database of product IDs in an inventory system.	02 Hr
12.	Program to sort a data base using Bubble sort Program to sort a data base using Quick sort	02 Hr
13.	**Write a C program that implements Depth First Search (DFS) for an undirected graph using an adjacency matrix and a stack.**	02 Hr
14.	**Write a C program that implements Breadth First Search (BFS) for an undirected graph using an adjacency matrix and a stack.**	02 Hr

Experiment 1:

a) **Write a C Program to Find 2 Elements in the Array such that Difference between them is Largest**

Problem Solution

1. **Input:**
    - An array of integers array[] of size arr_size.
2. **Initialize:**
    - Set max_diff = array[1] - array[0] to store the maximum difference between two elements.
    - Declare two loop variables i and j.
3. **Iterate through the Array:**
    - Start an outer loop with i from 0 to arr_size - 1.
        - For each element array[i], start an inner loop with j from i + 1 to arr_size - 1.
            - Compute the difference between array[j] and array[i].
            - If this difference is greater than max_diff, update max_diff with the new difference.
4. **Return the Maximum Difference:**
    - After all iterations, return max_diff as the largest difference between two elements in the array.
5. **Output the Result:**
    - In the main function, define the array of integers.
    - Call the maximum_difference function with the array and its size.
    - Print the result showing the largest difference between two elements.

Code Flow Example:

For the array {10, 15, 90, 200, 110}:

- Differences between all pairs of elements are calculated.
- The largest difference found is 200 - 10 = 190.

Thus, the maximum difference is 190.

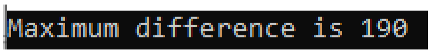
```
#include <stdio.h>
```

```
#include <stdlib.h>
int maximum_difference(int array[], int arr_size)
{
   int max_diff = array[1] - array[0];
   int i, j;
   for (i = 0; i < arr_size; i++)
   {
      for (j = i + 1; j < arr_size; j++)
      {
         if (array[j] - array[i] > max_diff)
            max_diff = array[j] - array[i];
      }
   }
   return max_diff;
}

int main()
{
   int array[ ] = {10, 15, 90, 200, 110};
   printf("Maximum difference is %d",  maximum_difference(array, 5));
   return 0;
}
```

Time Complexity:

- The algorithm uses two nested loops, resulting in a time complexity of **O(n^2)**, where n is the number of elements in the array.

Output:



```
Maximum difference is 190
```

b) **Write a C Program to calculate the largest two numbers in a given Array**

Expected Input and Output

1. Finding Largest 2 numbers in an array with unique elements:

If we are entering 5 elements (N = 5), with array element values as 2,4,5,8 and 7 then,

The FIRST LARGEST = 8

THE SECOND LARGEST = 7

2. Finding Largest 2 numbers in an array with recurring elements:

If we are entering 6 elements (N = 6), with array element values as 2,1,1,2,1 and 2 then,

The FIRST LARGEST = 2

THE SECOND LARGEST = 1

Problem Solution

In this program, we have to find the largest and second-largest elements present in the array. We will do this by first saving the values of the first element in the variable **'largest'** and second element in the variable **'second-largest'**. Then we will start from the third element and compare and swap with **'largest'** and **'second-largest'** numbers if another larger number is found in this array. This will go on N-2 times and the program ends.

1. **Input the Size of the Array:**
   - Prompt the user to enter the size of the array n.
   - Create an integer array array[n] of size n.
2. **Input Array Elements:**
   - Prompt the user to input the elements of the array.
   - Store the elements in the array.
3. **Display Array Elements:**
   - Print all the elements of the array.
4. **Initialize Largest Values:**
   - Initialize largest1 and largest2 to store the two largest numbers.
     - Set largest1 to the first element (array[0]).
     - Set largest2 to the second element (array[1]).

   - If largest1 is smaller than largest2, swap them using a temporary variable temp.

5. **Find the Two Largest Elements:**
   - Loop through the array starting from the 3rd element (i = 2 to n-1):
     - If the current element is greater than largest1:
       - Assign largest2 the value of largest1.
       - Assign largest1 the value of the current element.
     - Else, if the current element is greater than largest2 and not equal to largest1, assign the current element to largest2.
6. **Output the Results:**
   - Print largest1 as the first largest element.
   - Print largest2 as the second largest element.

Example Walkthrough:

- For the input array {10, 5, 8, 20, 15}:
  - Initial values: largest1 = 10, largest2 = 5
  - After iterating through the array:
    - largest1 = 20, largest2 = 15
  - Output: First largest = 20, Second largest = 15.

Time Complexity:

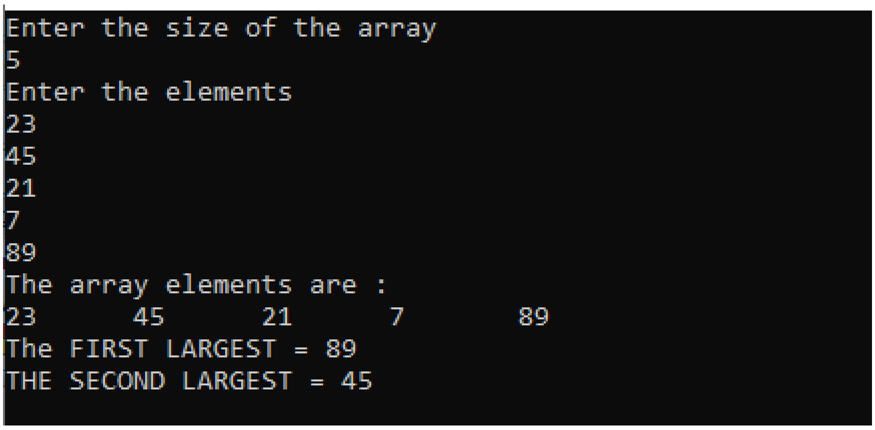
- **O(n)** where n is the number of elements in the array. The algorithm requires only one pass through the array to find the two largest elements.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
  int n = 0, i = 0, largest1 = 0, largest2 = 0, temp = 0;

  printf ("Enter the size of the array\n");
  scanf ("%d", &n);
  int array[n];
```

```
printf ("Enter the elements\n");
for (i = 0; i < n; i++)
{
    scanf ("%d", &array[i]);
}

printf ("The array elements are : \n");
for (i = 0; i < n; i++)
{
    printf ("%d\t", array[i]);
}

printf ("\n");

largest1 = array[0];
largest2 = array[1];

if (largest1 < largest2)
{
    temp = largest1;
    largest1 = largest2;
    largest2 = temp;
}

for (int i = 2; i < n; i++)
{
    if (array[i] > largest1)
    {
        largest2 = largest1;
        largest1 = array[i];
    }
```

```
        else if (array[i] > largest2 && array[i] != largest1)
        {
            largest2 = array[i];
        }
    }

    printf ("The FIRST LARGEST = %d\n", largest1);
    printf ("THE SECOND LARGEST = %d\n", largest2);

    return 0;
}
```

Output:



```
Enter the size of the array
5
Enter the elements
23
45
21
7
89
The array elements are :
23      45      21      7       89
The FIRST LARGEST = 89
THE SECOND LARGEST = 45
```

c) **Write a C Program to perform insert, delete operations on a given Array**

Problem Solution

In this program, we have to

- Insert an element into an array at a specified position.
- Delete an element from an array at a specified position.

1. **Initialize Array:**
   - Declare an array array[] of size MAX (5).
   - Input the array elements from the user.
2. **Display Array Elements:**
   - Print the elements of the array.
3. **Insert Element at a Given Position:**

   - Define a function insert(array[], pos, num) to insert an element into the array.
   - Shift elements from the given position pos to the right by one position.
   - Insert the new element num at the specified position pos.
   - The last element of the array is lost due to the shift.

4. **Delete Element at a Given Position:**
   - Define a function delete(array[], pos) to delete an element from the array.
   - Shift elements from the specified position pos to the left by one position.
   - Set the last position to 0 after the deletion.
5. **Insert 74 at Position 3:**
   - Call the insert() function to insert the number 74 at position 3 in the array.
   - Display the array after insertion.
6. **Delete Element at Position 2:**
   - Call the delete() function to delete the element at position 2.
   - Display the array after deletion.

Example Walkthrough:

- Input: {10, 20, 30, 40, 50}
  - Insert 74 at position 3:
    - Output: {10, 20, 30, 74, 40}
  - Delete element at position 2:
    - Output: {10, 30, 74, 40, 0}

Time Complexity:

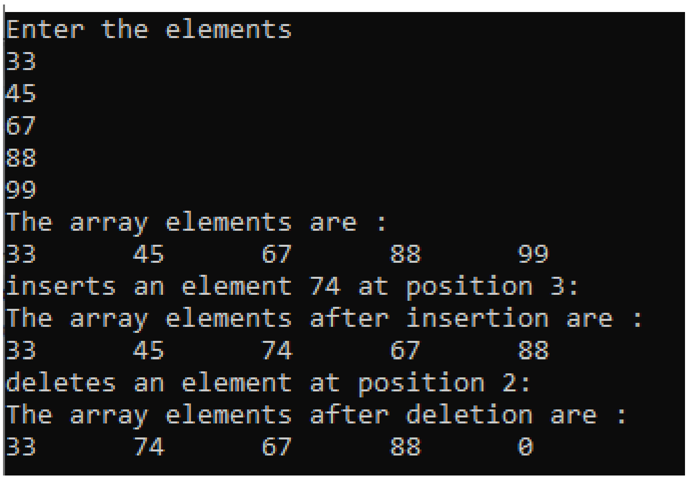
- **Insertion:** O(n) where n is the number of elements (shifting elements takes linear time).
- **Deletion:** O(n) (shifting elements left also takes linear time).

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

```
const int MAX=5;

void insert(int [ ],int , int );
void delete(int [ ],int );

int main ()
{
   int n = MAX, i = 0;



   int array[n];
   printf ("Enter the elements\n");
   for (i = 0; i < n; i++)
   {
      scanf ("%d", &array[i]);
   }

   printf ("The array elements are : \n");
   for (i = 0; i < n; i++)
   {
      printf ("%d\t", array[i]);
   }

   printf ("\n");

   printf ("inserts an element 74 at position 3: \n");
   insert(array, 3,  74);
   printf ("The array elements after insertion are : \n");
   for (i = 0; i < n; i++)
   {
      printf ("%d\t", array[i]);
```

```
    }

    printf ("\n");

    printf ("deletes an element at position 2: \n");
    delete(array, 2);
    printf ("The array elements after deletion are : \n");
    for (i = 0; i < n; i++)
    {
        printf ("%d\t", array[i]);
    }

    printf ("\n");


}
void insert(int array[ ],int pos, int num)
{
    int i;
    for(i=MAX-1;i>=pos;i--)
    {
        array[i]=array[i-1];
    }
    array[i]=num;
}
void delete(int array[ ],int pos)
{
    int i;
    for(i=pos;i<MAX;i++)
    {
       array[i-1]=array[i];
    }
    array[i-1]=0;
}
```

Output:



```
Enter the elements
33
45
67
88
99
The array elements are :
33      45      67      88      99
inserts an element 74 at position 3:
The array elements after insertion are :
33      45      74      67      88
deletes an element at position 2:
The array elements after deletion are :
33      74      67      88      0
```

Follow up Questions:

Give explanation for the selection made for MCQ questions

1) What is the problem in the following C declarations?

int func(int);

double func(int);

int func(float);

a) A function with same name cannot have different signatures

b) A function with same name cannot have different return types

c) A function with same name cannot have different number of parameters

d) All of the mentioned

2) What are the elements present in the array of the following C code?

int array[5] = {5};

a) 5, 5, 5, 5, 5

b) 5, 0, 0, 0, 0

c) 5, (garbage), (garbage), (garbage), (garbage)

d) (garbage), (garbage), (garbage), (garbage), 5

3) what will be the output of following program code?

```
int main(void)
{
char P;
char buf[10 ] ={1,2,3,4,5,6,9,8};
P = (buf +1)[5];
printf(“%d”,P);
return 0;
}
```

a) 5 b) 6 c) 9 d) error

4) Let X be an array, which of the following operations are illegal

a) ++X b) X+1 c) X++ d) X*2

5) Write a C Program to perform reverse, search and display operations on a given Array

Experiment 2:

a) **Write a C Program to perform swapping of two numbers using pass by reference**

Problem Solution

In this program, we have to Swap two numbers using pointers.

1. **Initialize Two Numbers:**
   - Declare two integer variables n1 = 5 and n2 = 10.

2. **Display the Values Before Swapping:**
   - Print the initial values of n1 and n2.

3. **Swap the Two Numbers Using Pointers:**
   - Define a function swap(int* a, int* b) that takes two pointers as arguments.
   - Inside the function, use a temporary variable temp to store the value of *a (i.e., n1).
   - Assign the value of *b (i.e., n2) to *a.
   - Assign the value of temp (original n1) to *b.
   - This effectively swaps the values of n1 and n2 by manipulating their memory addresses.

4. **Call the Swap Function:**
   - Pass the addresses of n1 and n2 (&n1 and &n2) to the swap() function.

5. **Display the Values After Swapping:**
   - After returning from the function, print the updated values of n1 and n2.

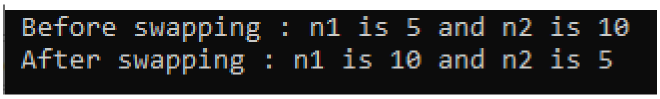
Example Walkthrough:

- Initially: n1 = 5 and n2 = 10.
- After swapping: n1 = 10 and n2 = 5.

Time Complexity:

- **O(1)** (constant time) since swapping two values takes a fixed amount of operations.

This simple swap operation demonstrates how pointers can be used to pass variables by reference in C, allowing their values to be modified in the calling function.



```
Before swapping : n1 is 5 and n2 is 10
After swapping : n1 is 10 and n2 is 5
```

b) Write a C Program to concatenate two strings using pointers without using built-in string functions.

Problem Solution

In this program, we have to concatenate two strings using pointers without using built-in string functions.

1. Declare character arrays str1[MAX_SIZE], str2[MAX_SIZE]

2. Input str1 and str2 from the user

3. Initialize pointer s1 to point to the start of str1

4. While *s1 is not null Move s1 to the next character (s1++)

5. Initialize pointer s2 to point to the start of str2

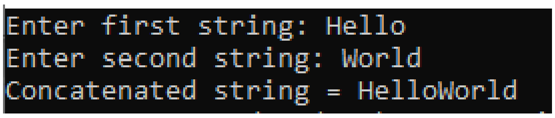
6. While *s2 is not null Copy *s2 to *s1 (s1++ = s2++)

7. Copy null terminator from str2 to str1 to properly terminate the string

8. Print the concatenated string using str1

```
#include <stdio.h>
#define MAX_SIZE 100 // Maximum string size

int main()
{
    char str1[MAX_SIZE], str2[MAX_SIZE];
    char * s1 = str1;
    char * s2 = str2;

    /* Input two strings from user */
    printf("Enter first string: ");
    gets(str1);
    printf("Enter second string: ");
    gets(str2);

    /* Move till the end of str1 */
    while(*(s1))
    {
        s1++;  // Move past null terminator
    }



    /* Copy str2 to str1 */
    while(*(s1++) = *(s2++));

    printf("Concatenated string = %s", str1);

    return 0;
}
```

Output:

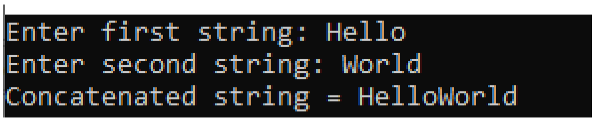


```
Enter first string: Hello
Enter second string: World
Concatenated string = HelloWorld
```

Or

```
#include <stdio.h>
#define MAX_SIZE 100 // Maximum string size

int main()
{
    char str1[MAX_SIZE], str2[MAX_SIZE];
    char * s1 = str1;
    char * s2 = str2;

    /* Input two strings from user */
    printf("Enter first string: ");
    gets(str1);
    printf("Enter second string: ");
    gets(str2);

    /* Move till the end of str1 */
    while(*s1 !='\0')
    {
      s1++;  // Move past null terminator
    }



    /* Copy str2 to str1 */
    while(*s2!='\0')
    {
      *s1 = *s2;
      s1++;
      s2++;
    }
    *s1 = '\0';
    printf("Concatenated string = %s", str1);

    return 0;
}
```

Output:



```
Enter first string: Hello
Enter second string: World
Concatenated string = HelloWorld
```

C Dynamic Memory Allocation

As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

To allocate memory dynamically, library functions are malloc(), calloc(), realloc() and free() are used. These functions are defined in the <stdlib.h> header file.

malloc( )

The name "malloc" stands for memory allocation.

The malloc( ) function reserves a block of memory of the specified number of bytes. And, it returns a pointer of void which can be casted into pointers of any form.

Syntax of malloc( )

```
ptr = (castType*) malloc(size);
```

Example

```
ptr = (float*) malloc(100 * sizeof(float));
```

The above statement allocates 400 bytes of memory. It's because the size of float is 4 bytes. And, the pointer ptr holds the address of the first byte in the allocated memory.

The expression results in a NULL pointer if the memory cannot be allocated.

calloc( )

The name "calloc" stands for contiguous allocation.

The malloc( ) function allocates memory and leaves the memory uninitialized, whereas the calloc( ) function allocates memory and initializes all bits to zero.

Syntax of calloc( )

```
ptr = (castType*)calloc(n, size);
```

Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

The above statement allocates contiguous space in memory for 25 elements of type float.

free( )

Dynamically allocated memory created with either calloc( ) or malloc( ) doesn't get freed on their own. You must explicitly use free( ) to release the space.

Syntax of free( )

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr.

c) **Write a C Program to calculate the sum of n numbers entered by the user, Dynamically allocate memory to store the n numbers.**

Problem Solution

The program is to Dynamically allocate memory for an integer array, take input for its elements using scanf, calculate the sum of the elements, and finally free the memory.

1. Start

2. Declare integer variables n, i, sum = 0, and pointer ptr

3. Input the value of n (number of elements)

4. Allocate memory for n integers using malloc: ptr = (int*) malloc(n * sizeof(int))

5. If memory allocation fails (ptr == NULL), print error message and exit

6. Input elements: For i from 0 to n-1:

a. Input element into ptr[i] using scanf

b. Add ptr[i] to sum

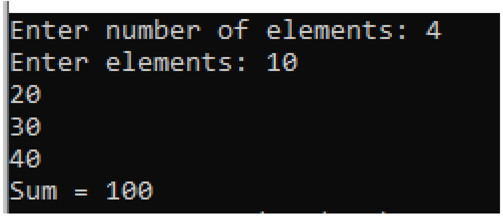
7. Print the sum of elements

8. Free the allocated memory using free(ptr)

9. End

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

int main() {
  int n, i, *ptr, sum = 0;

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

  ptr = (int*) malloc(n * sizeof(int));

  // if memory cannot be allocated
  if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
  }

  printf("Enter elements: ");
  for(i = 0; i < n; ++i) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
  }

  printf("Sum = %d", sum);

  // deallocating the memory
  free(ptr);

  return 0;
}
```

Output:

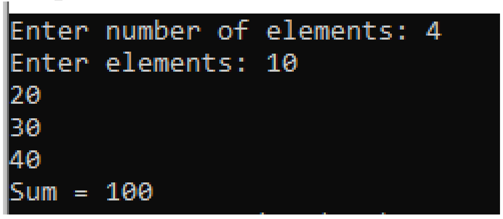


```
Enter number of elements: 4
Enter elements: 10
20
30
40
Sum = 100
```

Or

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

int main() {
  int n, i, *ptr, sum = 0;

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

  ptr = (int*) malloc(n * sizeof(int));
  // if memory cannot be allocated
  if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
  }

  printf("Enter elements: ");
  for(i = 0; i < n; ++i) {
    scanf("%d", &ptr[i]);
    sum += ptr[i];
  }

  printf("Sum = %d", sum);

  // deallocating the memory
  free(ptr);

  return 0;
}
```

Output:



```
Enter number of elements: 4
Enter elements: 10
20
30
40
Sum = 100
```

Follow up Questions:

1) Program to find maximum of n numbers using dynamic arrays (Use malloc( ) and calloc( ) for creating dynamic array)
2) What will happen if the following program segment is executed?
   int *ptr;
   ptr = (int *) malloc (10);
3) Find the final values stored in the variables x, y and z at the end of the program:

```
void main( )
 {
 int x, y, z, *p, *q;
 x = 10;
y = 15;
z= 20;
 p=&x;
q=&z;
*q = *p + y - 3;
y = y- (*p);
*p = *q-z;
}
```

Experiment 3:

a) **Create a structure named Complex to represent a complex number with real and imaginary parts. Write a C program to add and multiply two complex numbers.**

Problem Solution:

1) **Define Structure**:

- Define a structure named Complex with two members:
    - real: a float to store the real part of the complex number.
    - imag: a float to store the imaginary part of the complex number.

2) **Define Function for Addition**:

- Create a function addComplex that takes two Complex structures as parameters and returns their sum as a Complex structure:
    - Inside the function, create a result variable of type Complex.
    - Set result.real to the sum of the real parts of the two input numbers.
    - Set result.imag to the sum of the imaginary parts of the two input numbers.
    - Return the result.

3) **Define Function for Multiplication**:

- Create a function multiplyComplex that takes two Complex structures as parameters and returns their product as a Complex structure:
    - Inside the function, create a result variable of type Complex.
    - Set result.real to the result of the formula: (num1.real×num2.real)−(num1.imag×num2.imag)(num1.real \times num2.real) - (num1.imag \times num2.imag)(num1.real×num2.real)−(num1.imag×num2.imag).
    - Set result.imag to the result of the formula: (num1.real×num2.imag)+(num1.imag×num2.real)(num1.real \times num2.imag) + (num1.imag \times num2.real)(num1.real×num2.imag)+(num1.imag×num2.real).
    - Return the result.

4) **Define Function for Displaying Complex Numbers**:

- Create a function displayComplex that takes a Complex structure as a parameter and prints its value in the format real + imagi.

5) **Main Function**:

- Declare variables of type Complex for the first and second complex numbers, and for the results of addition and multiplication:
    - complexNum1, complexNum2, sumResult, productResult.
- Prompt the user to input the real and imaginary parts for complexNum1:
    - Use scanf to read the values into complexNum1.real and complexNum1.imag.
- Prompt the user to input the real and imaginary parts for complexNum2:
    - Use scanf to read the values into complexNum2.real and complexNum2.imag.
- Call the addComplex function to add complexNum1 and complexNum2, storing the result in sumResult.
- Call the multiplyComplex function to multiply complexNum1 and complexNum2, storing the result in productResult.
- Print the sum of the complex numbers by calling displayComplex with sumResult.
- Print the product of the complex numbers by calling displayComplex with productResult.

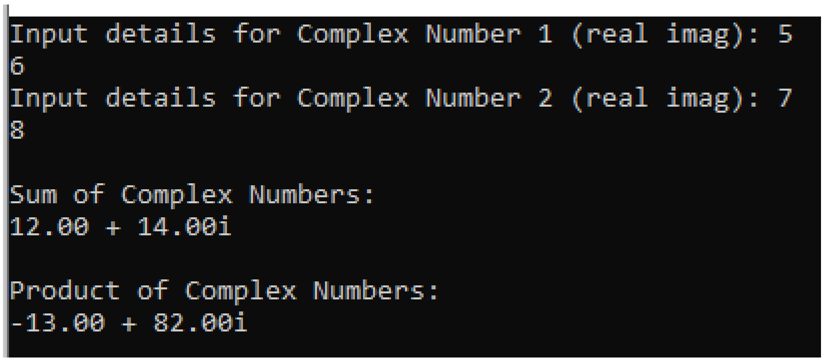
6) **End Program**:

- Return 0 to indicate successful completion of the program.

```
#include <stdio.h>
#include <stdlib.h>
// Define the structure "Complex"
struct Complex {
   float real;
   float imag;
};

// Function to add two complex numbers
struct Complex addComplex(struct Complex num1, struct Complex num2) {
   struct Complex result;
   result.real = num1.real + num2.real;
   result.imag = num1.imag + num2.imag;
   return result;
}

// Function to multiply two complex numbers
struct Complex multiplyComplex(struct Complex num1, struct Complex num2) {
   struct Complex result;
   result.real = (num1.real * num2.real) - (num1.imag * num2.imag);
   result.imag = (num1.real * num2.imag) + (num1.imag * num2.real);
   return result;
}
```

```
// Function to display a complex number
void displayComplex(struct Complex num) {
    printf("%.2f + %.2fi\n", num.real, num.imag);
}

int main() {
    // Declare variables to store two complex numbers
    struct Complex complexNum1, complexNum2, sumResult, productResult;

    // Input details for the first complex number
    printf("Input details for Complex Number 1 (real imag): ");
    scanf("%f %f", &complexNum1.real, &complexNum1.imag);

    // Input details for the second complex number
    printf("Input details for Complex Number 2 (real imag): ");
    scanf("%f %f", &complexNum2.real, &complexNum2.imag);

    // Add the two complex numbers
    sumResult = addComplex(complexNum1, complexNum2);

    // Multiply the two complex numbers
    productResult = multiplyComplex(complexNum1, complexNum2);

    // Display the results
    printf("\nSum of Complex Numbers:\n");
    displayComplex(sumResult);

    printf("\nProduct of Complex Numbers:\n");
    displayComplex(productResult);

    return 0;
}
```

Output:



```
Input details for Complex Number 1 (real imag): 5
6
Input details for Complex Number 2 (real imag): 7
8

Sum of Complex Numbers:
12.00 + 14.00i

Product of Complex Numbers:
-13.00 + 82.00i
```

b) Write a C Program to create a new structure called Student that contains the following fields: Name (a string of maximum 20 characters), Roll (an integer), and avg_marks (a float). Then, write a main function defines a student structure, initializes a student instance S1, dynamically allocates memory for another student instance S2, and uses pointers to access and print values for both S1 and S2.

Problem Solution:

1. **Define Structure**:

- Create a structure student with the following fields:
  - name: Character array of size 10 to store the student's name.
  - roll: Integer to store the student's roll number.
  - avg_marks: Float to store the student's average marks.

2. **Initialize Static Student Instance**:

- Declare and initialize a student instance S1 with predefined values: name as "abc", roll as 1, and avg_marks as 87.0.

3. **Allocate Memory for Dynamic Student Instance**:

- Dynamically allocate memory for a new student instance S2 using malloc.
- Check if the allocation was successful:
  - If memory allocation fails, print an error message and exit the program.

4. **Print Details of Static Instance (S1)**:

- Display the name, roll, and avg_marks of S1 using printf.

5. **Input Data for Dynamic Instance (S2)**:

- Prompt the user to input values for name, roll, and avg_marks of S2.
- Use scanf to read user input:
  - name with a 9-character limit to avoid buffer overflow.
  - roll as an integer.
  - avg_marks as a float.

6. **Print Details of Dynamic Instance (S2)**:

- Display the name, roll, and avg_marks of S2 using printf.

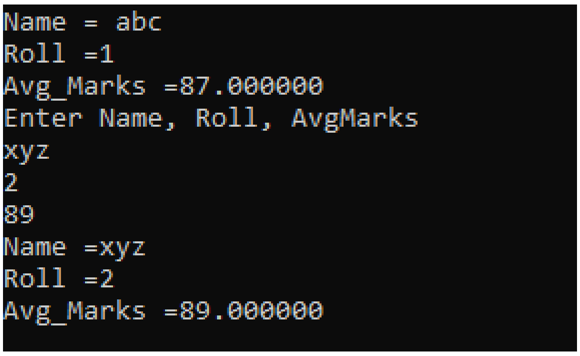
7. **Free Allocated Memory**:

- Release the memory allocated for S2 using free.
- Set the pointer S2 to NULL to avoid dangling pointers.

8. **End Program**:

- Return from main, signaling successful completion.

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

struct student
{
   char name[10];
   int roll;
   float avg_marks;
};
struct student S1 = {"abc",1,87};

int main()
{
  struct student *S2;

  S2 = (struct student*)malloc(sizeof(struct student));
  if(S2 == NULL)
  {
     printf("error allocating memory");
     exit(0);
  }

  printf("Name = %s\n",S1.name);
  printf("Roll =%d\n",S1.roll);
  printf("Avg_Marks =%f\n",S1.avg_marks);

  printf("Enter Name, Roll, AvgMarks\n");
  scanf("%s",S2->name);
  scanf("%d",&S2->roll);
  scanf("%f",&S2->avg_marks);

  printf("Name =%s\n",S2->name);
  printf("Roll =%d\n",S2->roll);
  printf("Avg_Marks =%f\n",S2->avg_marks);

  free(S2);
  S2= NULL;
  return 0;
}
```



```
Name = abc
Roll =1
Avg_Marks =87.000000
Enter Name, Roll, AvgMarks
xyz
2
89
Name =xyz
Roll =2
Avg_Marks =89.000000
```

c) Create a structure named Book to store book details like title, author, and price. Write a C program to input details for three books, find the most expensive and the lowest priced books, and display their information

Problem Solution:

1) **Define Structure**:

- Define a structure Book with the following members:
  - o title: a character array of size 100 to store the book title.
  - o author: a character array of size 100 to store the author's name.
  - o price: a float to store the book's price.

2) **Declare Book Variables**:

- In the main function, declare three variables of type Book: book1, book2, and book3.

3) **Input Book Details**:

- For each book (from book1 to book3):
  1. Print a prompt to input details for the book (e.g., "Input details for Book 1").
  2. Read the title using scanf. Assume the title does not contain spaces.
  3. Read the author using scanf. Assume the author does not contain spaces.
  4. Read the price using scanf.

4) **Determine the Most Expensive Book**:

- Initialize a Book variable mostExpensive.
- Compare the prices of book1, book2, and book3:
  - o If book1 has the highest price, set mostExpensive to book1.
  - o Else if book2 has the highest price, set mostExpensive to book2.
  - o Otherwise, set mostExpensive to book3.

5) **Determine the Lowest Priced Book**:

- Initialize a Book variable lowestPriced.
- Compare the prices of book1, book2, and book3:
    - If book1 has the lowest price, set lowestPriced to book1.
    - Else if book2 has the lowest price, set lowestPriced to book2.
    - Otherwise, set lowestPriced to book3.

6) **Display Most Expensive Book Details**:

- Print the details of the most expensive book (mostExpensive.title, mostExpensive.author, and mostExpensive.price).

7) **Display Lowest Priced Book Details**:

- Print the details of the lowest priced book (lowestPriced.title, lowestPriced.author, and lowestPriced.price).

8) **End Program**:

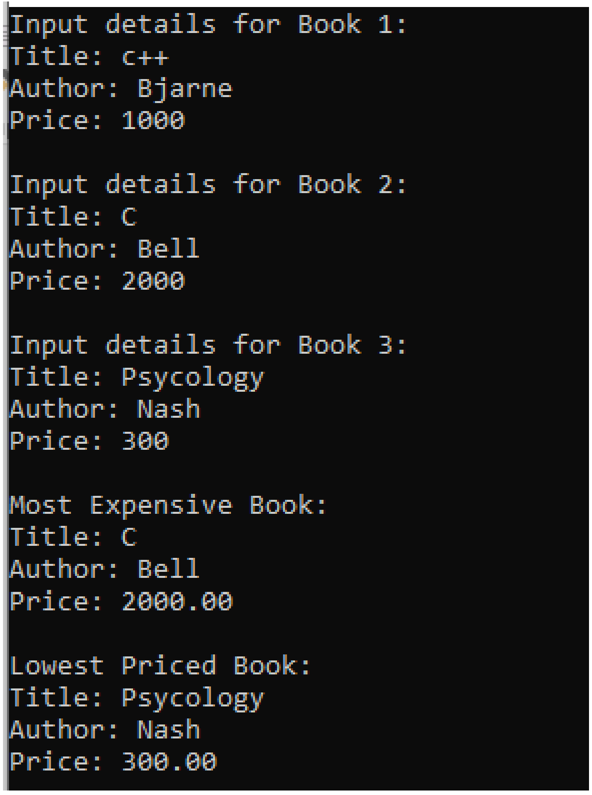
- Return 0 to indicate successful completion of the program.

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

// Define the structure "Book"
struct Book {
   char title[100];
   char author[100];
   float price;
};

int main() {
   // Declare variables to store details for three books
   struct Book book1, book2, book3;

   // Input details for the first book
   printf("Input details for Book 1:\n");
   printf("Title: ");
   scanf("%s", book1.title); // Assuming titles do not contain spaces
   printf("Author: ");
   scanf("%s", book1.author); // Assuming authors do not contain spaces
   printf("Price: ");
   scanf("%f", &book1.price);

   // Input details for the second book
```

```
printf("\nInput details for Book 2:\n");
printf("Title: ");
scanf("%s", book2.title);
printf("Author: ");
scanf("%s", book2.author);
printf("Price: ");
scanf("%f", &book2.price);

// Input details for the third book
printf("\nInput details for Book 3:\n");
printf("Title: ");
scanf("%s", book3.title);
printf("Author: ");
scanf("%s", book3.author);
printf("Price: ");
scanf("%f", &book3.price);

// Find the most expensive book
struct Book mostExpensive;
if (book1.price >= book2.price && book1.price >= book3.price) {
   mostExpensive = book1;
} else if (book2.price >= book1.price && book2.price >= book3.price) {
   mostExpensive = book2;
} else {
   mostExpensive = book3;
}

// Find the lowest priced book
struct Book lowestPriced;
if (book1.price <= book2.price && book1.price <= book3.price) {
   lowestPriced = book1;
} else if (book2.price <= book1.price && book2.price <= book3.price) {
   lowestPriced = book2;
} else {
   lowestPriced = book3;
}

// Display information for the most expensive book
printf("\nMost Expensive Book:\n");
printf("Title: %s\n", mostExpensive.title);
printf("Author: %s\n", mostExpensive.author);
printf("Price: %.2f\n", mostExpensive.price);

// Display information for the lowest priced book
printf("\nLowest Priced Book:\n");
printf("Title: %s\n", lowestPriced.title);
```

```
    printf("Author: %s\n", lowestPriced.author);
    printf("Price: %.2f\n", lowestPriced.price);


    return 0;
}
```

Output:



```
Input details for Book 1:
Title: c++
Author: Bjarne
Price: 1000

Input details for Book 2:
Title: C
Author: Bell
Price: 2000

Input details for Book 3:
Title: Psycology
Author: Nash
Price: 300

Most Expensive Book:
Title: C
Author: Bell
Price: 2000.00

Lowest Priced Book:
Title: Psycology
Author: Nash
Price: 300.00
```

Follow up Questions:

1) Write a C program that defines a structure String containing a char pointer str for storing a string. In the main function, dynamically allocate memory to store a string of up to 10 characters in str. Prompt the user to enter a string, store it in str, and then display the entered string. Next, allocate enough memory for another character pointer. Copy the contents of str into str2 and display the copied string. Ensure that the program handles potential memory allocation failures, avoids buffer overflow, and performs proper memory management by freeing any allocated memory before the program ends.

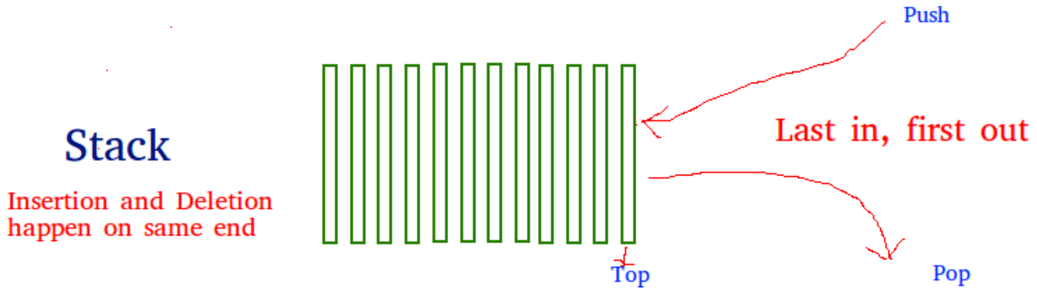
2) Write a C program defines a structure Student with Name, USN, Mark1, Mark2 as members. Write a program to arrange student's information in ascending order using bubble sort algorithm

Experiment 4:

a) Write a program to illustrate forward and backward surfing in the web browser using stack (Array implementation/ Linked list implementation). Display the appropriate messages in case of exceptions. Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

Problem Solution:

There are many real-life examples of a stack. Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order.



Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose,

the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

Verifying whether the Stack is full: isFull( )

The isFull() operation checks whether the stack is full. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

```
1. START
2. If the size of the stack is equal to the top position of the stack,
   the stack is full. Return 1.
3. Otherwise, return 0.
4. END
```

Verifying whether the Stack is empty: isEmpty( )

The isEmpty() operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

```
1. START
2. If the top value is -1, the stack is empty. Return 1.
3. Otherwise, return 0.
4. END
```

Stack Insertion: push( )

Algorithm

```
1. Checks if the stack is full.
2. If the stack is full, produces an error and exit.
3. If the stack is not full, increments top to point next empty space.
4. Adds data element to the stack location, where top is pointing.
5. Returns success.
```

Stack Deletion: pop( )

Algorithm

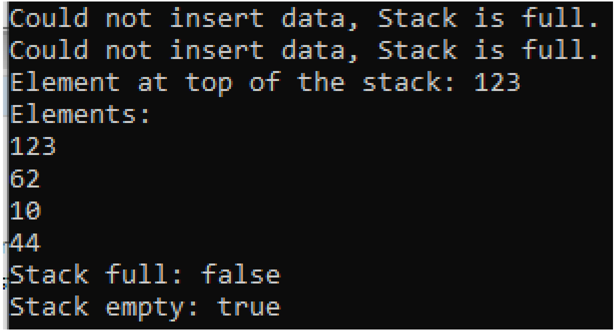
```
1. Checks if the stack is empty.
2. If the stack is empty, produces an error and exit.
3. If the stack is not empty, accesses the data element at which top is pointing.
4. Decreases the value of top by 1.
5. Returns success.
```

```
#include <stdio.h>
#define STACKSIZE  4

int stack[STACKSIZE];
int top = -1;
/* Check if the stack is empty */
int isempty(){
   if(top == -1)
      return 1;
   else
      return 0;
}
/* Check if the stack is full */
int isfull(){
   if(top == STACKSIZE-1)
      return 1;
   else
      return 0;
```

```
}

/* Function to return the topmost element in the stack */
int peek(){
   return stack[top];
}

/* Function to delete from the stack */
int pop(){
   int data;
   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}

/* Function to insert into the stack */
void push(int data){
   if(!isfull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
/* Main function */
int main(){
   push(44);
   push(10);
   push(62);
   push(123);
   push(15);
   push(75);
   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:



```
Could not insert data, Stack is full.
Could not insert data, Stack is full.
Element at top of the stack: 123
Elements:
123
62
10
44
Stack full: false
Stack empty: true
```

b) Write a program to convert and print a given valid parenthesized infix arithmetic expression to postfix expression. The expression consists of single character operands and binary operators + - * /.

Problem Solution:

Algorithm for Infix to Postfix Conversion

1. If the input character is an operand, print it.
2. If the input character is an operator-
    1. If stack is empty push it to the stack.
    2. If its precedence value is greater than the precedence value of the character on top, push.
    3. If its precedence value is lower or equal then pop from stack and print while precedence of top char is more than the precedence value of the input character.
3. If the input character is ‘)’, then pop and print until top is ‘(‘. (Pop ‘(‘ but don’t print it.)
4. If stack becomes empty before encountering ‘(‘, then it’s a invalid expression.
5. Repeat steps 1-4 until input expression is completely read.
6. Pop the remaining elements from stack and print them.

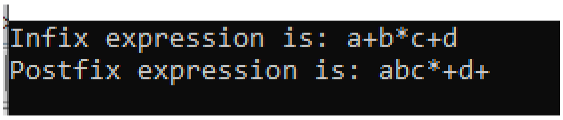
- The above method handles right associativity of exponentiation operator (here, ^) by assigning it higher precedence value outside stack and lower precedence value inside stack whereas it’s opposite for left associative operators.

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

// Function to return precedence of operators
int prec(char c) {

    if (c == '^')
        return 3;
    else if (c == '/' || c == '*')
```

```
        return 2;
    else if (c == '+' || c == '-')
        return 1;
    else
        return -1;
}



// Function to perform infix to postfix conversion
void infixToPostfix(char* exp) {
    int len = strlen(exp);
    char result[len + 1];
    char stack[len];
    int j = 0;
    int top = -1;

    for (int i = 0; i < len; i++) {
        char c = exp[i];

        // If the scanned character is
        // an operand, add it to the output string.
        if (isalnum(c))
            result[j++] = c;

        // If the scanned character is
        // an '(', push it to the stack.
        else if (c == '(' )
            stack[++top] = '(';

        // If the scanned character is an ')',
        // pop and add to the output string from the stack
        // until an '(' is encountered.
        else if (c == ')' )
        {
            while (top != -1 && stack[top] != '(' )
            {
                result[j++] = stack[top--];
            }
            top--;    //Pop '(' from stack
        }

        // If an operator is scanned
        else {
            while (top != -1 && (prec(c) < prec(stack[top]) ||
                                 prec(c) == prec(stack[top])))
```

```
          {
             result[j++] = stack[top--];
          }
          stack[++top] = c;     // if Prec of (c) > prec(stack[top] or if stack is empty, Push
operator onto stack
          }
   }

   // Pop all the remaining elements from the stack
   while (top != -1) {
      result[j++] = stack[top--];
   }

   result[j] = '\0';
   printf("Postfix expression is: %s\n", result);
}

int main() {
   //char exp[] = "a+b*(c^d-e)^(f+g*h)-i";
   char exp[] ="a+b*c+d";
   printf("Infix expression is: %s\n" , exp);
   infixToPostfix(exp);
   return 0;
}
```

OUTPUT:



```
Infix expression is: a+b*c+d
Postfix expression is: abc*+d+
```

c) A call center phone system has to hold the phone calls from customers and provide service based on the arrival time of the calls. Write a C program to simulate this system using queue data structure. Program should have options to add and remove the phone calls in appropriate order for their service.

Problem Solution:

Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is **F**irst **I**n **F**irst **O**ut (FIFO). A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.

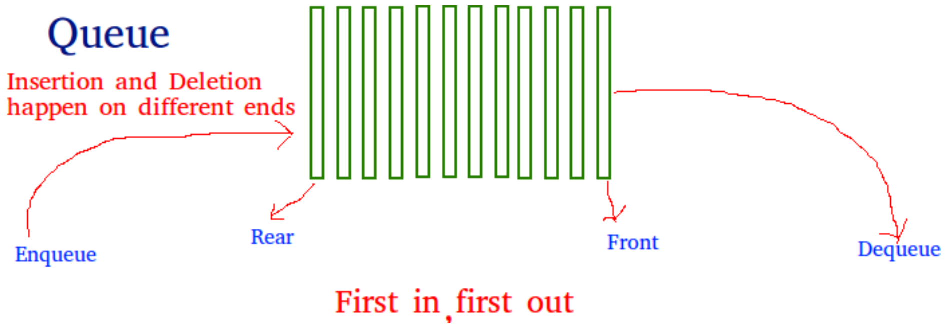
The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added. **Operations on Queue:**

Mainly the following four basic operations are performed on queue:

Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
Front: Get the front item from queue.

Rear: Get the last item from queue.



Algorithm for addq( ):
The addq function:

- Checks if the queue is full (if rear is at the maximum index MAX-1).
- If not full, increments rear and adds the new item to the rear of the queue.
- If this is the first item added, updates the front pointer to 0.

Algorithm for delq( ):
The delq function:

- Check if the queue is empty by checking if front is -1. If so, print a message and return a special value (e.g., -1) to indicate an empty queue.
- If the queue is not empty, store the data at the front of the queue in the data variable.
- Reset the value at the front of the arr to 0.
- If front is equal to rear, it means this was the only element in the queue, so reset both front and rear to -1 to indicate an empty queue.
- Otherwise, just increment the front pointer to dequeue the element.
- Return the dequeued data.

Algorithm for display( ):
The display function:

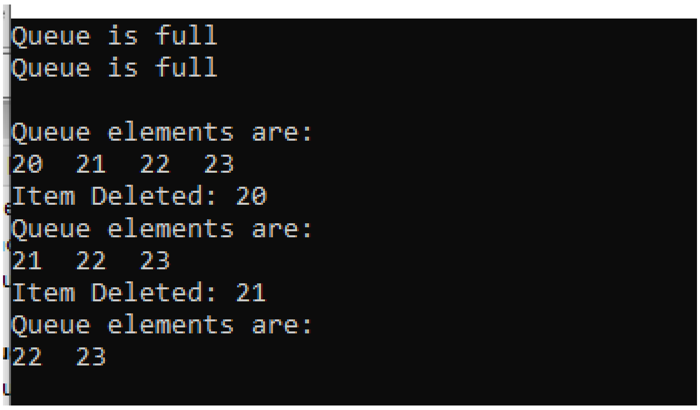
- Check if the queue is empty by checking if rear is -1. If so, print a message indicating that the queue is empty and return.
- If the queue is not empty, print a header message "Queue elements are:".

- Use a for loop to iterate from the front index to the rear index, printing each element of the arr array.
- After the loop, print a newline character to provide a clean output.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX  4

int arr[MAX];
int front=-1,rear=-1;

void addq(int item)
{
   if(rear==MAX-1)
   {
      printf("Queue is full\n");
      return;
   }
   rear++;
   arr[rear]=item;
   if(front==-1)
   {
      front=0;
   }
}

int delq()
{
   int data;
   if(front==-1)
   {
      printf("Queue is Empty:\n");
      return NULL;
   }
   data=arr[front];
   arr[front]=0;
   if(front==rear)
      front=rear=-1;
   else
      front++;
   return data;
}

void display() {
  if (rear == -1)
```

```
    printf("\nQueue is Empty!!!");
  else {
    int i;
    printf("\nQueue elements are:\n");
    for (i = front; i <= rear; i++)
      printf("%d  ", arr[i]);
  }
  printf("\n");
}
int main()
{
    addq(20);
    addq(21);
    addq(22);
    addq(23);
    addq(24);
    addq(25);
    display();
    int i=delq();
    printf("Item Deleted: %d ",i);
    display();
    printf("Item Deleted: %d", delq());
    display();
    return 0;
}
```

OUTPUT:



```
Queue is full
Queue is full

Queue elements are:
20   21   22   23
Item Deleted: 20
Queue elements are:
21   22   23
Item Deleted: 21
Queue elements are:
22   23
```

Follow up Questions:

1) Create structure STACK, include data members integer array and integer variable top. Write a program to implement STACK operations using the structure STACK.

2) Write a program to reverse a string using Stack data structure.

Experiment 5:

Write a program to illustrate memory allocation to files based on their size using singly linked list. Program must support the following operations on a singly linked list.

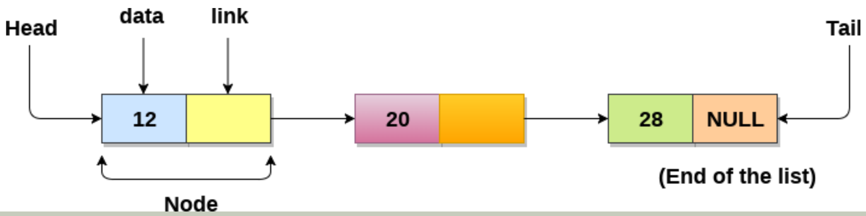
a. Program to add and delete nodes at the end of the Singly linked list

Problem Solution:

Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.

A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.

The last node of the list contains pointer to the null.



SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted.

1. Data Structure:

- The linked list is represented using a structure node with two fields:
  - info: stores the integer data for each node.
  - link: a pointer to the next node in the list.
- Head is a pointer of type NODE* which points to the first node of the list, and it is initially set to NULL (indicating the list is empty).

2. Function AddatEnd(int num):

This function adds a node containing the integer num at the end of the linked list.

Steps of AddatEnd:

1. **Allocate Memory for the New Node**:
    - Allocate memory dynamically for a new node (temp) using malloc(). The temp node will store the value num and its link will be NULL initially (since it will be added at the end).
2. **Assign Data to the New Node**:
    - Set temp->info = num to store the data passed as the argument to the function.
    - Set temp->link = NULL as this node will be the last one in the list.
3. **Handle the Case Where the List is Empty**:
    - If Head == NULL (i.e., the list is empty), set Head = temp to make the newly created node the first node in the list.
4. **Traverse to the End of the List**:
    - If the list is not empty (Head != NULL), use a pointer CurrPtr to traverse the list from the head node.
    - Traverse the list until you find the last node (i.e., a node whose link is NULL).
5. **Add the New Node at the End**:
    - Once the last node is found, set its link to point to the new node (CurrPtr->link = temp).

Function DelAtEnd():

1. **If the list is empty**:
    - Head == NULL, so the function prints "Linked List Empty" and returns.
    - No nodes are deleted.
2. **If the list contains only one node**:
    - Head->link == NULL, indicating there's only one node.
    - The function prints the message, deletes the node, and frees the memory.
    - Head is set to NULL, making the list empty.
3. **If the list contains multiple nodes**:
    - The function starts from Head and moves through the list to find the last node (NextPtr->link == NULL).
    - The second-to-last node is tracked using Currptr.
    - Currptr->link = NULL is used to remove the last node.
    - The node is deleted and its memory is freed.

Function DisplayNode:

1. Initialize a pointer CurrPtr:

- CurrPtr is a pointer of type NODE* and is initialized to point to Head. This pointer is used to traverse the linked list.

2. Check if the list is empty:

- **Condition**: if (CurrPtr == NULL).
- **Action**: If CurrPtr is NULL (indicating that the list is empty), print the message "Empty Linked List" and return from the function.

3. Traverse and Display Node Contents:

- **Condition**: else (i.e., when the list is not empty).
- **Action**:

  - Start a while loop that continues until CurrPtr == NULL.
  - Inside the loop, print the value of the node (CurrPtr->info), followed by the string --> to visually represent the linked list structure.
  - Move to the next node by updating CurrPtr = CurrPtr->link.

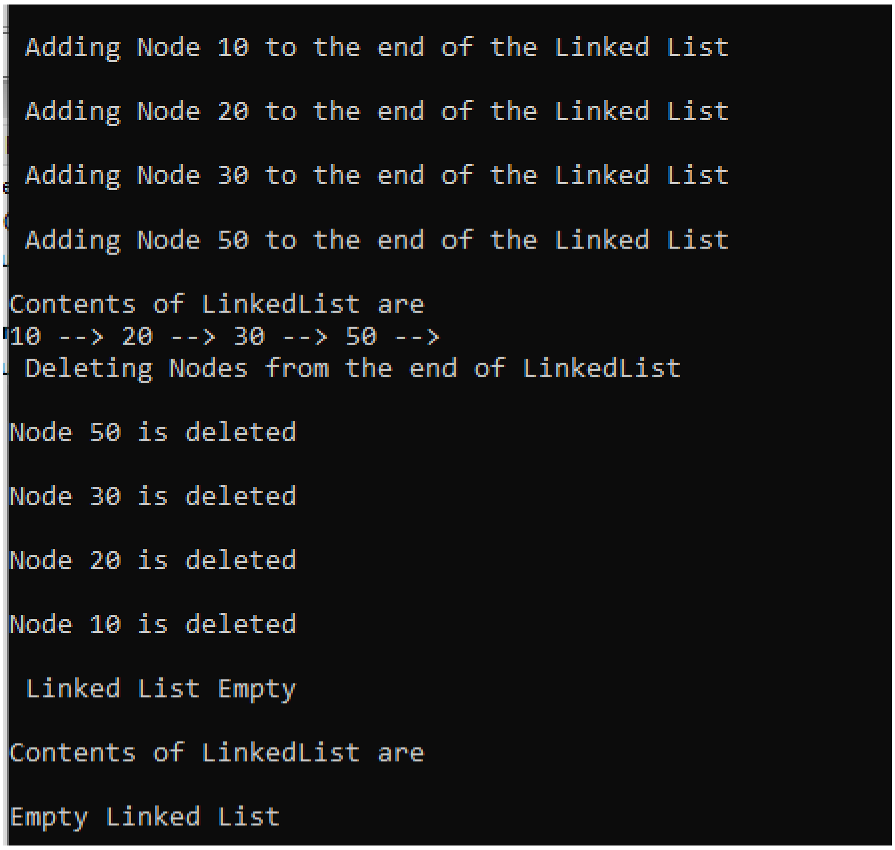
4. End of List:

- When CurrPtr becomes NULL (i.e., the end of the list is reached), the loop ends, and the function finishes.

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
   int info;
   struct node *link;
};
typedef struct node NODE;

NODE *Head = NULL;

void AddatEnd(int num)
{
   NODE *temp, *CurrPtr;
   temp = (NODE*)malloc(sizeof(NODE));
   temp->info = num;
   temp->link = NULL;
   printf("\n Adding Node %d to the end of the Linked List\n",temp->info);
   if(Head == NULL)
      Head = temp;
   else
   {
      CurrPtr = Head;
      while(CurrPtr->link != NULL)
      {
         CurrPtr = CurrPtr->link;
      }
      CurrPtr->link = temp;
   }
}
void DelAtEnd()
{
   NODE *Currptr, *NextPtr;
   NextPtr = Head;
   if(Head == NULL)
   {
      printf("\n Linked List Empty\n");
      return;
```

```
    }
    else if(Head->link == NULL)
    {
        printf("\nNode %d is deleted\n",NextPtr->info);
        Head = NULL;
        free(NextPtr);
        return;
    }
    else
    {
        while(NextPtr->link != NULL)
        {
        Currptr = NextPtr;
        NextPtr = NextPtr->link;
        }
        Currptr->link = NULL;
        printf("\nNode %d is deleted\n",NextPtr->info);
        free(NextPtr);
    }

}
void DisplayNode()
{
    printf("\nContents of LinkedList are\n");
    NODE *CurrPtr = Head;
    if(CurrPtr == NULL)
    {
        printf("\nEmpty Linked List\n");
    }
    else
    {
        while(CurrPtr != NULL)
        {
        printf("%d --> ",CurrPtr->info);
        CurrPtr = CurrPtr->link;
        }
    }
}
int main()
{
    printf("\nAdding Nodes to the end of LinkedList \n");
    AddatEnd(10);
    AddatEnd(20);
    AddatEnd(30);
    AddatEnd(50);
    DisplayNode();
    printf("\n Deleting Nodes from the end of LinkedList \n");
    DelAtEnd();
    DelAtEnd();
    DelAtEnd();
    DelAtEnd();
    DelAtEnd();
    DisplayNode();
    return 0;
}
```

OUTPUT:



```
 Adding Node 10 to the end of the Linked List

 Adding Node 20 to the end of the Linked List

 Adding Node 30 to the end of the Linked List

 Adding Node 50 to the end of the Linked List

Contents of LinkedList are
10 --> 20 --> 30 --> 50 -->
 Deleting Nodes from the end of LinkedList

Node 50 is deleted

Node 30 is deleted

Node 20 is deleted

Node 10 is deleted

 Linked List Empty

Contents of LinkedList are

Empty Linked List
```

b. Program to add and delete nodes at the beginning of the Singly linked list

Problem Solution:

Algorithm for the AddAtBeg Function:

The AddAtBeg function is designed to insert a new node with the value num at the beginning of a singly linked list. This function handles both the case where the list is empty and when the list already has nodes.

Function AddAtBeg:

1. Create a New Node:

- **Memory Allocation**: A new node (temp) is created using malloc(), which dynamically allocates memory for the node.
- **Assign Data**: The value num passed as an argument is assigned to temp->info.

- **Set the Link to NULL**: The link field of temp is set to NULL. This is because initially, this new node will be placed at the front, and it will point to the existing first node (if any).

2. Insert the New Node at the Beginning:

- **Condition 1**: If the linked list is empty (Head == NULL), set Head = temp. This means the new node becomes the first and only node in the list.
- **Condition 2**: If the list is not empty (Head != NULL), set temp->link = Head. This makes the new node's link point to the current first node, effectively inserting the new node at the beginning. Then, update Head = temp, making the new node the new first node in the list.

3. Print a Message:

- A message is printed to indicate that the node has been added at the beginning of the list, displaying the node's value (temp->info).

Algorithm for the DelAtBeg Function:

The DelAtBeg function is designed to delete the first node (head node) of a singly linked list. This function handles the case where the list is empty, and also removes the first node and updates the Head pointer when the list is non-empty.

Step-by-Step Breakdown of DelAtBeg:

1. Check if the List is Empty:

- **Condition**: if (Head == NULL).
- **Action**: If Head == NULL, the list is empty. Print the message "Linked List Empty" and return from the function, as there is no node to delete.

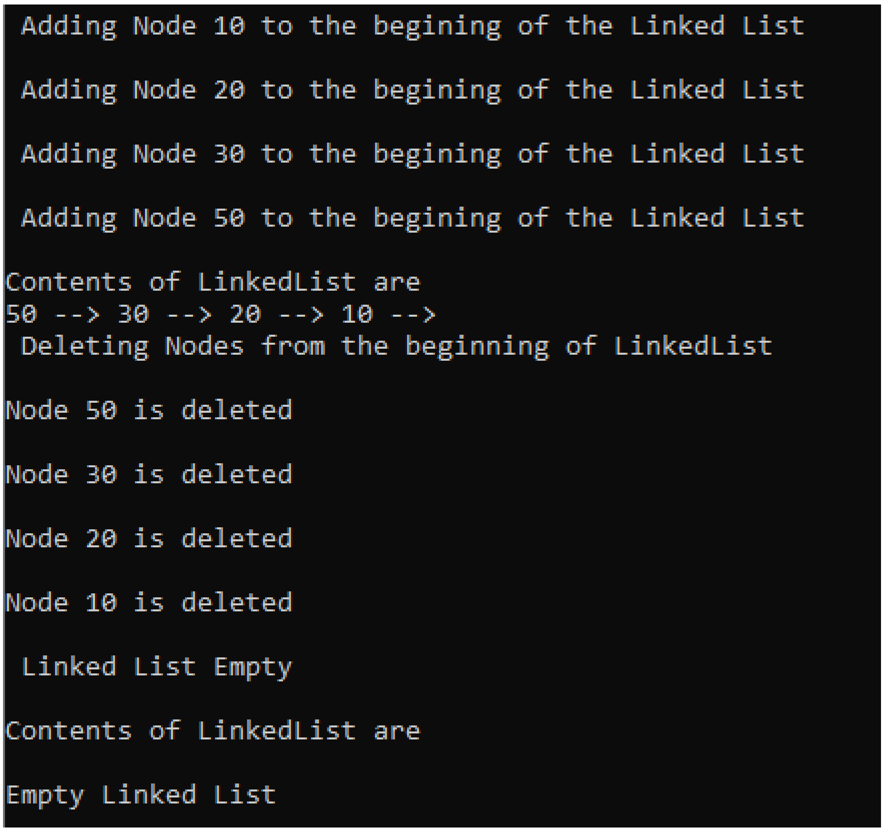
2. Deleting the First Node:

- **Condition**: else (the list is not empty).
- **Action**:
    - Set NextPtr = Head. This pointer will point to the node that is currently the first node in the list.
    - Print the value of the node to be deleted: printf("\nNode %d is deleted", NextPtr->info);.
    - Update Head = Head->link. This makes the second node (if any) the new head of the list. If there was only one node, Head becomes NULL.
    - Free the memory of the deleted node using free(NextPtr).

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
   int info;
   struct node *link;
};
typedef struct node NODE;

NODE *Head = NULL;
void AddAtBeg(int num)
{
   NODE *temp;
   temp = (NODE*)malloc(sizeof(NODE));
   temp->info = num;
   temp->link = NULL;
   printf("\n Adding Node %d to the beginning of the Linked List\n",temp->info);
   if(Head == NULL)
       Head = temp;
   else
   {
       temp->link = Head;
       Head = temp;
   }
}
void DelAtbeg()
{
   NODE *Currptr, *NextPtr;

   if(Head == NULL)
   {
       printf("\n Linked List Empty\n");
       return;
   }
   else
   {
       NextPtr = Head;
       printf("\nNode %d is deleted\n",NextPtr->info);
       Head = Head->link;
       free(NextPtr);
   }
}
void DisplayNode()
{
   printf("\nContents of LinkedList are\n");
   NODE *CurrPtr = Head;
   if(CurrPtr == NULL)
   {
       printf("\nEmpty Linked List\n");
   }
   else
   {
       while(CurrPtr != NULL)
       {
       printf("%d --> ",CurrPtr->info);
       CurrPtr = CurrPtr->link;
       }
   }
```

```
}

int main()
{
   printf("\nAdding Nodes to the beginning of LinkedList \n");
   AddAtBeg(10);
   AddAtBeg(20);
   AddAtBeg(30);
   AddAtBeg(50);
   DisplayNode();
   printf("\n Deleting Nodes from the beginning of LinkedList \n");
   DelAtbeg();
   DelAtbeg();
   DelAtbeg();
   DelAtbeg();
   DelAtbeg();
   DisplayNode();
   return 0;
}
```

OUTPUT:



```
 Adding Node 10 to the begining of the Linked List

 Adding Node 20 to the begining of the Linked List

 Adding Node 30 to the begining of the Linked List

 Adding Node 50 to the begining of the Linked List

Contents of LinkedList are
50 --> 30 --> 20 --> 10 -->
 Deleting Nodes from the beginning of LinkedList

Node 50 is deleted

Node 30 is deleted

Node 20 is deleted

Node 10 is deleted

 Linked List Empty

Contents of LinkedList are

Empty Linked List
```

Follow Up:

1) Write a program to perform following operations on a singly linked list.

a) Count number of nodes b) Concatenation of two linked list c) Sort the linked list (Use Bubble sort)

Experiment 6:
Write a program to illustrate memory allocation to files based on their size using singly linked list. Program must support the following operations on a singly linked list.
a. Program to create and reverse a Singly linked list and display the same.
Problem Solution:

The given C program implements the following operations on a **singly linked list**:

1. **Creation of a linked list**.
2. **Reversal of the linked list**.
3. **Display of the linked list contents**.

1. Create a Node and Append to the List

Function: Create(int num)

1. Allocate memory for a new node.
2. Assign num to the info field of the node.
3. Set the link of the new node to NULL.
4. Check if the linked list is empty (Head == NULL):
   - If yes, set Head to the new node.
5. If the list is not empty:
   - Traverse to the last node (CurrPtr->link != NULL).
   - Update the link of the last node to point to the new node.

2. Reverse the Linked List

Function: ReverseList()

1. Initialize:
   - Prev to NULL.
   - CurrPtr to Head.
   - Next to temporarily store the next node.
2. Traverse the list:
   - Store the link of the current node in Next (Next = CurrPtr->link).
   - Update the link of the current node to point to Prev (CurrPtr->link = Prev).
   - Move Prev to the current node (Prev = CurrPtr).
   - Move CurrPtr to the next node (CurrPtr = Next).
3. After traversal, set Head = Prev to update the head of the reversed list.

3. Display the Contents of the Linked List

Function: DisplayNode()

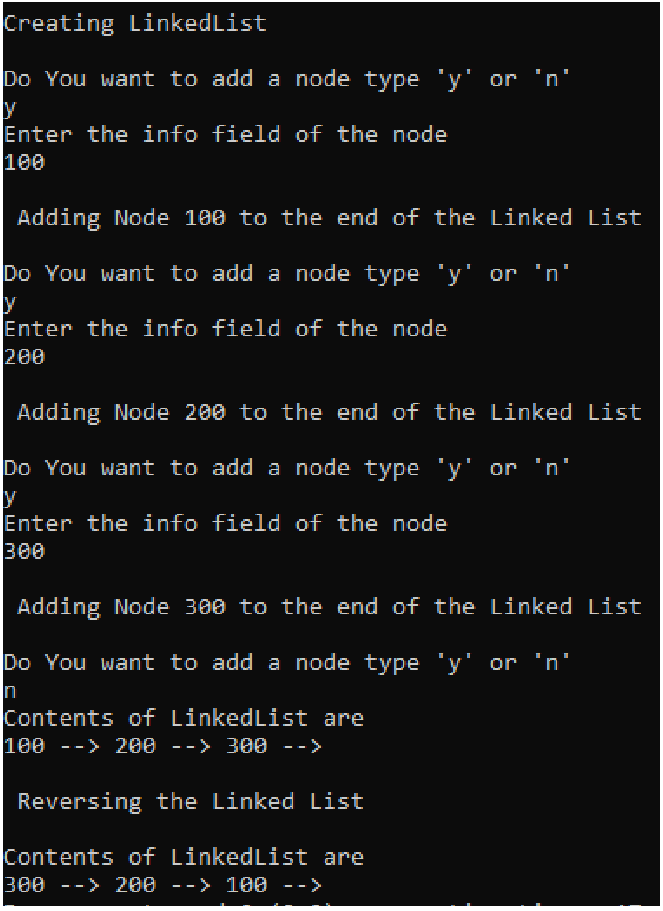
1. Check if the linked list is empty (Head == NULL):

  - If yes, print "Empty Linked List."
2. Otherwise, traverse the list:
    - Start at Head and print the info of each node.
    - Move to the next node using CurrPtr = CurrPtr->link until CurrPtr is NULL.

```
struct node
{
   int info;
   struct node *link;
};
typedef struct node NODE;

NODE *Head = NULL;

void ReverseList()
{
   NODE *CurrPtr = Head, *Prev = NULL, *Next;
   // Traverse all the nodes of Linked List
   while (CurrPtr != NULL)
   {

      // Store next
      Next = CurrPtr->link;

      // Reverse current node's next pointer
      CurrPtr->link = Prev;

      // Move pointers one position ahead
      Prev = CurrPtr;
      CurrPtr = Next;
   }
   // Assign the Head of the Reversed List
   Head = Prev;
}
void Create(int num)
{
   NODE *temp, *CurrPtr;
   temp = (NODE*)malloc(sizeof(NODE));
   temp->info = num;
   temp->link = NULL;
   printf("\n Adding Node %d to the end of the Linked List\n",temp->info);
   if(Head == NULL)
      Head = temp;
   else
   {
      CurrPtr = Head;
      while(CurrPtr->link != NULL)
      {
         CurrPtr = CurrPtr->link;
      }
      CurrPtr->link = temp;
   }
}
void DisplayNode()
{
   printf("\nContents of LinkedList are\n");
   NODE *CurrPtr = Head;
   if(CurrPtr == NULL)
   {
      printf("\nEmpty Linked List\n");
   }
   else
   {
      while(CurrPtr != NULL)
      {
      printf("%d --> ",CurrPtr->info);
      CurrPtr = CurrPtr->link;
      }
   }
}
int main()
{
   char ch = 'y'; int num;
   printf("\nCreating LinkedList \n");
   printf("\nDo You want to add a node type 'y' or 'n'\n");
   ch=getche();
   while(ch!='n')
   {
     printf("\nEnter the info field of the node \n");
     scanf("%d",&num);
     Create(num);
     printf("\nDo You want to add a node type 'y' or 'n'\n");
     ch=getche();
   }
   DisplayNode();
   printf("\n\n Reversing the Linked List\n");
   ReverseList();
```

```
    DisplayNode();

    return 0;
}
```

Output:



```
Creating LinkedList

Do You want to add a node type 'y' or 'n'
y
Enter the info field of the node
100

 Adding Node 100 to the end of the Linked List

Do You want to add a node type 'y' or 'n'
y
Enter the info field of the node
200

 Adding Node 200 to the end of the Linked List

Do You want to add a node type 'y' or 'n'
y
Enter the info field of the node
300

 Adding Node 300 to the end of the Linked List

Do You want to add a node type 'y' or 'n'
n
Contents of LinkedList are
100 --> 200 --> 300 -->

 Reversing the Linked List

Contents of LinkedList are
300 --> 200 --> 100 -->
```

b. Program to search a given element in a given Singly linked list
Problem Solution:

The provided C program implements the following operations for a singly linked list:

1. **Create a new linked list** or add nodes to the existing list.
2. **Display the contents of the linked list**.
3. **Search for a specific element** in the list.

1. Search for a Specific Element

Function: SearchList(int Key)

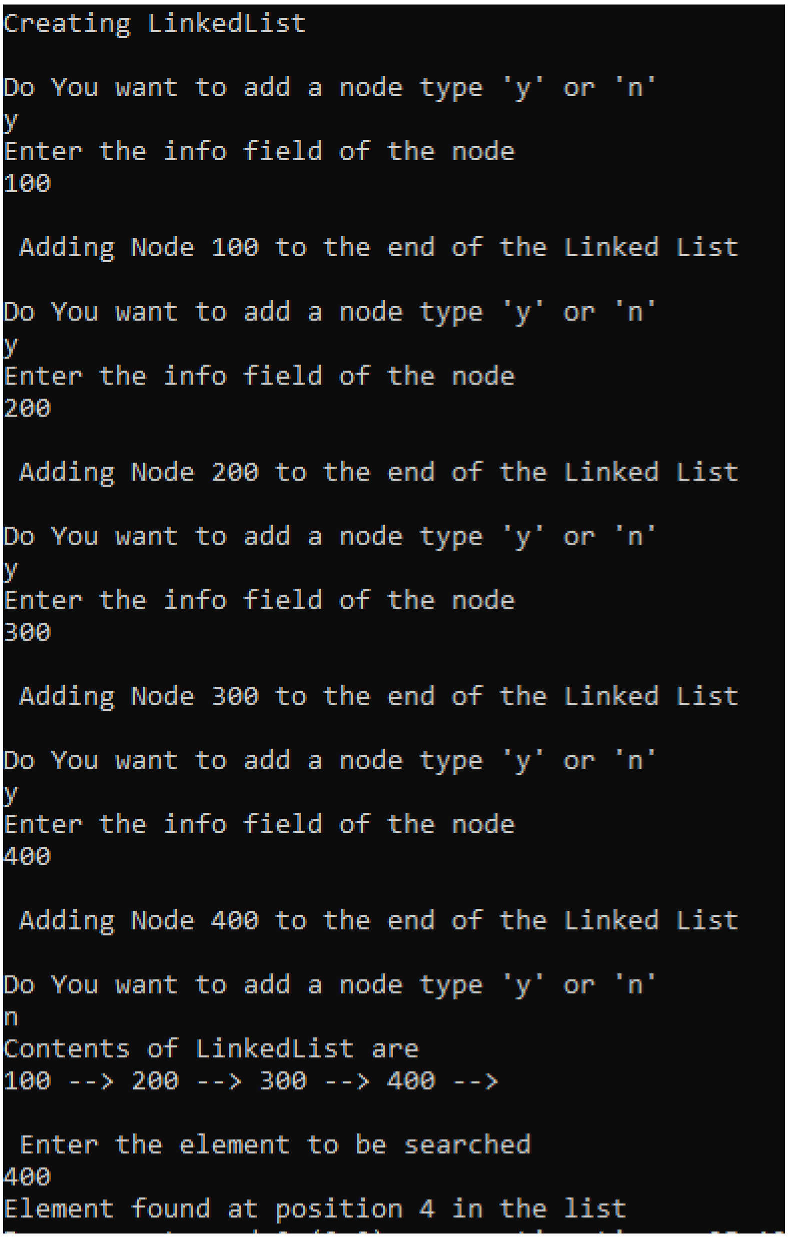
1. **Input**: An integer Key to search for.
2. Initialize:
   - CurrPtr to Head.
   - A counter i to track the position (starting at 1).

```
        o  A flag flag set to 0.
   3. Traverse the list:
        o  If CurrPtr->info == Key:
              ▪  Print "Element found at position i".
              ▪  Set flag = 1 and break.
        o  Otherwise, move to the next node and increment i.
   4. If the end of the list is reached and flag == 0:
        o  Print "Element not found in the list".
```

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

struct node
{
   int info;
   struct node *link;
};
typedef struct node NODE;

NODE *Head = NULL;

void SearchList(int Key)
{
   NODE * CurrPtr = Head;
   int i=1,flag=0;
   while(CurrPtr != NULL)
   {
      if(CurrPtr->info == Key)
      {
         printf("Element found at position %d in the list",i);
         flag = 1;
         break;
      }
      CurrPtr=CurrPtr->link;
      i++;
   }
   if(flag == 0)
      printf("Element not found in the list");
}
void Create(int num)
{
   NODE *temp, *CurrPtr;
   temp = (NODE*)malloc(sizeof(NODE));
   temp->info = num;
   temp->link = NULL;
```

```
    printf("\n Adding Node %d to the end of the Linked List\n",temp->info);
    if(Head == NULL)
        Head = temp;
    else
    {
        CurrPtr = Head;
        while(CurrPtr->link != NULL)
        {
            CurrPtr = CurrPtr->link;
        }
        CurrPtr->link = temp;
    }
}
void DisplayNode()
{
    printf("\nContents of LinkedList are\n");
    NODE *CurrPtr = Head;
    if(CurrPtr == NULL)
    {
        printf("\nEmpty Linked List\n");
    }
    else
    {
        while(CurrPtr != NULL)
        {
        printf("%d --> ",CurrPtr->info);
        CurrPtr = CurrPtr->link;
        }
    }
}

int main()
{
    char ch = 'y'; int num;
    printf("\nCreating LinkedList \n");
    printf("\nDo You want to add a node type 'y' or 'n'\n");
    ch=getche();
    while(ch!='n')
    {
       printf("\nEnter the info field of the node \n");
       scanf("%d",&num);
       Create(num);
       printf("\nDo You want to add a node type 'y' or 'n'\n");
       ch=getche();
    }
    DisplayNode();
    printf("\n\n Enter the element to be searched \n");
    scanf("%d",&num);
    SearchList(num);
    return 0;
}
```

OUTPUT:



```
Creating LinkedList

Do You want to add a node type 'y' or 'n'
y
Enter the info field of the node
100

 Adding Node 100 to the end of the Linked List

Do You want to add a node type 'y' or 'n'
y
Enter the info field of the node
200

 Adding Node 200 to the end of the Linked List

Do You want to add a node type 'y' or 'n'
y
Enter the info field of the node
300

 Adding Node 300 to the end of the Linked List

Do You want to add a node type 'y' or 'n'
y
Enter the info field of the node
400

 Adding Node 400 to the end of the Linked List

Do You want to add a node type 'y' or 'n'
n
Contents of LinkedList are
100 --> 200 --> 300 --> 400 -->

 Enter the element to be searched
400
Element found at position 4 in the list
```

Follow Up:

1) Write a program to perform following operations on a singly linked list.

a) Maximum element in the linked list b) Minimum element in the linked list c) Sum of all elements of the linked list

Experiment 7:

a) Write a program to manage integer data using a **doubly linked list**. The program should allow users to perform the following operations:

1. **Insert at the beginning**: Add an integer value at the beginning of the doubly linked list and **Delete at the beginning**: Remove a node at the beginning of the doubly linked list.

Problem Solution:

The provided program implements a **doubly linked list** that supports the following operations:

1. **Add at the Beginning**: Inserts a node containing an integer at the beginning of the list.
2. **Delete at the Beginning**: Deletes the node at the beginning of the list and frees the memory.
3. **Display Nodes**: Traverses and displays the content of the list.

1. AddAtBeg(int item):

- Allocates memory for a new node.
- Assigns the provided item to the info field.
- Links the new node as the first node:
    - Sets FORW of the new node to point to the current Start.
    - Updates BACK of the current Start node (if it exists) to the new node.
    - Updates Start to the new node.

2. DelAtBeg():

- Deletes the first node of the list:
    - If the list is empty, it prints a message.
    - If the list has only one node, it frees the node and sets Start to NULL.
    - Otherwise:
        - Updates Start to point to the second node.
        - Sets the BACK pointer of the new Start to NULL.
        - Frees the memory of the removed node.

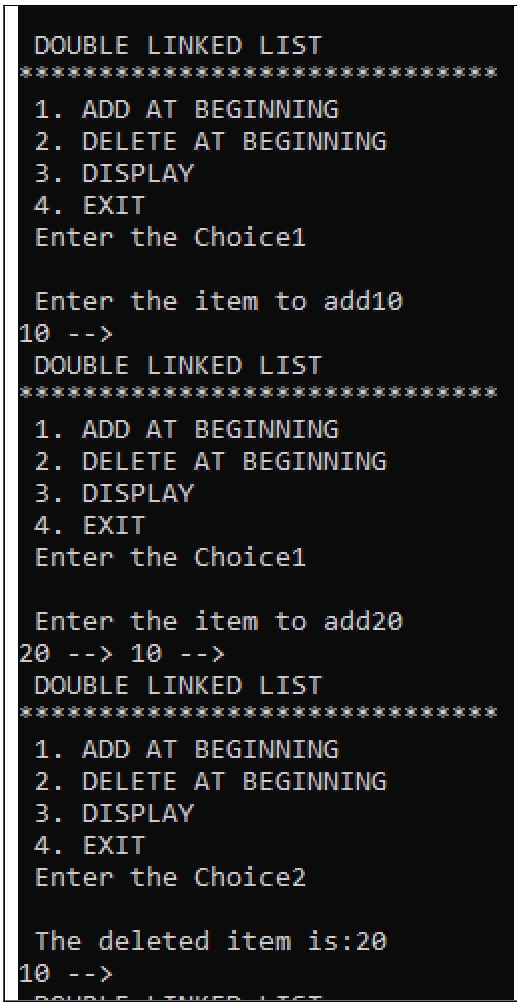
3. DisplayNode():

- Traverses the list from Start and prints the info field of each node. If the list is empty, it prints an appropriate message.

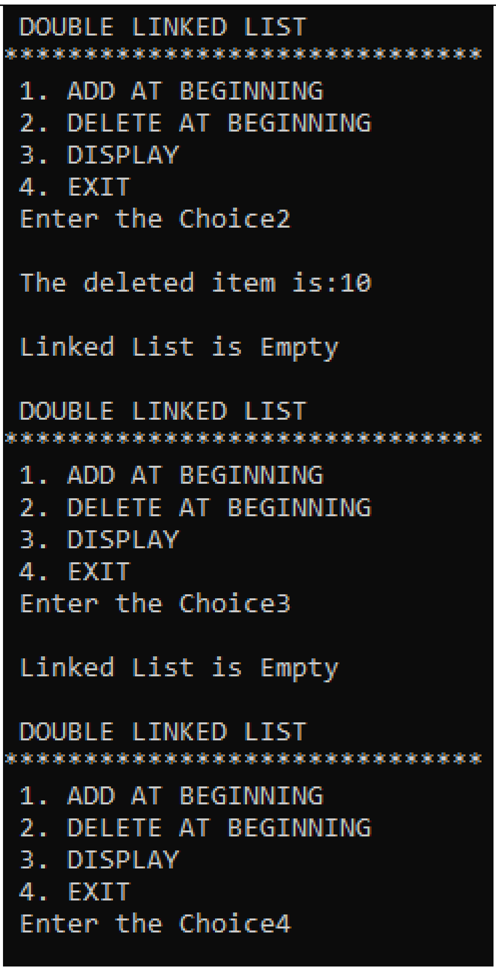
```
#include <stdio.h>
```

```
#include <stdlib.h>
struct Node
{   struct Node * BACK;
    int info;
    struct Node *FORW;
};
typedef struct Node NODE;
NODE *Start = NULL;
void AddAtBeg(int item)
{
    NODE *NewNode;
    NewNode = (NODE*)malloc(sizeof(NODE));
    NewNode->info=item;
    NewNode->BACK=NULL;
    NewNode->FORW=NULL;
    if(Start == NULL)
    {
        Start = NewNode;
        return;
    }
    NewNode->FORW=Start;
    Start->BACK=NewNode;
    Start = NewNode;
}
void DelAtbeg()
{
    NODE *Currptr = Start;
    if(Start == NULL)
    {
        printf("\n Linked List is Empty\n");
    }
    else if(Start->FORW==NULL)
    {
        Start=NULL;
        free(Currptr);
    }
    else
    {
        Start = Start->FORW;
        Start->BACK=NULL;
        free(Currptr);
        Currptr=NULL;
    }

}
void DisplayNode()
```

```
{
   NODE *Currptr = Start;
   if(Start == NULL)
   {
      printf("\n Linked List is Empty\n");
   }
   else
   {
      while(Currptr!=NULL)
      {
         printf("%d --> ",Currptr->info);
         Currptr = Currptr->FORW;
      }
   }
}
int main()
{
   int ch, item;
   while(1)
   {
      printf("\n DOUBLE LINKED LIST \n");
      printf("*******************************");
      printf("\n 1. ADD AT BEGINNING");
      printf("\n 2. DELETE AT BEGINNING");
      printf("\n 3. DISPLAY");
      printf("\n 4. EXIT");
      printf("\n Enter the Choice");
      scanf("%d",&ch);
      switch(ch)
      {
         case 1: printf("\n Enter the item to add");
               scanf("%d",&item);
               AddAtBeg(item);
               DisplayNode();
               break;
         case 2: if(Start!=NULL)
                  printf("\n The deleted item is:%d\n",Start->info);
               DelAtbeg();
               DisplayNode();
               break;
         case 3: DisplayNode();
               break;
         case 4: exit(0);

      }
   }
}
```



```
 DOUBLE LINKED LIST
*******************************
 1. ADD AT BEGINNING
 2. DELETE AT BEGINNING
 3. DISPLAY
 4. EXIT
 Enter the Choice1

 Enter the item to add10
10 -->
 DOUBLE LINKED LIST
*******************************
 1. ADD AT BEGINNING
 2. DELETE AT BEGINNING
 3. DISPLAY
 4. EXIT
 Enter the Choice1

 Enter the item to add20
20 --> 10 -->
 DOUBLE LINKED LIST
*******************************
 1. ADD AT BEGINNING
 2. DELETE AT BEGINNING
 3. DISPLAY
 4. EXIT
 Enter the Choice2

 The deleted item is:20
10 -->
```



```
 DOUBLE LINKED LIST
*******************************
 1. ADD AT BEGINNING
 2. DELETE AT BEGINNING
 3. DISPLAY
 4. EXIT
 Enter the Choice2

 The deleted item is:10

 Linked List is Empty

 DOUBLE LINKED LIST
*******************************
 1. ADD AT BEGINNING
 2. DELETE AT BEGINNING
 3. DISPLAY
 4. EXIT
 Enter the Choice3

 Linked List is Empty

 DOUBLE LINKED LIST
*******************************
 1. ADD AT BEGINNING
 2. DELETE AT BEGINNING
 3. DISPLAY
 4. EXIT
 Enter the Choice4
```

b) Write a program to manage integer data using a **doubly linked list**. The program should allow users to perform the following operations:

1. **Insert at the end**: Add an integer value at the end of the doubly linked list And **Delete at the end**: Remove a node at the end of the doubly linked list.

Problem Solution:

Algorithm for the Provided Doubly Linked List Operations

1. Add at End

1. Create a new node.
2. Set the new node's info to the given value and its FORW and BACK to NULL.
3. If the list is empty:
    - Set Start to the new node.
4. Otherwise:
    - Traverse to the last node in the list.

   - Update the FORW pointer of the last node to point to the new node.
   - Set the BACK pointer of the new node to the last node.

---

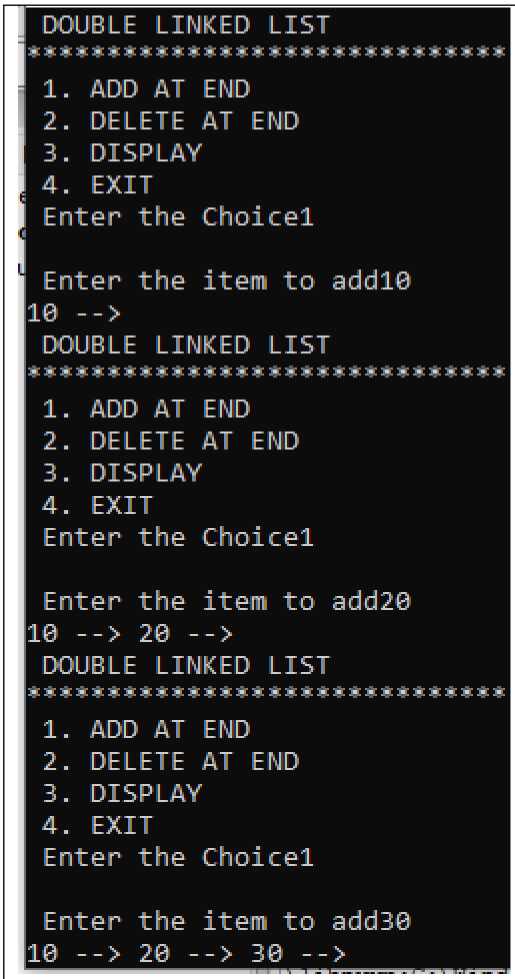
2. Delete at End

1. Check if the list is empty:
   - If yes, print "Linked List is Empty."
2. If the list has only one node:
   - Free the memory of the node and set Start to NULL.
3. Otherwise:
   - Traverse to the last node in the list.
   - Update the FORW pointer of the second-to-last node to NULL.
   - Free the memory of the last node.

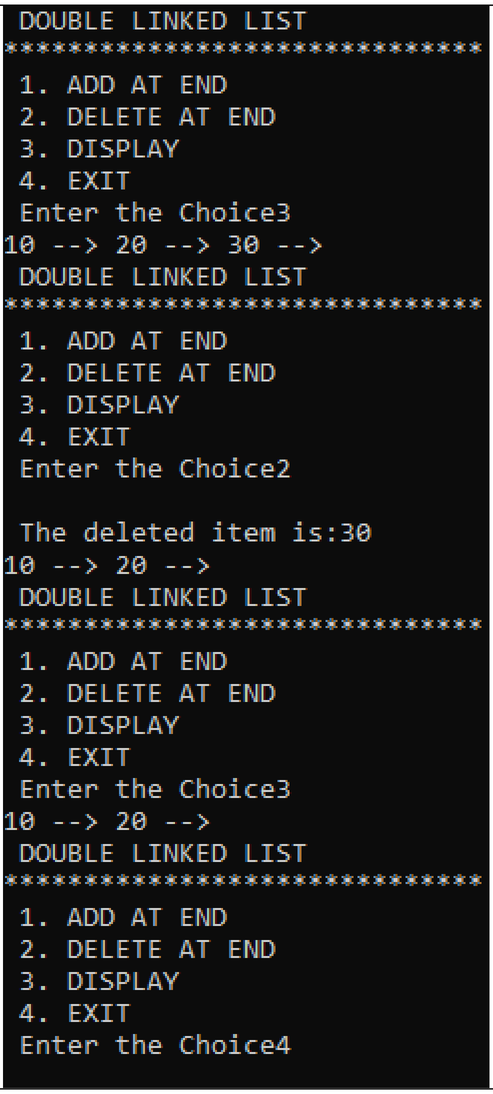
3. Display Nodes

1. Check if the list is empty:
   - If yes, print "Linked List is Empty."
2. Otherwise:
   - Start from Start and traverse the list.
   - Print the info of each node until the end of the list.

```
#include <stdio.h>
#include <stdlib.h>
struct Node
{   struct Node * BACK;
   int info;
   struct Node *FORW;
};
typedef struct Node NODE;
NODE *Start = NULL;

void AddAtEnd(int item)
{
   NODE *NewNode,*Currptr;
   NewNode = (NODE*)malloc(sizeof(NODE));
   NewNode->info=item;
   NewNode->BACK=NULL;
   NewNode->FORW=NULL;
   if(Start == NULL)
   {
      Start = NewNode;
      return;
   }
   else
   {
      Currptr=Start;
      while(Currptr->FORW!=NULL)
      {
         Currptr=Currptr->FORW;
      }
      Currptr->FORW=NewNode;
      NewNode->BACK=Currptr;
   }
}
void DelAtEnd()
{
   NODE *Currptr = Start;
   if(Start == NULL)
   {
      printf("\n Linked List is Empty\n");
   }
   else if(Start->FORW==NULL)
   {
     printf("\n The deleted item is:%d\n",Start->info);
      Start=NULL;
      free(Currptr);
   }
   else
   {
      while(Currptr->FORW!=NULL)
      {
         Currptr=Currptr->FORW;
      }
      printf("\n The deleted item is:%d\n", Currptr ->info);
      Currptr->BACK->FORW=NULL;
      free(Currptr);
      Currptr=NULL;
   }
}
void DisplayNode()
{
   NODE *Currptr = Start;
   if(Start == NULL)
   {
      printf("\n Linked List is Empty\n");
   }
   else
   {
      while(Currptr!=NULL)
      {
            printf("%d --> ",Currptr->info);
            Currptr = Currptr->FORW;
        }
    }
}
int main()
{
    int ch, item;
    while(1)
    {
        printf("\n DOUBLE LINKED LIST \n");
        printf("*****************************");
        printf("\n 1. ADD AT END");
        printf("\n 2. DELETE AT END");
        printf("\n 3. DISPLAY");
        printf("\n 4. EXIT");
        printf("\n Enter the Choice");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: printf("\n Enter the item to add");
                    scanf("%d",&item);
                    AddAtEnd(item);
                    DisplayNode();
                    break;
            case 2:
                    DelAtEnd();
                    DisplayNode();
                    break;
            case 3: DisplayNode();
                    break;
            case 4: exit(0);

        }

    }
}
```



```
 DOUBLE LINKED LIST
******************************
 1. ADD AT END
 2. DELETE AT END
 3. DISPLAY
 4. EXIT
 Enter the Choice1

 Enter the item to add10
10 -->
 DOUBLE LINKED LIST
******************************
 1. ADD AT END
 2. DELETE AT END
 3. DISPLAY
 4. EXIT
 Enter the Choice1

 Enter the item to add20
10 --> 20 -->
 DOUBLE LINKED LIST
******************************
 1. ADD AT END
 2. DELETE AT END
 3. DISPLAY
 4. EXIT
 Enter the Choice1

 Enter the item to add30
10 --> 20 --> 30 -->
```



```
 DOUBLE LINKED LIST
******************************
 1. ADD AT END
 2. DELETE AT END
 3. DISPLAY
 4. EXIT
 Enter the Choice3
10 --> 20 --> 30 -->
 DOUBLE LINKED LIST
******************************
 1. ADD AT END
 2. DELETE AT END
 3. DISPLAY
 4. EXIT
 Enter the Choice2

 The deleted item is:30
10 --> 20 -->
 DOUBLE LINKED LIST
******************************
 1. ADD AT END
 2. DELETE AT END
 3. DISPLAY
 4. EXIT
 Enter the Choice3
10 --> 20 -->
 DOUBLE LINKED LIST
******************************
 1. ADD AT END
 2. DELETE AT END
 3. DISPLAY
 4. EXIT
 Enter the Choice4
```

FOLLOW UP Questions:

Write a program to manage integer data using a **doubly linked list**. The program should allow users to perform the following operations:

1. **Insert after a value**: Insert an integer value after a specified value in the list.
2. **Search for a value**: Search for an integer value in the list and display its position (1-based index).
3. **Display the list**: Traverse and display all integer values in the doubly linked list in reverse order.

Experiment 8:
a. Implement a stack using a singly linked list in C. Design functions for push (insert), pop (delete), and display operations. Write a program to simulate the backtracking functionality in a web browser, where the user can visit new web pages (push) or go back to the previous page (pop). Test your program with at least five pages, displaying the stack after each operation.
Problem Solution:

Algorithm for Stack Implementation Using Linked List

1. Define Node Structure
2. Initialize Stack
3. Push Operation: (AddAtBeg)
4. Pop Operation (DelAtBeg)
5. Display Operation (DisplayNode)

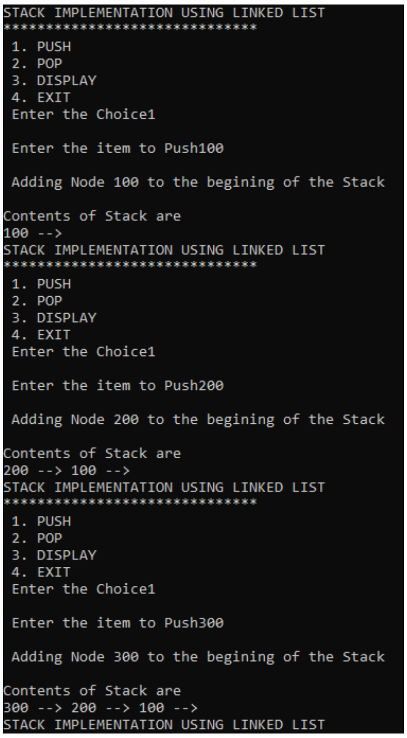
Main Function:

6. Display a menu with options for:
    1. **Push**: Call the push operation.
    2. **Pop**: Call the pop operation.
    3. **Display**: Call the display operation.
    4. **Exit**: Terminate the program.
7. Execute user-selected operations in a loop until "Exit" is chosen.

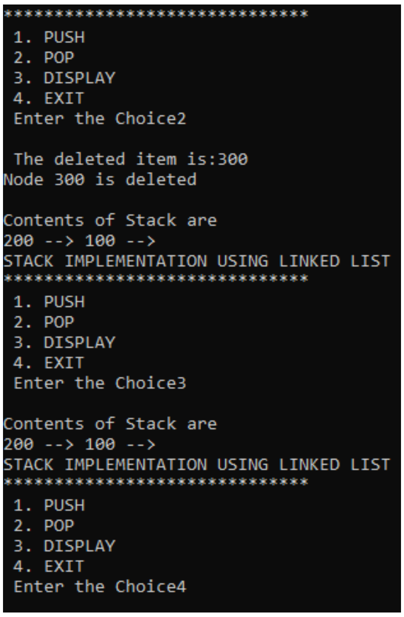
```
#include <stdio.h>
#include <stdlib.h>

struct node
{
   int info;
   struct node *link;
};
typedef struct node NODE;

NODE *Head = NULL;
void AddAtBeg(int num)
{
   NODE *temp;
   temp = (NODE*)malloc(sizeof(NODE));
   temp->info = num;
   temp->link = NULL;
   printf("\n Adding Node %d to the begining of the Stack\n",temp->info);
   if(Head == NULL)
      Head = temp;
   else
```

```
    {
        temp->link = Head;
        Head = temp;
    }
}
void DelAtbeg()
{
    NODE  *NextPtr;

    if(Head == NULL)
    {
        printf("\n Stack is Empty\n");
        return;
    }
    else
    {
        NextPtr = Head;
        printf("\nNode %d is deleted\n",NextPtr->info);
        Head = Head->link;
        free(NextPtr);
    }
}
void DisplayNode()
{
    printf("\nContents of Stack are\n");
    NODE *CurrPtr = Head;
    if(CurrPtr == NULL)
    {
        printf("\nStack is Empty \n");
    }
    else
    {
        while(CurrPtr != NULL)
        {
        printf("%d --> ",CurrPtr->info);
        CurrPtr = CurrPtr->link;
        }
    }
}

int main()
{
    int ch, item;
    while(1)
    {
        printf("\nSTACK IMPLEMENTATION USING LINKED LIST \n");
        printf("******************************");
        printf("\n 1. PUSH");
        printf("\n 2. POP");
        printf("\n 3. DISPLAY");
        printf("\n 4. EXIT");
        printf("\n Enter the Choice");
        scanf("%d",&ch);
        switch(ch)
        {
          case 1: printf("\n Enter the item to Push");
                  scanf("%d",&item);
                  AddAtBeg(item);
                  DisplayNode();
                  break;
          case 2: if(Head!=NULL)
                     printf("\n The deleted item is:%d",Head->info);
                  DelAtbeg();
                  DisplayNode();
                  break;
          case 3: DisplayNode();
                  break;
          case 4: exit(0);
        }
      }
    }
```

OUTPUT:



```
STACK IMPLEMENTATION USING LINKED LIST
******************************
 1. PUSH
 2. POP
 3. DISPLAY
 4. EXIT
 Enter the Choice1

 Enter the item to Push100

 Adding Node 100 to the begining of the Stack

Contents of Stack are
100 -->
STACK IMPLEMENTATION USING LINKED LIST
******************************
 1. PUSH
 2. POP
 3. DISPLAY
 4. EXIT
 Enter the Choice1

 Enter the item to Push200

 Adding Node 200 to the begining of the Stack

Contents of Stack are
200 --> 100 -->
STACK IMPLEMENTATION USING LINKED LIST
******************************
 1. PUSH
 2. POP
 3. DISPLAY
 4. EXIT
 Enter the Choice1

 Enter the item to Push300

 Adding Node 300 to the begining of the Stack

Contents of Stack are
300 --> 200 --> 100 -->
STACK IMPLEMENTATION USING LINKED LIST
```



```
******************************
 1. PUSH
 2. POP
 3. DISPLAY
 4. EXIT
 Enter the Choice2

 The deleted item is:300
Node 300 is deleted

Contents of Stack are
200 --> 100 -->
STACK IMPLEMENTATION USING LINKED LIST
******************************
 1. PUSH
 2. POP
 3. DISPLAY
 4. EXIT
 Enter the Choice3

Contents of Stack are
200 --> 100 -->
STACK IMPLEMENTATION USING LINKED LIST
******************************
 1. PUSH
 2. POP
 3. DISPLAY
 4. EXIT
 Enter the Choice4
```

b. Implement a queue using a singly linked list in C. Design functions for enqueue (insert), dequeue (delete), and display operations. Write a program to simulate the customer service system at a bank where customers join the queue for service. Test your program with at least five customers, displaying the queue after each operation.

Problem Solution:

Algorithm: Queue Implementation Using Singly Linked List

1. **Adding at Rear (Enqueue):**
   - **AddatEnd()** function appends a new node to the end of the list.
   - It checks if the Head is NULL (empty list) and updates the Head.
   - If not empty, it traverses to the end of the list and links the new node.

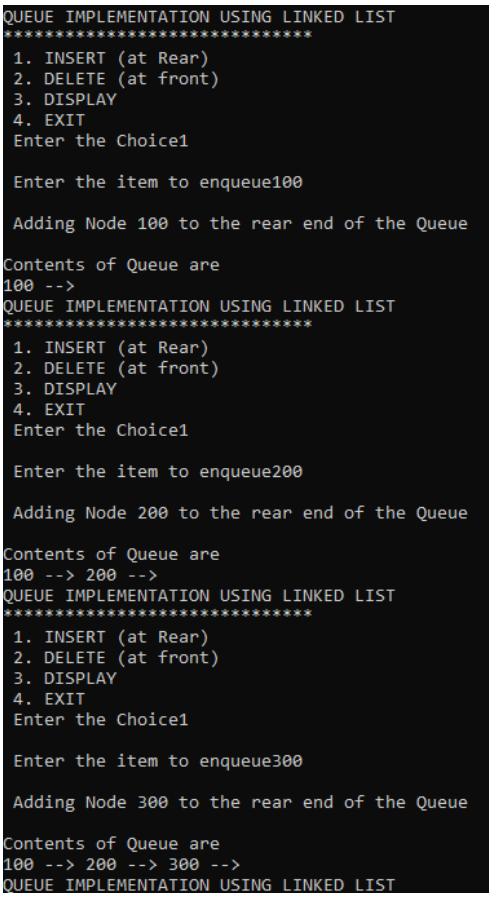
2. **Deleting at Front (Dequeue):**
   - **DelAtbeg()** function deletes the front node.
   - It checks if the Head is NULL (empty list) to prevent deletion from an empty queue.
   - Deletes the node, updates the Head pointer, and frees the memory.

3. **Displaying the Queue:**
   - **DisplayNode()** function traverses and prints all elements in the list.

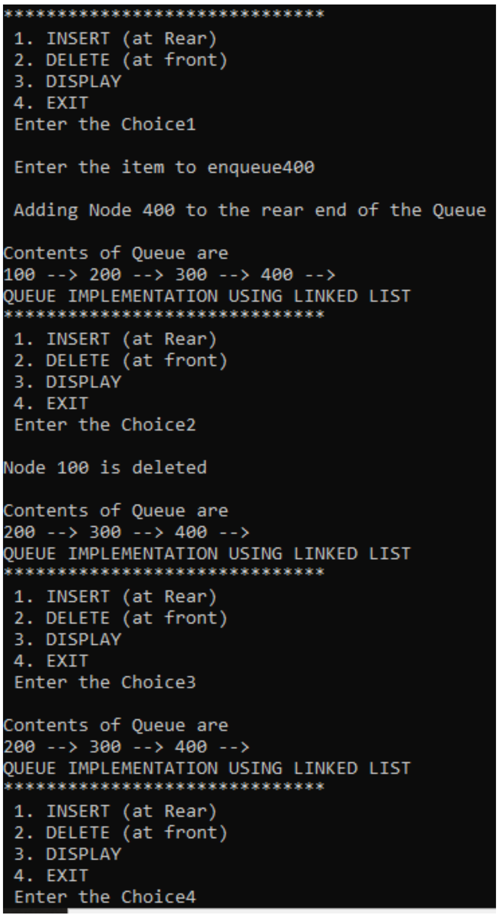
4. **Menu-Driven Interface in main:**
   - Options for enqueue (insert at rear), dequeue (delete from front), display, and exit.
   - Keeps looping until the user exits.

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

struct node
{
   int info;
   struct node *link;
};
typedef struct node NODE;
NODE *Head = NULL;

void AddatEnd(int num)
{
   NODE *temp, *CurrPtr;
   temp = (NODE*)malloc(sizeof(NODE));
   temp->info = num;
   temp->link = NULL;
   printf("\n Adding Node %d to the rear end of the Queue\n",temp->info);
   if(Head == NULL)
      Head = temp;
   else
   {
      CurrPtr = Head;
      while(CurrPtr->link != NULL)
      {
         CurrPtr = CurrPtr->link;
      }
      CurrPtr->link = temp;
   }
}
void DelAtbeg()
{
   NODE  *NextPtr;

   if(Head == NULL)
   {
      printf("\n Queue is Empty\n");
      return;
   }
   else
   {
      NextPtr = Head;
      printf("\nNode %d is deleted\n",NextPtr->info);
      Head = Head->link;
      free(NextPtr);
   }
}
void DisplayNode()
{
   printf("\nContents of Queue are\n");
   NODE *CurrPtr = Head;
   if(CurrPtr == NULL)
   {
      printf("\nEmpty Queue\n");
   }
   else
   {
      while(CurrPtr != NULL)
      {
      printf("%d --> ",CurrPtr->info);
      CurrPtr = CurrPtr->link;
      }
   }
}
```

```
int main()
{
   int ch, item;
   while(1)
   {
      printf("\nQUEUE IMPLEMENTATION USING LINKED LIST \n");
      printf("******************************");
      printf("\n 1. INSERT (at Rear)");
      printf("\n 2. DELETE (at front)");
      printf("\n 3. DISPLAY");
      printf("\n 4. EXIT");
      printf("\n Enter the Choice");
      scanf("%d",&ch);
      switch(ch)
      {
         case 1: printf("\n Enter the item to enqueue");
               scanf("%d",&item);
               AddatEnd(item);
               DisplayNode();
               break;
         case 2: DelAtbeg();
               DisplayNode();
               break;
         case 3: DisplayNode();
               break;
         case 4: exit(0);
      }
   }
}
OUTPUT:
```



```
QUEUE IMPLEMENTATION USING LINKED LIST
******************************
 1. INSERT (at Rear)
 2. DELETE (at front)
 3. DISPLAY
 4. EXIT
 Enter the Choice1

 Enter the item to enqueue100

 Adding Node 100 to the rear end of the Queue

Contents of Queue are
100 -->
QUEUE IMPLEMENTATION USING LINKED LIST
******************************
 1. INSERT (at Rear)
 2. DELETE (at front)
 3. DISPLAY
 4. EXIT
 Enter the Choice1

 Enter the item to enqueue200

 Adding Node 200 to the rear end of the Queue

Contents of Queue are
100 --> 200 -->
QUEUE IMPLEMENTATION USING LINKED LIST
******************************
 1. INSERT (at Rear)
 2. DELETE (at front)
 3. DISPLAY
 4. EXIT
 Enter the Choice1

 Enter the item to enqueue300

 Adding Node 300 to the rear end of the Queue

Contents of Queue are
100 --> 200 --> 300 -->
QUEUE IMPLEMENTATION USING LINKED LIST
```



```
******************************
 1. INSERT (at Rear)
 2. DELETE (at front)
 3. DISPLAY
 4. EXIT
 Enter the Choice1

 Enter the item to enqueue400

 Adding Node 400 to the rear end of the Queue

Contents of Queue are
100 --> 200 --> 300 --> 400 -->
QUEUE IMPLEMENTATION USING LINKED LIST
******************************
 1. INSERT (at Rear)
 2. DELETE (at front)
 3. DISPLAY
 4. EXIT
 Enter the Choice2

Node 100 is deleted

Contents of Queue are
200 --> 300 --> 400 -->
QUEUE IMPLEMENTATION USING LINKED LIST
******************************
 1. INSERT (at Rear)
 2. DELETE (at front)
 3. DISPLAY
 4. EXIT
 Enter the Choice3

Contents of Queue are
200 --> 300 --> 400 -->
QUEUE IMPLEMENTATION USING LINKED LIST
******************************
 1. INSERT (at Rear)
 2. DELETE (at front)
 3. DISPLAY
 4. EXIT
 Enter the Choice4
```

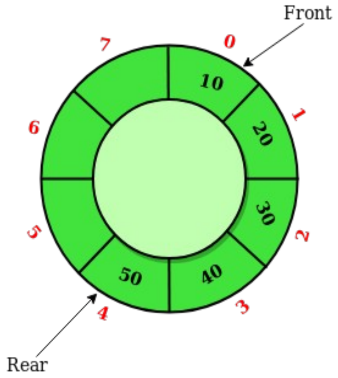
FOLLOW UP:
1) Modify the Queue implementation using single link list with the following requirements
a. Create FRONT and REAR pointers (Initially NULL) b. Use REAR pointer to insert node to the end of the queue(if REAR is NULL, New node added will be pointed by both REAR and FRONT)
c. USE FRONT pointer to delete node from the beginning of the queue.(while deletion if FRONT and REAR pointer are same , they should be made NULL) d. display nodes of the queue.

Experiment 9:

Write a program to demonstrate round robin processor scheduling using circular queue (array implementation) with suitable inputs. Program should have options to add, remove and display elements of the queue.

Problem Solution:

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



Operations on Circular Queue:

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

1. Initialization:

- **Input:**
  - Define an array arr of size MAX = 4 to hold the queue elements.
  - Initialize two variables:
    - front = -1 (to indicate an empty queue).
    - rear = -1 (to indicate an empty queue).

2. Add an Element to the Queue (addCq):

- **Input:** An element item to be added.
- **Process:**
  1. **Check if the queue is full**:
     - If (rear == MAX-1 && front == 0) or (rear + 1 == front), the queue is full. Print an error message and return.
  2. **Update the rear pointer**:
     - If rear == MAX-1, set rear = 0 (circular behavior).
     - Otherwise, increment rear by 1.
  3. **Add the item to the queue**:
     - Set arr[rear] = item.
  4. **Update the front pointer**:
     - If front == -1, set front = 0 to indicate that the queue is no longer empty.

- **Output:** Item is added to the queue.

3. Delete an Element from the Queue (delCq):

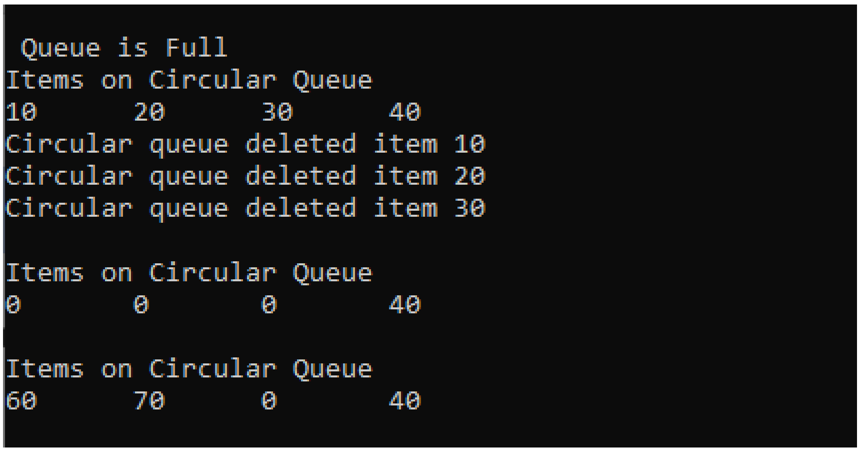
- **Output:** The item removed from the queue.
- **Process:**
    1. **Check if the queue is empty**:
        - If front == -1, the queue is empty. Print an error message and return -1.
    2. **Store the item at the front**:
        - Store data = arr[front].
    3. **Update the queue**:
        - Set arr[front] = 0 (clear the deleted slot).
    4. **Check if the queue becomes empty**:
        - If front == rear, the queue is now empty. Set front = -1 and rear = -1.
    5. **Move the front pointer**:
        - If front == MAX-1, set front = 0 (circular behavior).
        - Otherwise, increment front by 1.
- **Output:** Return the data that was deleted.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 4
int arr[MAX];
int rear = -1, front = -1;

void addCq(int item)
{
   if((rear == MAX-1 && front ==0) || rear + 1 == front)
   {
      printf("\n Queue is Full");
      return;
   }
   if(rear == MAX -1)
      rear = 0;
   else
      rear++ ;
   arr[rear] = item;

   if(front == -1)
      front = 0;

}

int delCq()
{
   int data;
```

```
    if(front == -1)
    {
        printf("\n Queue is Empty");
        return -1;
    }
    data = arr[front];
    arr[front]=0;
    if(front == rear)
    {
        front = -1;
        rear = -1;
    }

    else
    {
        if(front == MAX-1)
            front = 0;
        else
            front++;
    }
    return data;
}
void display()
{
    printf("\n");

     if (front == -1)
    {
        printf("\nQueue is Empty");
        return;
    }
    printf("\nElements in Circular Queue are: ");
    if (rear >= front)
    {
        for (int i = front; i <= rear; i++)
            printf("%d ",arr[i]);
    }
    else
    {
        for (int i = front; i < MAX; i++)
            printf("%d ", arr[i]);

        for (int i = 0; i <= rear; i++)
            printf("%d ", arr[i]);
    }
    printf("\n");
```

```
}
int main()
{
   addCq(10);
   addCq(20);
   addCq(30);
   addCq(40);
   addCq(50);
   display();
   printf("Circular queue deleted item %d \t\n",delCq());
   printf("Circular queue deleted item %d \t\n",delCq());
   printf("Circular queue deleted item %d \t\n",delCq());
   display();
   addCq(60);
   addCq(70);
   display();
   return 0;
}
```

OUTPUT:



```
 Queue is Full
Items on Circular Queue
10      20      30      40
Circular queue deleted item 10
Circular queue deleted item 20
Circular queue deleted item 30

Items on Circular Queue
0       0       0       40

Items on Circular Queue
60      70      0       40
```

Follow Up:

1) Write a program to implement circular queue using structures.

2) How a stack of queues can be implemented?

Experiment 10:
a. Construct a binary tree, Write programs for in order, preorder and post order traversal of trees.
Problem Solution:

Algorithm

1. Define the Node Structure

- Create a structure `node` with:
  - `info` (to store data).
  - `left` (pointer to the left child).
  - `right` (pointer to the right child).

2. Initialize Root

- Start with `root = NULL`.

3. Create the Tree (`create_tree` Function)

1. If the current node is not `NULL`:
   - Ask the user to input data for the current node.
   - Store the data in `ptr->info`.
2. Ask the user if they want to create a left child:
   - If "yes," allocate memory for the left child and recursively call `create_tree` for the left child.
   - If "no," set `ptr->left = NULL`.
3. Ask the user if they want to create a right child:
   - If "yes," allocate memory for the right child and recursively call `create_tree` for the right child.
   - If "no," set `ptr->right = NULL`.

4. Traversals

- **Preorder:** Visit the current node → Traverse the left child → Traverse the right child.
- **Postorder:** Traverse the left child → Traverse the right child → Visit the current node.
- **Inorder:** Traverse the left child → Visit the current node → Traverse the right child.

5. Display Tree Structure (`disp` Function)

1. Recursively display the right child with increased level.
2. Print the current node with indentation based on level.
3. Recursively display the left child with increased level.

6. Main Function

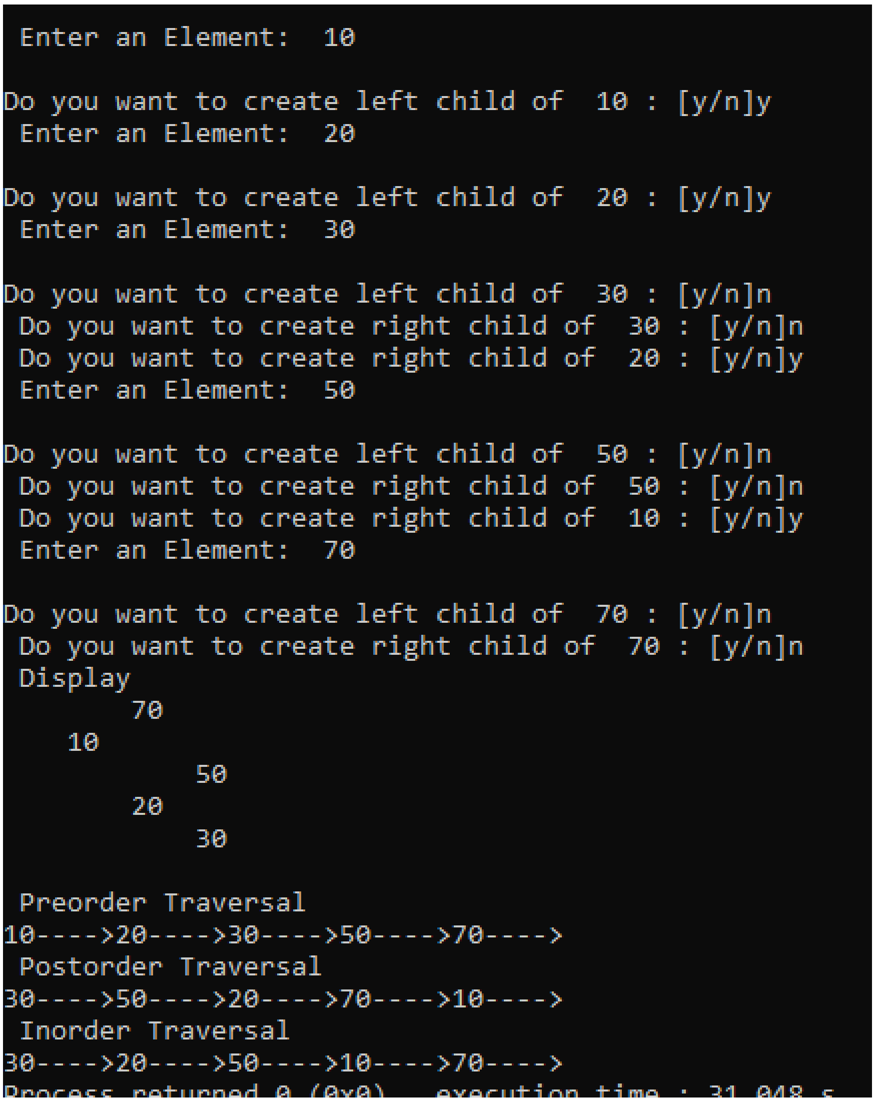
1. Allocate memory for `root` and initialize its child pointers to `NULL`.
2. Call `create_tree` to build the tree starting from `root`.
3. Call `disp` to display the tree structure.

4. Perform the following traversals and display the results:
   - Preorder.
   - Postorder.
   - Inorder.

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

struct node
{
   struct node *left;
   int info;
   struct node *right;
};
typedef struct node NODE;
NODE * root=NULL;
int flag=0;
void create_tree(NODE *ptr)
{
   NODE *newleft,*newright;
   int item;
   char ch;
   if(ptr!=NULL)
   {
      printf("\n Enter an Element:  ");
      scanf("%d",&item);
      ptr->info=item;
      printf("\nDo you want to create left child of  %d : [y/n]",ptr->info);
      ch=getche();
      if(ch=='Y'||ch=='y')
      {
         newleft = (NODE*)malloc(sizeof(NODE));
         ptr->left=newleft;
         create_tree(newleft);
      }
      else
      {
         ptr->left=NULL;
         create_tree(NULL);
      }
      printf("\n Do you want to create right child of  %d : [y/n]",ptr->info);
       ch=getche();
      if(ch=='Y'||ch=='y')
      {
         newright = (NODE*)malloc(sizeof(NODE));
         ptr->right=newright;
         create_tree(newright);
      }
      else
      {
         ptr->right=NULL;
         create_tree(NULL);
      }

   }
}
void preorder(NODE * ptr)
{
   if(ptr)
   {
      printf("%d---->",ptr->info);
      preorder(ptr->left);
      preorder(ptr->right);
   }
}
void postorder(NODE * ptr)
{
   if(ptr)
   {
      postorder(ptr->left);
      postorder(ptr->right);
      printf("%d---->",ptr->info);
   }
}
void Inorder(NODE * ptr)
{
   if(ptr)
   {
      Inorder(ptr->left);
      printf("%d---->",ptr->info);
      Inorder(ptr->right);
   }
}
void disp(NODE *ptr,int level)
{
   int i;
   if(ptr!=NULL)
   {
      disp(ptr->right,level+1);
      for(i=0;i<level;i++)
```

```
            printf("    ");
        printf("%2d\n",ptr->info);
        disp(ptr->left,level+1);
    }
}
int main()
{
    int l=0,item;
    root=NULL;
    NODE *root = (NODE*)malloc(sizeof(NODE));
    root->left = root->right = NULL;
    create_tree(root);
    printf("\n Display\n");
    disp(root,1);
    printf("\n Preorder Traversal\n");
    preorder(root);
    printf("\n Postorder Traversal\n");
    postorder(root);
    printf("\n Inorder Traversal\n");
    Inorder(root);
    return 0;
}
```

OUTPUT



```
 Enter an Element:  10

Do you want to create left child of  10 : [y/n]y
 Enter an Element:  20

Do you want to create left child of  20 : [y/n]y
 Enter an Element:  30

Do you want to create left child of  30 : [y/n]n
 Do you want to create right child of  30 : [y/n]n
 Do you want to create right child of  20 : [y/n]y
 Enter an Element:  50

Do you want to create left child of  50 : [y/n]n
 Do you want to create right child of  50 : [y/n]n
 Do you want to create right child of  10 : [y/n]y
 Enter an Element:  70

Do you want to create left child of  70 : [y/n]n
 Do you want to create right child of  70 : [y/n]n
 Display
        70
    10
            50
        20
            30

 Preorder Traversal
10---->20---->30---->50---->70---->
 Postorder Traversal
30---->50---->20---->70---->10---->
 Inorder Traversal
30---->20---->50---->10---->70---->
Process returned 0 (0x0)   execution time : 31.048 s
```

b. Write a program that implements a Binary Search Tree (BST) in C, including functions for insertion, preorder traversal, postorder traversal, inorder traversal, and a function to display the tree structure.

Problem Solution:

Algorithm:

1. Define the Node Structure

- Define a structure `node` to represent a tree node.
    - o `info`: stores the node's data.
    - o `left`: pointer to the left child.
    - o `right`: pointer to the right child.

2. Initialize the Tree

- Set the root of the tree to `NULL` initially.

3. Insert a Node (`create` function)

1. Allocate memory for a new node.
2. Set the `info` field of the node to the given value (`item`).
3. Set the `left` and `right` child pointers to `NULL`.
4. If the tree is empty (i.e., `root == NULL`):
   - Set the root to the new node.
5. Otherwise, find the appropriate position for the new node by:
   - Comparing `item` with the current node's `info`.
   - If `item` is smaller, move to the left child.
   - If `item` is larger, move to the right child.
6. Insert the new node in the correct position.

4. Traversals

1. **Preorder Traversal**:
   - Visit the current node.
   - Recursively visit the left subtree.
   - Recursively visit the right subtree.

2. **Postorder Traversal**:
   - Recursively visit the left subtree.
   - Recursively visit the right subtree.
   - Visit the current node.

3. **Inorder Traversal**:
   - Recursively visit the left subtree.
   - Visit the current node.
   - Recursively visit the right subtree.

5. Display Tree Structure (`disp` function)

1. Recursively print the right child with increased indentation.
2. Print the current node with the appropriate level of indentation.
3. Recursively print the left child with increased indentation.

6. Main Function

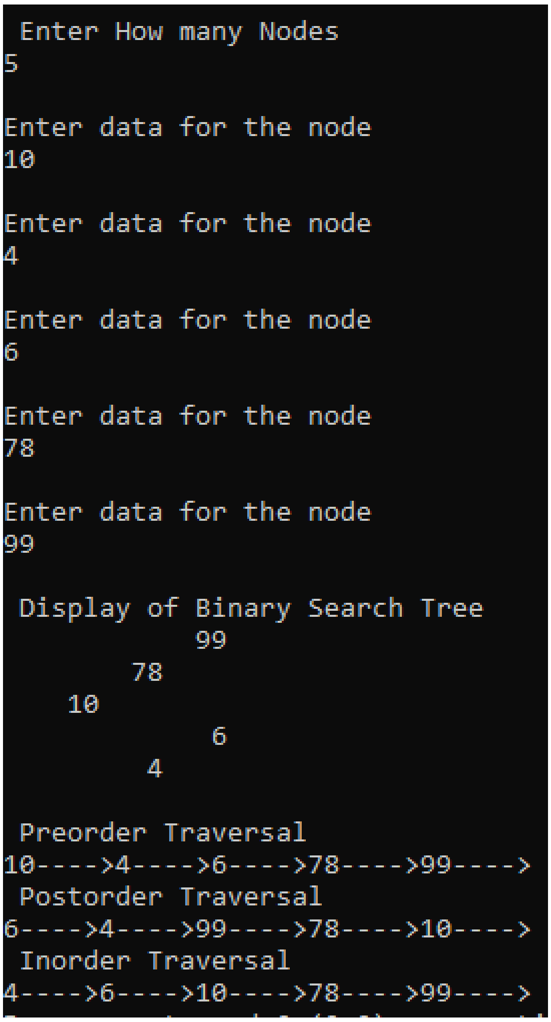
1. Ask the user how many nodes to insert.
2. Insert each node by calling the `create` function with the given data.
3. Call the `disp` function to display the tree.
4. Perform and display the tree traversals: preorder, postorder, and inorder.

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

```
struct node
{
   struct node *left;
   int info;
   struct node *right;
};
typedef struct node NODE;
NODE * root=NULL;
void create(int item)
{
   NODE* newnode, *currptr, *ptr;
   newnode = (NODE*)malloc(sizeof(NODE));
   newnode->info=item;
   newnode->left=NULL;
   newnode->right=NULL;
   if(root == NULL)
   {
      root = newnode;
   }
   else
   {
      currptr = root;
      while(currptr!=NULL)
      {
         ptr = currptr;
         currptr=(item>currptr->info)?currptr->right:currptr->left;
      }
      if(item<ptr->info)
         ptr->left=newnode;
      else
         ptr->right=newnode;
   }
}
void preorder(NODE * ptr)
{
   if(ptr)
   {
      printf("%d---->",ptr->info);
      preorder(ptr->left);
      preorder(ptr->right);
   }
}
void postorder(NODE * ptr)
{
   if(ptr)
   {
        postorder(ptr->left);
        postorder(ptr->right);
        printf("%d---->",ptr->info);
    }
}
void Inorder(NODE * ptr)
{
    if(ptr)
    {
        Inorder(ptr->left);
        printf("%d---->",ptr->info);
        Inorder(ptr->right);
    }
}
void disp(NODE *ptr,int level)
{
    int i;
    if(ptr!=NULL)
    {
        disp(ptr->right,level+1);
        for(i=0;i<level;i++)
            printf("    ");
        printf("%2d\n",ptr->info);
        disp(ptr->left,level+1);
    }
}
int main()
{
    int n, i,item;
    printf("\n Enter How many Nodes\n");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("\nEnter data for the node\n");
        scanf("%d",&item);
        create(item);
    }
    printf("\n Display of Binary Search Tree\n");
    disp(root,1);
    printf("\n Preorder Traversal\n");
    preorder(root);
    printf("\n Postorder Traversal\n");
    postorder(root);
    printf("\n Inorder Traversal\n");
    Inorder(root);
    return 0;
```

}

OUTPUT:



```
 Enter How many Nodes
5

Enter data for the node
10

Enter data for the node
4

Enter data for the node
6

Enter data for the node
78

Enter data for the node
99

 Display of Binary Search Tree
            99
        78
    10
             6
         4

 Preorder Traversal
10---->4---->6---->78---->99---->
 Postorder Traversal
6---->4---->99---->78---->10---->
 Inorder Traversal
4---->6---->10---->78---->99---->
```

Experiment 11:
a. Write a program to implement a linear search algorithm to search a specific record in a database containing student roll numbers.
Problem Solution:

Algorithm

1. **Input**:
   - An array arr[] containing predefined roll numbers.
   - The size of the array n.
   - The roll number to be searched (key) provided by the user.

2. **Linear Search**:
   - Traverse the array using a loop from index 0 to n-1.
   - Compare each element of the array with the key.
   - If a match is found, return the index of the matching element.
   - If the loop completes without finding the key, return -1.

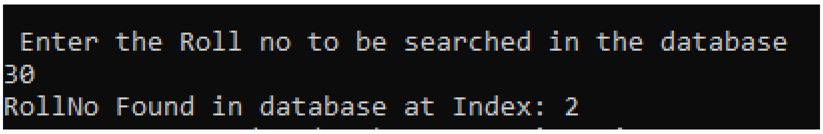
3. **Output**:
   - If the returned index is -1, print "RollNo Not Found."
   - Otherwise, print the index at which the roll number is found.

Key Points

1. **Linear Search**:
   - Simple and effective for small datasets.
   - Time complexity: O(n).

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

// C program to implement linear search using loop
int linearSearch(int* arr, int n, int key) {

   // Starting the loop and looking for the key in arr
   for (int i = 0; i < n; i++) {

      // If key is found, return key
      if (arr[i] == key) {
         return i;
      }
   }

   // If key is not found, return some value to indicate
   // end
   return -1;
}

int main() {
   int arr[] = { 10, 50, 30, 70, 80, 60, 20, 90, 40 };
   int n = sizeof(arr) / sizeof(arr[0]);
   int key = 30;
   printf("\n Enter the Roll no to be searched in the database\n");
   scanf("%d",&key);
   // Calling linearSearch() for arr with key
   int i = linearSearch(arr, n, key);

   // printing result based on value returned by
   // linearSearch()
   if (i == -1)
      printf("RollNo Not Found");
   else
      printf("RollNo Found in database at Index: %d", i);

   return 0;
}
```

OUTPUT:

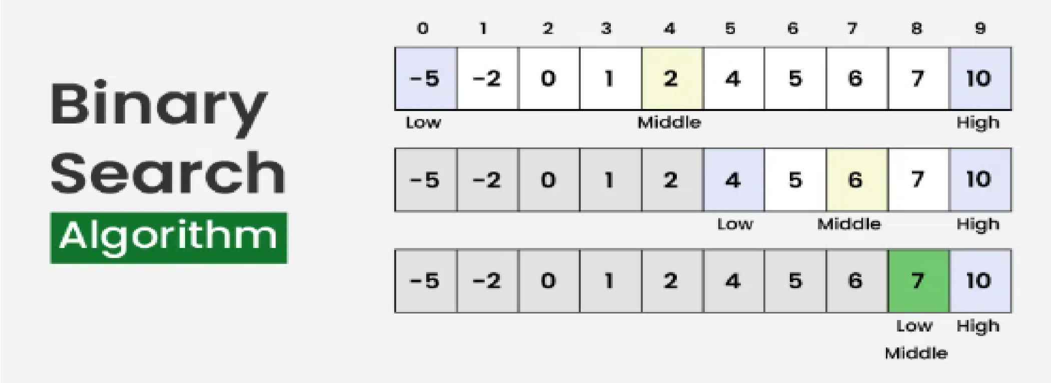


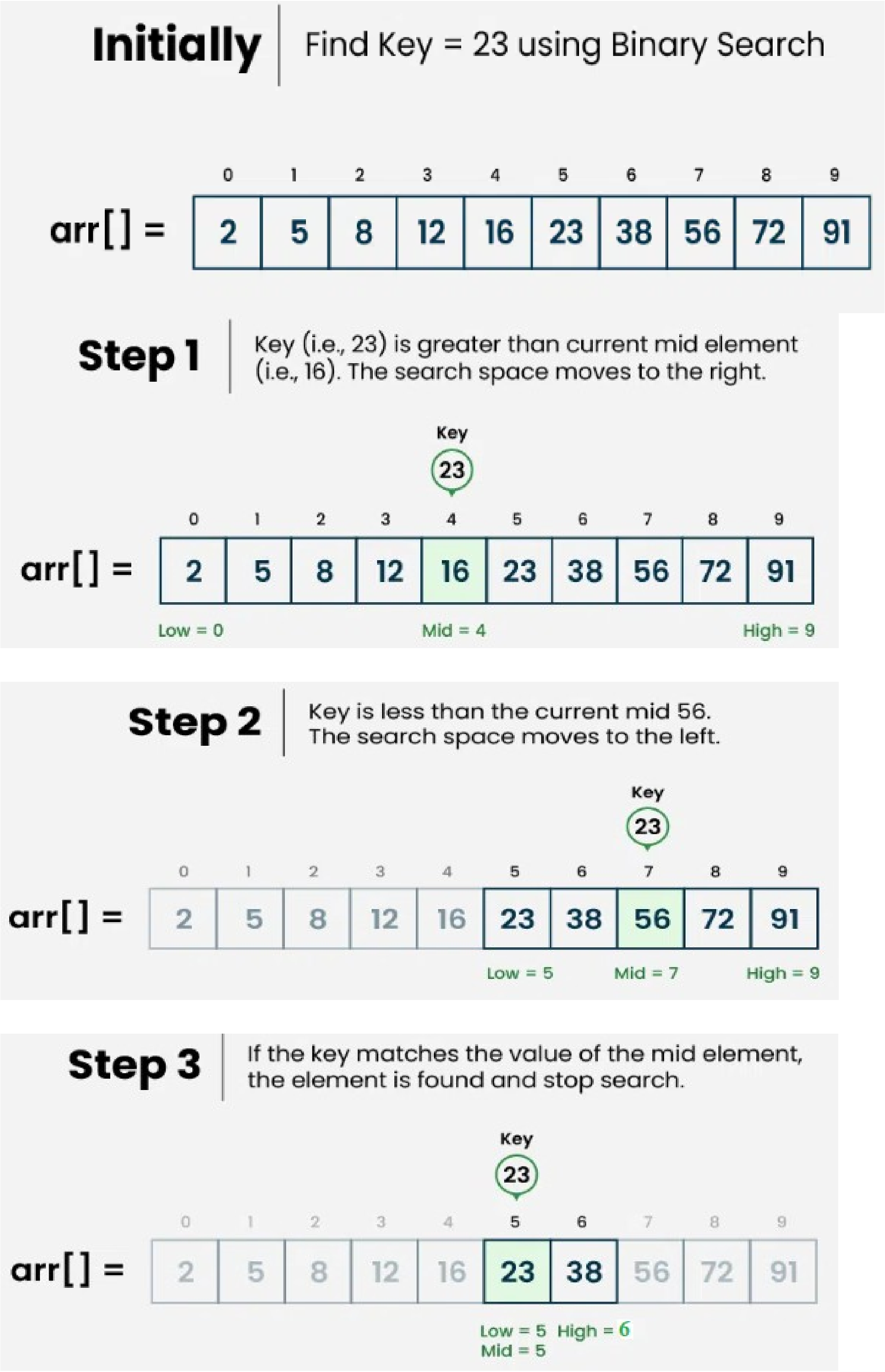
```
 Enter the Roll no to be searched in the database
30
RollNo Found in database at Index: 2
```

b. Write a program to implement the binary search algorithm to efficiently search a sorted database of product IDs in an inventory system.

Problem Solution:

Binary Search Algorithm is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).





Initially | Find Key = 23 using Binary Search

Step 1 | Key (i.e., 23) is greater than current mid element (i.e., 16). The search space moves to the right.

Step 2 | Key is less than the current mid 56. The search space moves to the left.

Step 3 | If the key matches the value of the mid element, the element is found and stop search.

Binary Search Algorithm (Iterative Implementation)

1. **Input**:
   - A sorted array arr[] of size n.

- The element x to search in the array.

2. **Initialization**:
    - Set low to 0 (starting index).
    - Set high to n-1 (last index).

3. **Iterative Search**:
    - Repeat until low is less than or equal to high:
        1. Compute mid as the middle index: mid=low+high−low2\text{mid} = \text{low} + \frac{\text{high} - \text{low}}{2}mid=low+2high−low
        2. Compare the middle element arr[mid] with x:
            - **If** arr[mid] == x:
                - Return mid (index of the found element).
            - **Else if** arr[mid] < x:
                - Update low to mid + 1 (ignore the left half).
            - **Else**:
                - Update high to mid - 1 (ignore the right half).

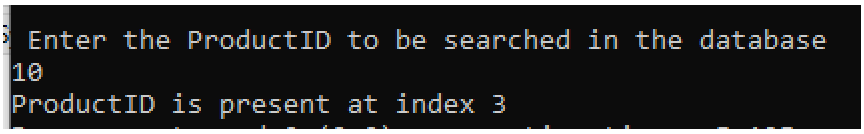
4. **Result**:
    - If the loop terminates without finding x, return -1 (element not present).

This algorithm efficiently searches sorted data using the divide-and-conquer method, minimizing the number of comparisons.

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

// C program to implement iterative Binary Search
// An iterative binary search function.
int binarySearch(int arr[], int low, int high, int x)
{
   while (low <= high) {
      int mid = low + (high - low) / 2;

      // Check if x is present at mid
      if (arr[mid] == x)
         return mid;

      // If x greater, ignore left half
      if (arr[mid] < x)
         low = mid + 1;

      // If x is smaller, ignore right half
      else
         high = mid - 1;
```

```
    }

    // If we reach here, then element was not present
    return -1;
}

// Driver code
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x;
    printf("\n Enter the ProductID to be searched in the database\n");
    scanf("%d",&x);
    int result = binarySearch(arr, 0, n - 1, x);
    if(result == -1)
        printf("ProductID is not present in database");
    else
        printf("ProductID is present at index %d",result);

}
```

OUTPUT:



```
 Enter the ProductID to be searched in the database
10
ProductID is present at index 3
```

Experiment 12:
a. Program to sort a data base using Bubble sort
Problem Solution:

Algorithm: Bubble Sort

1. **Input:**
   - An array arr of n elements.

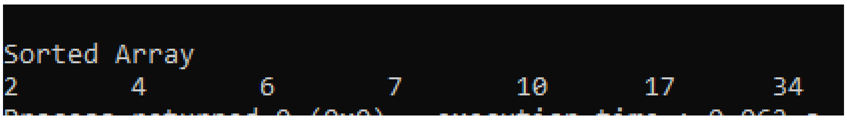
2. **Process:**
   - Start a loop with variable i from 0 to n-1.
     - This determines the number of passes (outer loop).
   - For each pass, start another loop with variable j from 0 to n-i-1.
     - This ensures comparisons are limited to unsorted parts of the array (inner loop).
   - In each iteration of the inner loop:
     - Compare adjacent elements arr[j] and arr[j+1].
     - If arr[j] > arr[j+1]:
       - Swap the two elements.
   - Repeat until no more swaps are required in a complete pass or until all passes are complete.

3. **Output:**
   - The input array is sorted in ascending order.

This implementation is a simple Bubble Sort, with a time complexity of O(n2) for the worst and average cases.

```
#include <stdio.h>
#include <stdlib.h>
void BubbleSort(int arr[],int n)
{
   for(int i=0;i<n-1;i++)
      for(int j=0;j<n-i-1;j++)
      {
         if(arr[j]>arr[j+1])
         {
            int temp = arr[j];
            arr[j]=arr[j+1];
            arr[j+1]=temp;
         }
      }
}
int main()
{
   int arr[7]={10,2,4,17,34,6,7};
   int n = sizeof(arr)/sizeof(arr[0]);
   BubbleSort(arr,n);
   printf("\nSorted Array\n");
   for(int i=0;i<n;i++)
   {
      printf("%d\t",arr[i]);
   }
   return 0;
}
```

OUPUT:



```

Sorted Array
2       4       6       7       10      17      34
```

b) Program to sort a data base using Quick sort

Problem Solution:

Algorithm

1. **Partition Function (`partition`):**
   - Select the first element (`arr[lb]`) as the pivot.
   - Initialize `start` to the lower bound (`lb`) and `end` to the upper bound (`ub`).
   - Move `start` forward until an element larger than the pivot is found.
   - Move `end` backward until an element smaller than or equal to the pivot is found.
   - Swap `arr[start]` and `arr[end]` if `start < end`.
   - After exiting the loop, swap the pivot with `arr[end]` to place the pivot in its correct position.
   - Return the pivot's index (`end`).

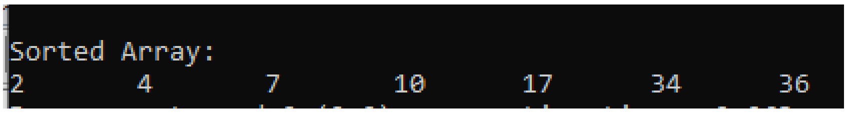
2. **QuickSort Function (`QuickSort`):**
   - If the subarray has more than one element (`lb < ub`), perform the following steps:
     - Partition the array and obtain the pivot index (`loc`).
     - Recursively sort the left subarray (`arr[lb]` to `arr[loc-1]`).
     - Recursively sort the right subarray (`arr[loc+1]` to `arr[ub]`).

3. **Main Function:**
   - Initialize the array and calculate its size.
   - Call `QuickSort` to sort the entire array.
   - Print the sorted array.

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

void swap(int *x, int *y)
{
   int temp = *x;
```

```
    *x = *y;
    *y = temp;
}

int partition(int arr[], int lb, int ub)
{
    int pivot = arr[lb];
    int start = lb;
    int end = ub;

    while (start < end)
    {
        while (arr[start] <= pivot && start <= ub - 1)
        {
            start++;
        }
        while (arr[end] > pivot && end >= lb + 1)
        {
            end--;
        }
        if (start < end)
            swap(&arr[start], &arr[end]);
    }
    swap(&arr[lb], &arr[end]);
    return end;
}

void QuickSort(int arr[], int lb, int ub)
{
    if (lb < ub)
    {
        int loc = partition(arr, lb, ub);
        QuickSort(arr, lb, loc - 1);
        QuickSort(arr, loc + 1, ub);
    }
}

int main()
{
    int arr[7] = {10, 2, 4, 17, 34, 36, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    QuickSort(arr, 0, n - 1);

    printf("\nSorted Array:\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d\t", arr[i]);
    }

    return 0;
}
```

OUTPUT:



```
Sorted Array:
2       4       7       10      17      34      36
```

Experiment 13:
a) Write a C program that implements Depth First Search (DFS) for an undirected graph using an adjacency matrix and a stack.
Problem Solution:

Algorithm for Depth First Search (DFS)

Input:

- A graph G=(V,E) represented as an adjacency matrix.
- A set of vertices, each labelled and unvisited initially.

Output:

- A sequence of vertices in DFS traversal order.

1. **Initialization:**

- Set all elements in the adjacency matrix adjMatrix to 0.
- Add vertices to the graph with a unique label and initialize each vertex as unvisited.
- Add edges between vertices by updating the adjacency matrix for undirected edges.

2. **DFS Procedure:**

- Start from the first vertex v0:
  - Mark v0 as visited.
  - Display the label of v0.
  - Push v0's index onto the stack.

3. **Traverse the Graph:**

- While the stack is not empty:
  - Peek the top vertex of the stack.
  - Find an unvisited adjacent vertex v_{adj} to the top vertex:
    - If v_{adj} exists:
      - Mark v_{adj} as visited.
      - Display the label of v_{adj} .
      - Push v_{adj} 's index onto the stack.
    - If no unvisited adjacent vertex exists:
      - Pop the vertex from the stack.

4. **Completion:**

- When the stack becomes empty, the DFS traversal is complete.
- Reset the visited status of all vertices to false for potential future searches.

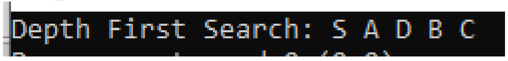
5. **Output:**

- Print the DFS traversal sequence as the search progresses.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 5
struct Vertex {
  char label;
  bool visited;
};
//stack variables
int stack[MAX];
int top = -1;
//graph variables
//array of vertices
struct Vertex* lstVertices[MAX];
//adjacency matrix
int adjMatrix[MAX][MAX];
//vertex count
int vertexCount = 0;
//stack functions
void push(int item) {
  stack[++top] = item;
}
int pop() {
  return stack[top--];
}
int peek() {
  return stack[top];
}
bool isStackEmpty() {
  return top == -1;
}
//graph functions

//add vertex to the vertex list
void addVertex(char label) {
  struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
  vertex->label = label;
  vertex->visited = false;
  lstVertices[vertexCount++] = vertex;
}
//add edge to edge array
```

```
void addEdge(int start,int end) {
  adjMatrix[start][end] = 1;
  adjMatrix[end][start] = 1;
}
//display the vertex
void displayVertex(int vertexIndex) {
  printf("%c ",lstVertices[vertexIndex]->label);
}
//get the adjacent unvisited vertex
int getAdjUnvisitedVertex(int vertexIndex) {
  int i;
  for(i = 0; i <vertexCount; i++) {
    if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false) {
      return i;
    }
  }
  return -1;
}
void depthFirstSearch() {
  int i;
  //mark first node as visited
  lstVertices[0]->visited = true;
  //display the vertex
  displayVertex(0);
  //push vertex index in stack
  push(0);
  while(!isStackEmpty()) {
    //get the unvisited vertex of vertex which is at top of the stack
    int unvisitedVertex = getAdjUnvisitedVertex(peek());
    //no adjacent vertex found
    if(unvisitedVertex == -1) {
      pop();
    } else {
      lstVertices[unvisitedVertex]->visited = true;
      displayVertex(unvisitedVertex);
      push(unvisitedVertex);
    }
  }
  //stack is empty, search is complete, reset the visited flag
  for(i = 0;i < vertexCount;i++) {
    lstVertices[i]->visited = false;
  }
}
int main() {
  int i, j;
  for(i = 0; i < MAX; i++) {   // set adjacency
    for(j = 0; j < MAX; j++) // matrix to 0
      adjMatrix[i][j] = 0;
  }
  addVertex('S');   // 0
  addVertex('A');   // 1
  addVertex('B');   // 2
  addVertex('C');   // 3
  addVertex('D');   // 4
  addEdge(0, 1);    // S - A
  addEdge(0, 2);    // S - B
  addEdge(0, 3);    // S - C
  addEdge(1, 4);    // A - D
  addEdge(2, 4);    // B - D
  addEdge(3, 4);    // C - D
  printf("Depth First Search: ");
  depthFirstSearch();
  return 0;
}
```

Output:



```
Depth First Search: S A D B C
```

Experiment 14:

a) Write a C program that implements Breadth First Search (BFS) for an undirected graph using an adjacency matrix and a stack.

Problem Solution:

Algorithm for Breadth-First Search (BFS)

Input:

- A graph G=(V,E), represented as an adjacency matrix.
- A set of vertices, each labeled and unvisited initially.

Output:

- A sequence of vertices in BFS traversal order.

1. **Initialization:**

   - Create an adjacency matrix adjMatrix and set all elements to 0.
   - Create an array lstVertices to store graph vertices.
   - Initialize a queue to manage the BFS traversal.

2. **Add Vertices:**

   - For each vertex v, add it to lstVertices and set its visited status to false.

3. **Add Edges:**

   - For each edge between vertices u and v, update adjMatrix[u][v] and adjMatrix[v][u] to 1.

4. **Start BFS Traversal:**

   - Mark the starting vertex v0 as visited: lstVertices[0].visited=true.
   - Display the label of v0.
   - Enqueue v0's index: insert(0).

5. **Traverse the Graph:**

   - While the queue is not empty:
     - Dequeue a vertex index tempVertex: tempVertex = removeData().
     - Find all unvisited adjacent vertices of tempVertex:
       - For each adjacent vertex :
         - If v_{adj} is unvisited:
           - Mark v_{adj} as visited.
           - Display the label of v_{adj} .
           - Enqueue v_{adj} 's index: insert(v_{adj}).

6. **Completion:**

- When the queue becomes empty, the BFS traversal is complete.
- Reset the visited status of all vertices to false for potential future searches.

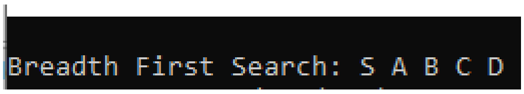
7. **Output:**

- Print the BFS traversal sequence as the search progresses.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 5
struct Vertex {
  char label;
  bool visited;
};
//queue variables
int queue[MAX];
int rear = -1;
int front = 0;
int queueItemCount = 0;
//graph variables
//array of vertices
struct Vertex* lstVertices[MAX];
//adjacency matrix
int adjMatrix[MAX][MAX];
//vertex count
int vertexCount = 0;
//queue functions
void insert(int data) {
  queue[++rear] = data;
  queueItemCount++;
}
int removeData() {
  queueItemCount--;
  return queue[front++];
}
bool isQueueEmpty() {
  return queueItemCount == 0;
}
//graph functions
//add vertex to the vertex list
void addVertex(char label) {
  struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
  vertex->label = label;
  vertex->visited = false;
  lstVertices[vertexCount++] = vertex;
}
//add edge to edge array
void addEdge(int start,int end) {
  adjMatrix[start][end] = 1;
  adjMatrix[end][start] = 1;
}
//display the vertex
void displayVertex(int vertexIndex) {
  printf("%c ",lstVertices[vertexIndex]->label);
}
//get the adjacent unvisited vertex
int getAdjUnvisitedVertex(int vertexIndex) {
  int i;

  for(i = 0; i<vertexCount; i++) {
    if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false)
      return i;
  }
  return -1;
}
void breadthFirstSearch() {
  int i;
  //mark first node as visited
  lstVertices[0]->visited = true;
  //display the vertex
  displayVertex(0);
  //insert vertex index in queue
  insert(0);
  int unvisitedVertex;
  while(!isQueueEmpty()) {
    //get the unvisited vertex of vertex which is at front of the queue
    int tempVertex = removeData();
    //no adjacent vertex found
    while((unvisitedVertex = getAdjUnvisitedVertex(tempVertex)) != -1) {
      lstVertices[unvisitedVertex]->visited = true;
      displayVertex(unvisitedVertex);
      insert(unvisitedVertex);
    }
  }
  //queue is empty, search is complete, reset the visited flag
  for(i = 0;i<vertexCount;i++) {
    lstVertices[i]->visited = false;
  }
```

```
}
int main() {
  int i, j;

  for(i = 0; i<MAX; i++) { // set adjacency
    for(j = 0; j<MAX; j++) // matrix to 0
      adjMatrix[i][j] = 0;
  }
  addVertex('S');   // 0
  addVertex('A');   // 1
  addVertex('B');   // 2
  addVertex('C');   // 3
  addVertex('D');   // 4
  addEdge(0, 1);    // S - A
  addEdge(0, 2);    // S - B
  addEdge(0, 3);    // S - C
  addEdge(1, 4);    // A - D
  addEdge(2, 4);    // B - D
  addEdge(3, 4);    // C - D
  printf("\nBreadth First Search: ");
  breadthFirstSearch();
  return 0;
}
```

OUTPUT:



```
Breadth First Search: S A B C D
```

References

[1] Data Structure using C, Aaron M. Tanenbaum, YedidyahLangsam& Moshe J. Augenstein, Pearson Education/PHI, 2009

[2] The C Programming Language, Brian W Kernighan and Dennis M Ritchie, Prentice Hall Software Series, 2nd Edition

[1] https://www.geeksforgeeks.org

[2] https://www.greatlearning.in

[3] https://www.edureka.co

[4] https://www.programiz.com

[5] https://www.tutorialspoint.com