EXPERIMENT N0:2

AIM:

To implement different types of searching techniques

1.Linear search (or) sequential search

2.Binary search

DESCRIPTION:

Searching is the process of finding specific element or value or information from the collection of data.

searching techniques significantly reduce the time and resources needed to find desired information.

Each searching technique has its strengths and weaknesses, making it crucial to select the appropriate one based on factors such as data size, structure, and desired performance.

LINEAR SEARCH:

Linear search, also known as sequential search.It is a basic searching algorithm used to find a specific element within a collection of data.

In a linear search, each element is checked one after the other in order until the desired element is found or until all elements have been checked.

Linear search is used where the data is small(n<=16) or not sorted (where n is number of elements in an array).

Linear search begins at the starting of data and compares the search element with all elements in order.

If the search element is found the index where element is found is returned and search is terminated and if element is not found

It returns that element is not found.

ADVANTAGES OF LINEAR SEARCH:

Linear search is easy to understand and implement.

Linear search works efficiently on unsorted data. It doesn't require the data to be sorted or structured in a specific way.

Linear search performs searching operation for small data.

Linear search can be applied to various types of data structures, including arrays, linked lists etc.

Linear search does not require extra memory.

DISADVANTAGES OF LINEAR SEARCH:

Linear search does work efficiently if the size of the data is large.

As the number of elements in an array increases the time taken to search the element also increases.

APPLICATIONS OF LINEAR SEARCH:

Library: Library is the best example for linear search.inorder to find the book in a library we need to verify the title of each and every book.

Phonebook Search: Linear search can be used to searches a phonebook to find a person's name as given according to their phone number.

EXAMPLE FOR LINEAR SEARCH:

Consider an array

0	1	2	3	4	5
30	46	25	12	78	22

Let the search element be 12

Search=12

Step 1:

The element 12 is compared with the element in the first index(30)

0	1	2	3	4	5
30	46	25	12	78	22

12

i=0 ,i<n, true

a[i]==x (30==12) false

The element is not found move to next element

i=i+1=0+1=1

Step 2:

The element 12 is compared with the next element (46)

0	1	2	3	4	5
30	46	25	12	78	22

12

i=1 ,i<n (True)

a[i]==search (46==12) (False)

the element is not matched move to the next element

i=i+1=1+1=2

Step 3:

The element 12 is compared with next element (35)

0	1	2	3	4	5
30	46	35	12	78	22

12

i=2 ,i<n(true)

a[i]==search (35==12) (false)

the element is not matched move to the next element

i=i+1=2+1=3

Step 4:

The element 12 is compared with next element (12)

0	1	2	3	4	5
30	46	35	12	78	22

12

i=3 , i<n(true)

a[i]==search (12==12) (true)

the element is matched then return the index and terminate the loop.

ALGORITHM:

1.Read the length of the array n.

2.create an array and store values in it.

3.Read the element to be searched.

4.initialize the I value to 0.

5.transverse the array from index 0 to n.

6.if element is found increment count from 0 to 1 and break the loop.

7.increment I value for every iteration.

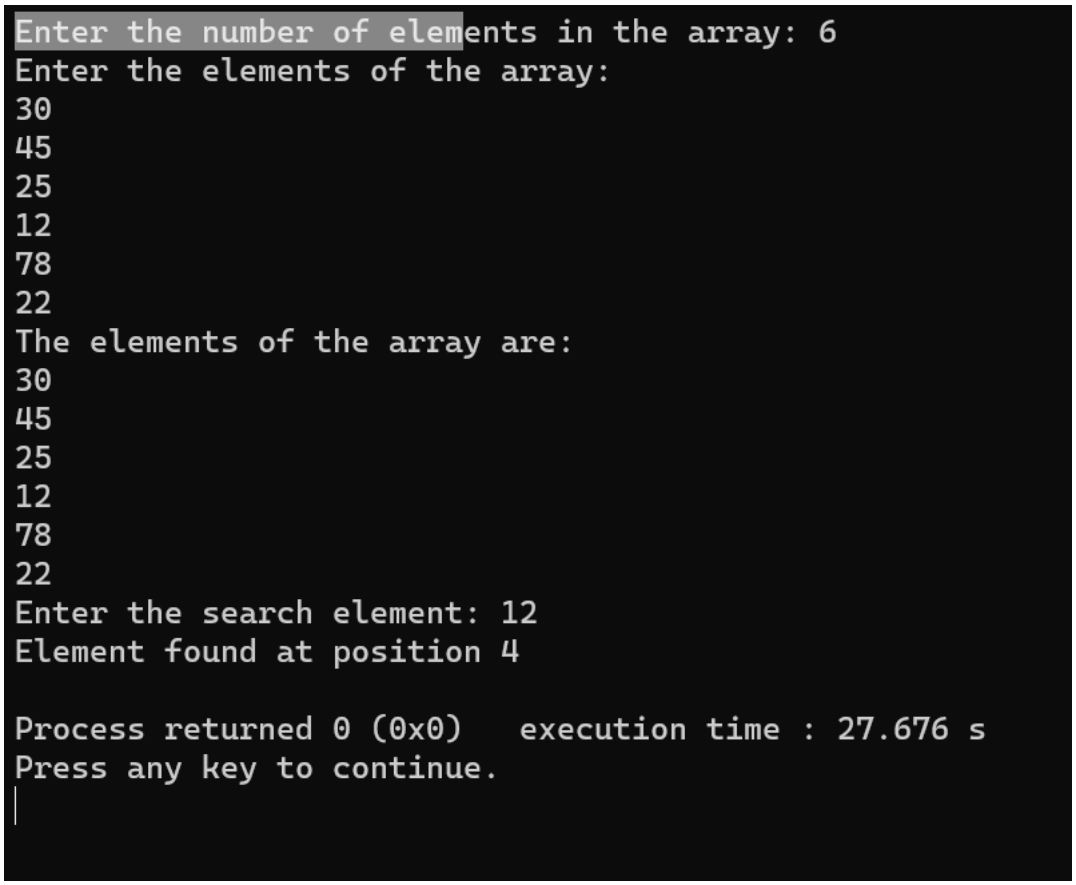
8.if count=0, print the element is not found.

LINEAR SEARCH ON INTEGERS:

```
/*the following program executes linear search*/
#include <stdio.h>
#define MAX 100

int main () {
   int a[MAX]; // Array declaration to hold elements
   int i, n, flag = 0; // Variables for loop control and search flag
   int key; // Variable to store the search element

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

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

```
    }

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

    printf("Enter the search element: ");
    scanf("%d", &key);

    // Searching process begins here
    for (i = 0; i < n; i++) {
        if (a[i] == key) {
            printf("Element found at position %d\n", i + 1);
            flag = 1; // Indicate that the element has been found
            break; // Exit the loop since the element has been found
        }
    }

    if (flag == 0) {
        printf("Element is not found in the array\n");
    }

    return 0;
}
```

OUTPUT:



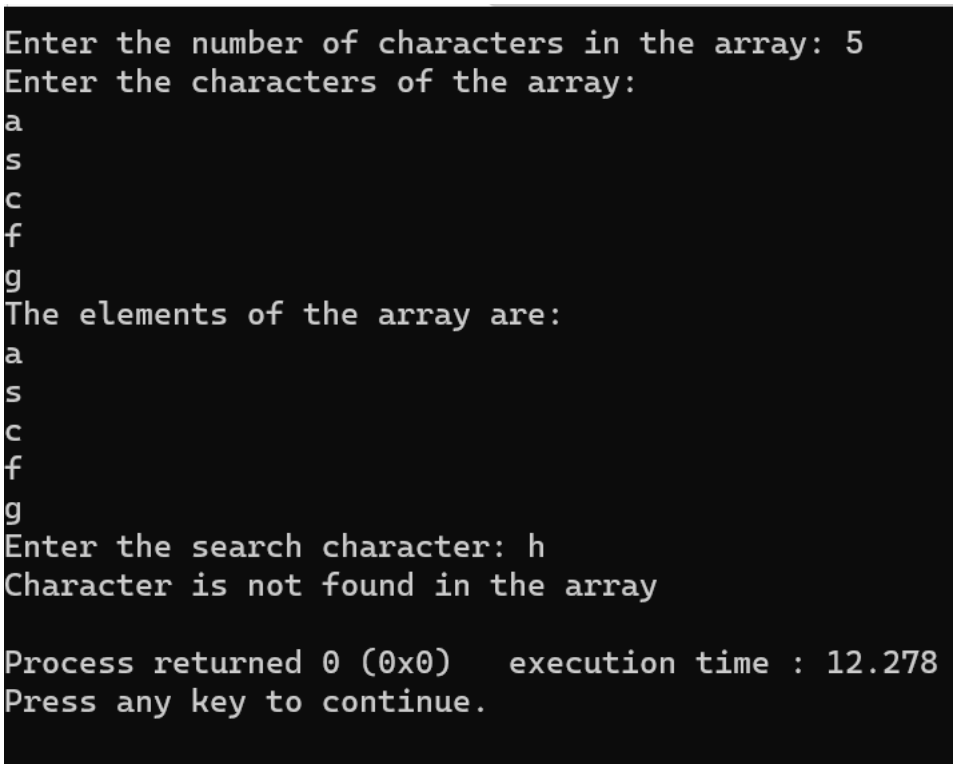
LINEAR SEARCH ON CHARACTERS:

```
/*the following program executes the linear search of characters*/
#include<stdio.h>
#define MAX 100

int main() {
    char arr[MAX]; // Array declaration to hold characters
    int i, n, flag = 0;
    char key; // Variable to store the search character

    // Prompt the user to enter the number of characters in the array
    printf("Enter the number of characters in the array: ");
```

```
    scanf("%d", &n);

    // Prompt the user to enter the characters for the array
    printf("Enter the characters of the array:\n");
    // Scan characters, consuming leading whitespace with a
space before %c
    for (i = 0; i < n; i++) {
        scanf(" %c", &arr[i]);
    }

    // Display the entered array elements
    printf("The elements of the array are:\n");
    for (i = 0; i < n; i++) {
        printf("%c\n", arr[i]);
    }

    // Prompt the user to enter the search character
    printf("Enter the search character: ");
    // Consume leading whitespace before reading the character
    scanf(" %c", &key);

    // Searching process begins here
    for (i = 0; i < n; i++) {
        if (arr[i] == key) {
            printf("Character '%c' found at position %d\n", key, i +
1);
            flag = 1; // Indicate that the character has been found
            break; // Exit the loop since the character has been
found
        }
    }
```

```
    // Display the result based on whether the character was
found or not
    if (flag == 0) {
        printf("Character is not found in the array\n");
    }

    return 0; // Return 0 to indicate successful completion
}
```

Output:



BINARY SEARCH:

Binary search is used for finding the selected element in the sorted collection of data.

Binary search algorithm works on the principle of divide and conquer and works on the large sized data.

Binary search performs by continuously dividing the search by half by decreasing the possibilities until the selected element is found .

This search performs by comparing the middle most element with the selected element if both are matched the index is returned.

If middle term is greater than the selected element then the element is searched in the subarray left of the middle element.

If middle element is smaller than the selected element then the element is searched in the subarray right to the middle element.

This process continuous until the subarray size reduce to zero.

ADVANTAGES OF BINARY SEARCH:

It works efficiently on large sized data as number of elements increases it works fast as it is suitable for it.

Binary search requires less number of comparisions compared to that of linear search.

Binary search has less compilation time so there will be better time complexity.

DISADVANTAGES OF BINARY SEARCH:

Binary search needs data to be sorted in ascending or descending order.

If the data is not sorted sorting process need to be executed which increases the time complexity.

Binary search is most effective when searching for a single specific element. It's not suitable for situations where multiple occurrences of the same element need to be located

Maintaining the sorted order when inserting or deleting elements can be complex and time-consuming.

APPLICATIONS OF BINARY SEARCH:

Dictionary: Dictionary is the best application of binary search as we check the word alphabetical wise first we check middle pages and check where the word will be present and leave the rest of the pages.

Ecommerce :Binary search is employed in e-commerce platforms for rapid product searches within large inventories, enhancing user experience

Text editors: In text editors and word processors, binary search can be used to locate specific points within documents, like chapters or sections

Geographic information: Geographic information systems (GIS) use binary search to find locations or landmarks based on coordinates or identifiers.

EXAMPLE OF BINARY SEARCH:

Consider an array

0	1	2	3	4	5	6
10	20	30	40	50	60	70

.

Let the search element be 20

Search=20

Low=0 and high=n-1=7-1=6

Step 1:

The search element (20) is compared with the middle element(40).

0	1	2	3	4	5	6
10	20	30	40	50	60	70

20

If(low<=high) (0<6) (true)

Mid=(low+high)/2=3

a[Mid]==search (40==20) (false)

Both the elements are not matching and the element 20 is smaller than the middle element so the left subarray is searched.

High=mid-1=3-1=2

Step 2:

The search element (20) is compared is the middle element

0	1	2	3	4	5	6
10	20	30	40	50	60	70

20

If(Low<high) (0<2) (true)

Mid=(low+high)/2=1

a[Mid]==search (20==20)

both the elements are matching so return index 1

ALGORITHM FOR BINARY SEARCH:

Step 1: Read the size of an array (n)

Step 2: Read the elements in an array and store it in the array (a[n]).

Step 3: Read the search element

Step 4: Find the middle element in the sorted array.

Step 5: Compare the search element with the middle element in the sorted array.

Step 6: if both the middle element and search element are matched then print that element is found in index i and terminate the loop

Step 7: If both are not matched, then check whether the search element is smaller or larger than the middle element.

Step 8: If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left subarray of the middle element.

Step 9: If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.

Step 10: Repeat the same process until we find the search element in the array or until subarray contains only one element.

Step 11: if the search element does not match with any element then print element not present.

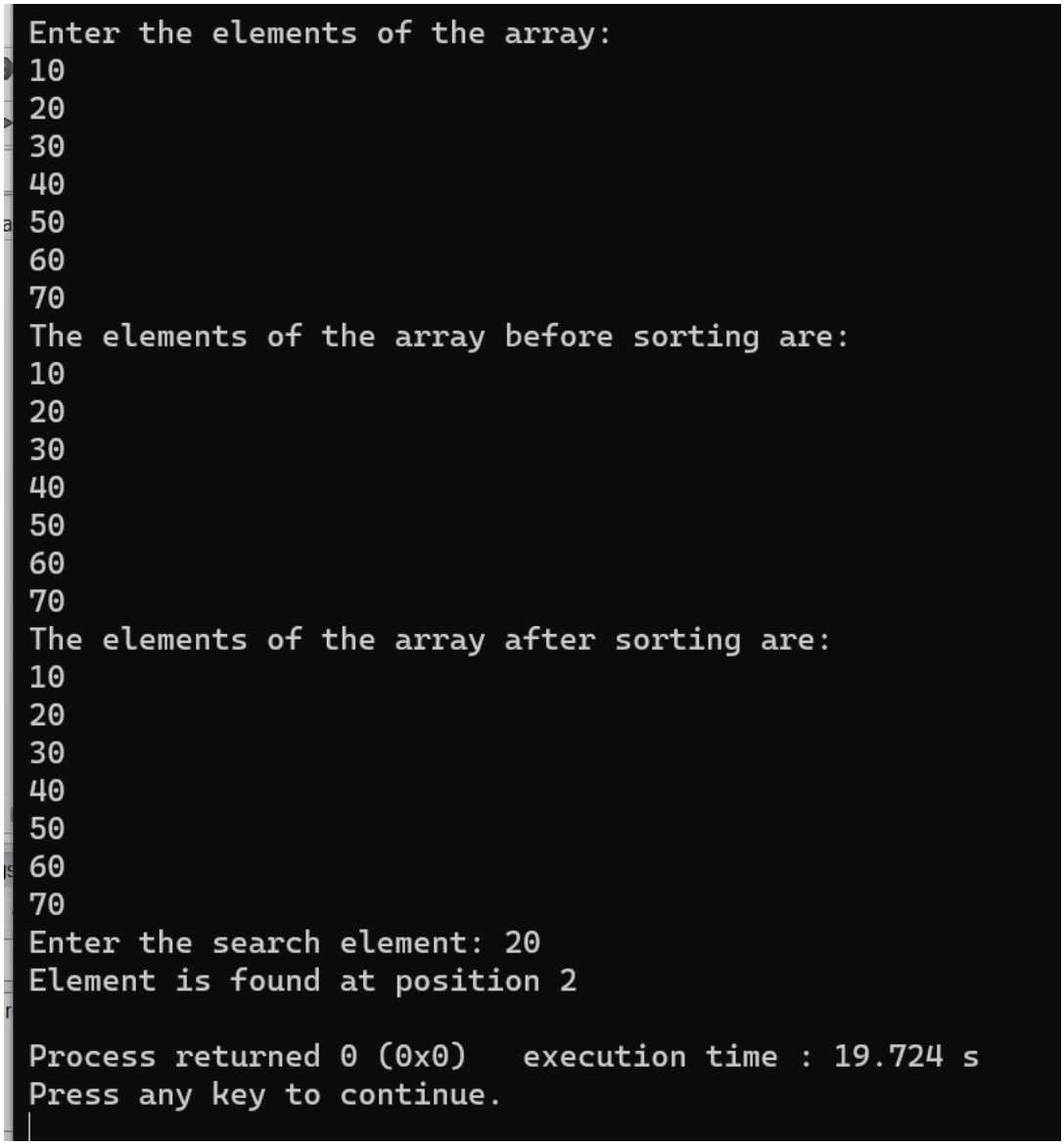
BINARY SEARCH FOR NUMERICALS:

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

int main() {
    int a[MAX]; // Array declaration to hold elements
    int i, j, n, flag = 0; // Variables for loop control and search flag
    int key, mid; // Variable to store the search element
    int low, high;

    // Prompt the user to enter the number of elements in the
array
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);

    // Prompt the user to enter the elements for the array
    printf("Enter the elements of the array:\n");
    // Scanning the array elements
    for (i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }
```

```
// Display the entered array elements before sorting
printf("The elements of the array before sorting are:\n");
for (i = 0; i < n; i++) {
    printf("%d\n", a[i]);
}

// Bubble sort the array
for (i = 0; i < n - 1; i++) {
    for (j = i + 1; j < n; j++) {
        if (a[i] > a[j]) {
            // Swap elements to sort the array in ascending order
            int temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }
}

// Display the sorted array elements
printf("The elements of the array after sorting are:\n");
for (i = 0; i < n; i++) {
    printf("%d\n", a[i]);
}

// Prompt the user to enter the search element
printf("Enter the search element: ");
scanf("%d", &search);

// Binary search process begins here
low = 0;
high = n - 1;
while (low <= high) {
```

```
        mid = (low + high) / 2;
        if (a[mid] == search) {
            printf("Element is found at position %d\n", mid + 1);
            flag = 1;
            break;
        } else if (search > a[mid]) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    // Display the result based on whether the element was
found or not
    if (flag == 0) {
        printf("Element is not found in the array\n");
    }

    return 0; // Return 0 to indicate successful completion
}
```

Output:


Enter the elements of the array:
10
20
30
40
50
60
70
The elements of the array before sorting are:
10
20
30
40
50
60
70
The elements of the array after sorting are:
10
20
30
40
50
60
70
Enter the search element: 20
Element is found at position 2
Process returned 0 (0x0) execution time : 19.724 s
Press any key to continue.

BINARY SEARCH FOR STRINGS:

```
#include<stdio.h>
#include<string.h>
int main(){
    char *a[50];
    int k;
    printf("no.of strings:\n");
scanf("%d",&k);
    printf("entering the strings:");
    for(int i=0;i<k;i++){
        scanf("%s",&a[i]);
    }
    char search[50];
    printf("enter the string to be search:");
    scanf("%s",&search);
    int low=0,high=k-1,mid;
    while(low<=high){
        mid=(low+high)/2;

        if (strcmp(search,a[mid])==0){
            printf("\nstring found at %d position",mid);
            return 0;
```

```
        }
        else if(strcmp(search,a[mid])>0)
        low=mid+1;
        else
        high=mid-1;
    }
    printf("\nelement not found");
return 0;
}
```

Output:

```
no.of strings:
3
entering the strings:adf
bhj
kli
enter the string to be search:kli

element not found
Process returned 0 (0x0)   execution time : 17.869 s
Press any key to continue.
```

DIFFERENCES BETWEEN LINEAR AND BINARY SEARCH:

LINEAR SEARCH	BINARY SEARCH
1.Linear search is simple to implement.	1.binary search is complex compared to that of linear search.
2.in linear search more number of comparisions are required	2.In binary search less number of comparisions are required.
3.It is not required to sort the Array before searching the element in an array	3.it is required to sort the array before searching the element in an array.
4.it is also known as sequential search.	4.it is also known as logarithimic search and half interval search.
5.best case is to find the element in the first position.	5.best case is to find the element in the middle position.
6.time complexity is O(n)	6.time complexity is O(logN).
7.Access the data sequentially.	7.Access the data randomly.

EXPERIMENT NO:3

AIM:

To implement different types of sorting techniques on a given list

1.Bubble sort

2.Selection sort

3.Insertion sort

4.Merge sort

5.Quick sort

DESCRIPTION:

Sorting is the process of arranging a collection of items or data elements in a specific order.

Sorting algorithms are the set of instructions that takes an array or list as input and arrange elements in a particular order.

Sorting is performed to make the the data more organized and easier to search (or) analyze.

There are various sorting algorithms which have their own advantages and disadvantages based on time and space complexity.

Sorting is usually performed on numericals and characters by placing them in numerical order or order of letters and words(Dictionary order (or) lexicographical order).

Sorting is usually done in two ways ,they are:

1.Internal sorting.

2.External sorting.

INTERNAL SORTING: Data to be sorted (or)number of objects is small enough that fits entirely within the computer's main memory (RAM) then sorting performed is called as internal sorting.

EXTERNAL SORTING: Data to be sorted (or) number of objects is too large to fit into the main memory.

It is slower than the internal sorting.

BUBBLE SORT:

Bubble sort algorithm is a comparision based algorithm and it uses the concept of sorting by exchange.

Bubble sort algortithm is a simple algorithm that steps through the elements to be sorted and compares the adjacent elements and the elements are swapped if they are in the wrong order.

In every pass of the bubble sort algorithm the largest element is bubbled and placed in the last position of the array.

The algorithm repeats this process until no more swaps are needed (the elements are sorted).

EXAMPLE FOR BUBBLE SORT:

Consider an array of integers which is unsorted

0	1	2	3	4	5
37	25	4	29	30	55

Pass 1:

Bubble sort starts from the starting index of the array and it compares the first and second element.

If the first element is greater than the second element then they are swapped

37	25	4	29	30	55

Here 37 is greater than 25(37>25),so swap the elements.

0	1	2	3	4	5
25	37	4	29	30	55

Now 37 and 4 are compared and 37 is greater than 4(37>4).swap the elements

0	1	2	3	4	5
25	4	37	29	30	55

Now compare 37 and 29 and 37 is greater than 29 (37>29).swap the elements.

0	1	2	3	4	5
25	4	29	37	30	55

Now compare 37 and 30 and 37 is greater than 30 (37>30).swap the elements.

0	1	2	3	4	5
25	4	29	30	37	55

Now compare 37 and 55 and 37 is smaller than 55(37<55) do not swap

Bubble the largest element placed in last position.

0	1	2	3	4	5
25	4	29	30	37	55

PASS 2:

Here last element is bubbled now repeat the steps of pass 1.

First compare 25 and 4 ,as 25 is greater than 4(25>4).swap the elements.

25	4	29	30	37	55

4	25	29	30	37	55

Now compare 25 and 29 ,as 25 is smaller than 29 (25<29) do not swap and move to next elements.

0	1	2	3	4	5
4	25	29	30	37	55

Now compare 29 and 30 ,as 29 is not greater than 30(29<30) do not swap and move to the next elements.

0	1	2	3	4	5
4	25	29	30	37	55

Now compare 30 and 37,as 30 is not greater than 37 (30<37) do not swap and bubble the last element(37).

0	1	2	3	4	5
4	25	29	30	37	55

PASS 3:

Here last two elements are bubbled.

0	1	2	3	4	5
4	25	29	30	37	55

Compare 4 and 25 ,as 4 is not greater than 25(4<25) do not swap and move to next elements.

0	1	2	3	4	5
4	25	29	30	37	55

Now compare 25 and 29,as 25 is not greater than 29(25<29) do not swap and move to next elements.

0	1	2	3	4	5
4	25	29	30	37	55

Now compare 29 and 30 ,as 29 is not greater than 30 (29<30).do not swap and bubble the last element.

0	1	2	3	4	5
4	25	29	30	37	55

PASS 4:

Here last three elements are bubbled.

0	1	2	3	4	5
4	25	29	30	37	55

Now compare 4 and 25 ,as 4 is not greater than 25 (4<25) do not swap and move to next elements.

0	1	2	3	4	5
4	25	29	30	37	55

Now compare 25 and 29 ,as 25 is not greater than 29 (25<29) do not swap and bubble the last element(29).

0	1	2	3	4	5
4	25	29	30	37	55

PASS 5:

Here last 4 elements are bubbled.

0	1	2	3	4	5
4	25	29	30	37	55

Now compare 4 and 25 ,as 4 s not greater than 25 (4<25) do not swap and bubble the last element(25).

Here all the elemets are sorted.

0	1	2	3	4	5
4	25	29	30	37	55

ALGORITHM:

1. Read the length of the array(n).
2. Create an array and enter the elements into it.
3. Compare the first two elements of the array.
4. If the first element is greater than the second, swap them.
5. Move to the next pair of elements and repeat step 2.
6. Continue this process for all elements in the list, repeatedly comparing and swapping adjacent elements until the largest element "bubbles" to the end of the list.
7. Repeat the above steps for the remaining elements, excluding the last one that has already been sorted.

ADVANTAGES OF BUBBLE SORT:

1. Easy to understand and implement.
2. Bubble sort does not require extra space (or) memory.
3. Bubble sort is a stable sort.

DISADVANTAGES OF BUBBLE SORT:

1. Bubble sort has worst and average case time complexities of O(n^2),so it is not suitable for large datasets.
2. It performs same no:of comparisions and swaps even array is partially or completely unsorted.(unnecessary comparisions and swaps).
3. Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

APPLICATIONS OF BUBBLE SORT:

1. Bubble sort is often used as an introductory example when teaching sorting algorithms and programming concepts.
2. Bubble sort is also used to demonstrate the inefficiency of certain sorting algorithms compared to more optimized ones like merge sort or quick sort.
3. Bubble sort is a sorting algorithm that is used to sort the elements in an ascending order.

PERFORMANCE:

TIME COMPLEXITY:O(n^2)

SPACE COMPLEXITY:O(1)

PROGRAM OF BUBBLE SORT:

```
#include<stdio.h>
#define MAX 20
Int main(){
    Int a[MAX],n;
    Printf(“enter the n value:”);
    Scanf(“%d”,&n);
    Printf(“Enter the values of an array”);
    for(int i=0;i<n;i++){
    scanf(“%d”,&a[i]);
     }
    for(int i=0 ; i<n-1 ; i++){
       for(int j=0 ;  j<n-i-1 ; j++){
              if(a[j]>a[j+1]){
                  int temp=a[j];
                       a[j]=a[j+1];
                        a[j+1]=temp;
                  }
               }
            }
       Printf(“the sorted array is :”);
       for(int i=0 ; i<n ; i++){
           printf(“%d”,a[i]);
}
```

```
return 0;
}
```

OUTPUT:

```
enter the number of elements in an array:6
Enter number of array elements: 25
4
37
29
55
30
Array elements after Sorting is:
4 25 29 30 37 55
Process returned 0 (0x0)   execution time : 18.250 s
Press any key to continue.
```

SELECTION SORT:

Selection sort is a simple sorting algorithm which is a inplace comparision based soerting algorithm.

Selection sort follows the concept of sorting by selection.

It works by selecting the minimum element of the given unsorted array and it is swapped with the leftmost most element of unsorted array now the element becomes part of sorted array.

This process is repeated until the array is sorted.

EXAMPLE FOR SELECTION SORT :

Consider an unsorted array

0	1	2	3	4	5
37	25	4	29	55	30

Now consider MinIndex=0 and MinValue=37

PASS 1:

0	1	2	3	4	5
37	25	4	29	55	30

MinIndex=0;

MinValue=a[MinIndex]=a[0]=37

37<25(False)=>update the MinIndex=0+1=1

MinValue=a[MinIndex]=a[1]=25

37	25	4	29	55	30

Now Minvalue=25,MinIndex=1;

25<4(False)=>update the MinIndex=1+1=2;

MinValue=a[2]=4;

0	1	2	3	4	5
37	25	4	29	55	30

Now MinIndex=2,MinValue=a[2]=4;

4<29(True)->go for next element

4<55(True)->go for next element

4<30(True)

Swap (a[0],a[MinIndex]);=>(swap(37,4))

PASS 2:

0	1	2	3	4	5
4	25	37	29	55	30

Here 4 is sorted,now MinIndex=1,MinValue=a[MinIndex]=a[1]=25

0	1	2	3	4	5
4	25	37	29	55	30

25<37(True)->go for next element

25<29(True)->go for next element

25<55(True)->go for next element

25<30(True)

PASS 3:

Here 4 and 25 are sorted,MinIndex=1+1=2,MinValue=a[MinIndex]=a[2]=37

0	1	2	3	4	5
4	25	37	29	55	30

37<29(False)->update MinIndex=2+1=3

MinValue=a[MinIndex]=a[3]=29

29<55(True)->go for next element

29<30(True)->go for next element

Swap(a[2],a[MinIndex])->swap(37,29);

0	1	2	3	4	5
4	25	29	37	55	30

PASS 4:

Here 4,25,29 are sorted,MinIndex=3,MinValue=a[3]=37;

0	1	2	3	4	5
4	25	29	37	55	30

37<55(True)->go for next element

37<30(False)->update MinIndex=5

MinValue=a[MinIndex]=a[5]=30

Swap(a[3],a[MinIndex])->swap(37,30)

0	1	2	3	4	5
4	25	29	30	55	37

PASS 5:

Here 4,25,29,30 are sorted,now MinIndex=4,

MinValue=a[MinIndex]=55

55<37(False)->update MinIndex=5

MinValue=a[MinIndex]=37;

Swap(a[4],a[MinIndex])->swap(37,55)

0	1	2	3	4	5
4	25	29	30	37	55

All the elements are sorted.

ALGORITHM OF SELECTION SORT:

1. Read the length of the given array(n).
2. Create an array and enter the elements into it.
3. Consider the first element of unsorted array as the minimum element of the array.
4. Search the minimum element and update the MinIndex
5. Swap the element in the MinIndex with first index element of unsorted array.
6. Again set the minindex value to first index of the unsorted array and repeat the steps.
7. Repeat the steps until all elements are sorted.

ADVANTAGES OF SELECTION SORT:

1. Selection sort is very simple to understand and implement. Its algorithmic logic is straightforward and involves minimal code complexity,
2. **Stable Sorting**: Selection sort maintains the relative order of equal elements in the sorted array. If two elements have the same value, the one that appears first in the input will also appear first in the sorted output, making it a stable sorting algorithm.

3.Selection sort is an inplace sorting algotihm which means it does not require extra memory.

DISADVANTAGE OF SELECTION SORT:

1. It is inefficient: The time complexity of selection sort is O(n^2), which means that the running time of the algorithm increases quadratically with the size of the input array. This makes selection sort a poor choice for sorting large arrays.

2. It is not adaptive: An adaptive sorting algorithm is one whose performance improves if the input array is already partially sorted. Selection sort is not adaptive, which means that its performance is the same regardless of whether the input array is sorted or not.

APPLICATIONS OF SELECTION SORT:

1. . Educational Use: Selection Sort is often used as an introductory example of a sorting algorithm in computer science and programming courses. Its straightforward logic helps students understand the basic concepts of sorting algorithms.

2. , Selection Sort is used as a subroutine within larger algorithms. For instance, in certain cases where the list is only partially sorted, it might be more efficient to use a small number of Selection Sort steps within a larger sorting strategy.

3. : Selection Sort can be useful for sorting very small arrays or lists where the overhead of more complex sorting algorithms might outweigh their benefits. For such small datasets, the simplicity of Selection Sort might be preferred over more efficient algorithms.

PERFORMANCE:

TIME COMPLEXITY: Selection sort has a n average and worst time complexity of O(n^2).

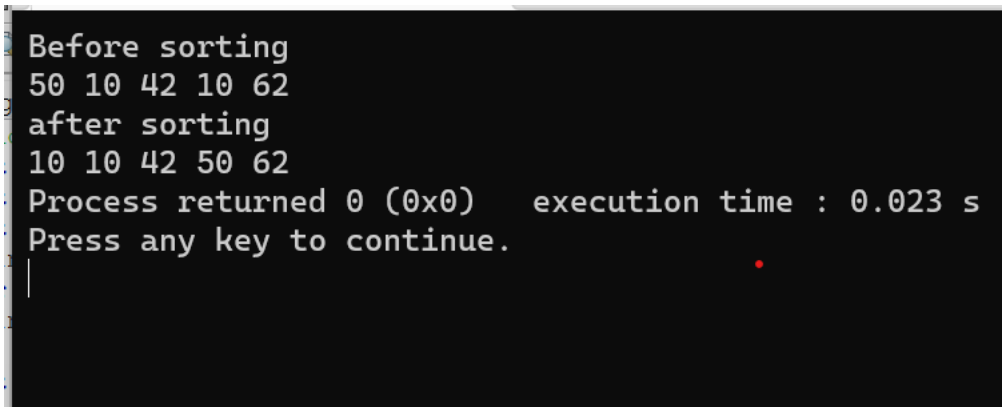
SPACE COMPLEXITY: It requires a space complexity of O(1) since it requires constant amount of additional memory for swapping.

PROGRAM:

```
#include<stdio.h>
int main(){
int temp,a[] = {50,10,42,10,62};
int n = sizeof(a)/sizeof(a[0]);
printf("Before sorting\n");
for(int i = 0;i<n;i++){
printf("%d ",a[i]);
}
int i,j,MinValue,MinIndex;
for(i=0;i<n-1;i++){
index=i;
for(j=i+1;j<n;j++){
if(a[MinIndex]>a[j]){
MinIndex=j;
}
}
if(MinIndex!=i){
temp=a[i];
a[i]=a[MinIndex];
a[MinIndex]=temp;
}
}
```

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

OUTPUT:



INSERTION SORT:

DESCRIPTION:

Insertion sort works on the concept of sorting by insertion.

It is performed by dividing the array into two parts ,sorted part and unsorted part of an array.

Initially the first element is considered as sorted part of an array and in each iteration it takes a value from unsorted portion and places in the correct position in the sorted part of the array.

After n-1 passes all the elements are in sorted order.

EXAMPLE OF INSERTION SORT:

Consider an unsorted array

0	1	2	3	4
30	4	37	29	55

PASS 1:

Here initially first element of the is considered as the sorted part

0	1	2	3	4
30	4	37	29	55

Consider the second element as the key element and check whether the key is smaller than the sorted element and place it in the correct position in the sorted portion

Key=a[1]=4

Key<a[0]->4<30(True)

0	1	2	3	4
4	30	37	29	55

PASS 2:

Now 4 and 30 are sorted

Now consider 37 as the key element by increasing key index by 1

Compare the key element(37) with the sorted list and place it in the correct position of the sorted list.

0	1	2	3	4
4	30	37	29	55

Key=a[2]=37

PASS 3:

The elements 4,30 37 are sorted.

0	1	2	3	4
4	30	37	29	55

Key=a[3]=29

Now compare the key element(29) with the elements in the sorted list and place the key element in the correct position of the sorted list.

0	1	2	3	4
4	29	30	37	55

PASS 4:

Here the elements 4,29,30,37 are sorted .

Key =a[4]=55

Compare the key element with the elements in the sorted list and place the key element to the correct position in the sorted list.

0	1	2	3	4
4	29	30	37	55

Here all the elements are sorted.

ALGORITHM OF INSERTION SORT:

1. Read the length of the array(n).
2. Create an array and enter the values into it.
3. Consider the first element of the array as sorted part .
4. Consider key as the first element in unsorted array.
5. Compare key with elements in the soerted portion of an array.
6. If elements in sorted part are greater than key the key is placed at required position in the sorted list.
7. In every iterartion the sorted array increases its elements by 1 and update key value for each iteration.
8. Repeat these steps until the array is sorted.

ADVANTAGES OF INSERTION SORT:

1. Insertion sort performs efficiently on small dataset elements (or) nearly sorted elements.
2. It is an adaptive sorting algorithm which requires less number of comparisions and swaps to sort elements of partially sorted array.
3. It is an inplace sorting algorithm where it do not require additional memory .
4. Insertion sort is simple and easy to understand.

DISADVANTAGES OF INSERTION SORT:

1. As the number of elements increases the number of comparisions also increases which leads to the poor performance of insertion sort.
2. Unlike bubble and selection sort it is not adaptive when it comes to the random data.

3. Insertion sort doesn't naturally lend itself to parallel processing or optimization. Other sorting algorithms can take advantage of parallel computing techniques to speed up sorting, which insertion sort lacks.
4. It is not suitable for external sorting.

APPLICATIONS OF INSERTION SORT:

1. Insertion sort is suitable for sorting small datasets or cases where data is mostly sorted.
2. **Debugging and Verification**: Due to its simplicity, insertion sort can be used for testing and debugging purposes to verify the correctness of other sorting algorithms.
3. Insertion sort takes part in the more complex sorting algorithms like Tim sort(used in python's built-in sorting).
4. Online Algorithms: It's used in scenarios where data is received in real-time and needs to be sorted as it arrives.
5. Real world Application of insertion sort is Tailors arrange shirts in a closet.

PERFORMANCES:

TIME COMPLEXITY: The best case time complexity of insertion sort is O(n) when the elements are already sorted and average and worst case time complexity is O(n^2).

SPACE COMPLEXITY: The space complexity of insertion sort is O(1) as it requires a constant amount of additional memory.

PROGRAM:

```
#include<stdio.h>
int main(){
int temp ,a[20];
int n ;
printf(“enter n value:”);
scanf(“%d”,&n);
printf("Before sorting\n");
for(int i = 0;i<n;i++){
printf("%d ",a[i]);
}
int i,j,key;
for(i=1; i<n; i++){
key = a[i];
j = i-1;
while(j>=0 && key<a[j]){
a[j+1] = a[j];
j--;
}
a[j+1] = key;
}
printf("\nafter sorting\n");
```

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

OUTPUT:

```
enter n value:5
enter the values of an array:25
29
4
37
30
Before sorting
25 29 4 37 30
after sorting
4 25 29 30 37
Process returned 0 (0x0)   execution time : 13.424 s
Press any key to continue.
```

MERGE SORT:

Merge sort is an efficient comparision based algorithm that follows divide and conquer principle.

It uses the concept of sorting by merging.

Merge sort follows three steps for sorting any unsorted array:

1.Divide

2.conquer

3.combine.

In merge sort the unsorted array is divided into smaller subarrays recursively until each subarray consists of one element.

The divided subarrays are merged in sorted manner by repeatedly comparing and merging the subarrays.

Merge sort has worst case time complexity of O(nlogn) so it works well on the large datasets too.

ALGORITHM :

1.Read the length of the array.

2.Create an array of length(n).

3.Read tha array and pass the array to a function Mergesort() which takes array first and last index as arguments.

4.Now calculate the mid in the unsorted array.

5.now in Mergesort function the array will be divided into 2 subarrays from first index to mid and mid to last index.

6.Now call a new function called Merge().

7.The Merge function sorts the array and combines the sub arrays.

8.The recursive function calls continuously until all the elements are sorted.

EXAMPLE FOR MERGE SORT:

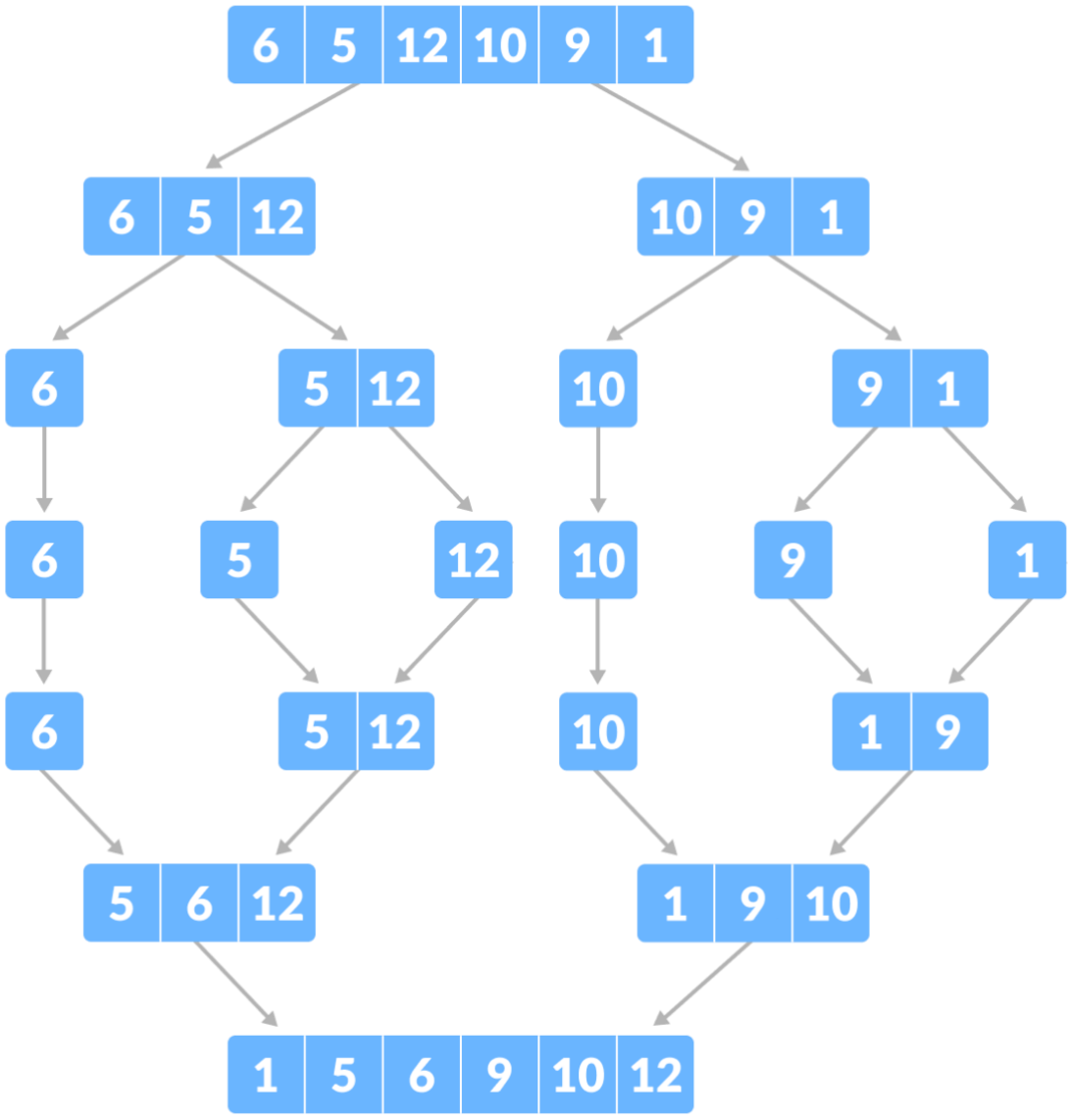
Consider an unsorted array

0	1	2	3	4	5
6	5	12	10	9	1

Now calculate the mid

Mid=(low+high)/2

Mid=(0+5)/2=2



Divide the array into subarrays a[low:mid] and a[mid+1:high]

0	1	2
6	5	12

3	4	5
10	9	1

Now further divide the left half and right half of the array into subarrays until they are divided into single elements

Left subarray:

0	1	2
6	5	12

6

5	12

6

5

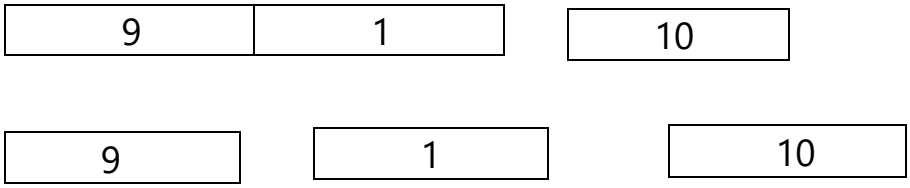
12

Now sort the elements in the left subarray and merge the elements.

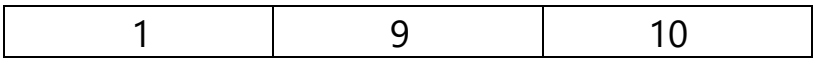
5	6	12

Consider the right subarray and divide it into subarrays containing only single element

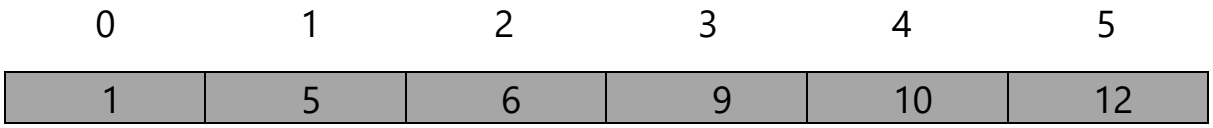
3	4	5
10	9	1



Now sort the elements in the right half of the array and merge the array



Now merge the left and right subarray by comparing and sorting recursively.



Here the array is sorted.

ADVANTAGES OF MERGE SORT:

1. It guarantees a stable sort, meaning that equal elements retain their relative order in the sorted output.
2. It has a consistent O(n log n) time complexity in all cases, making it efficient for large datasets.
3. **Parallelization**: Merge sort can be easily parallelized, allowing you to take advantage of multi-core processors or parallel computing environments to speed up sorting.

DISADVANTAGES OF MERGE SORT:

1. Merge sort requires additional memory space to store temporary subarrays during the sorting process.
2. It is not efficient for small datasets.

APPLICATIONS OF MERGE SORT:

1. It is used for sorting large datasets.

2. In e-commerce websites, merge sort can be used to sort and display products to customers. Products can be sorted by various criteria such as price, rating, or relevance.
3. **External Sorting**: In scenarios where data is too large to fit entirely into memory, merge sort is a key component of external sorting algorithms. It is used in applications like sorting large files, log files, or data sets stored on disk.
4. Many programming languages and libraries use Merge Sort for their built-in sorting functions.

PERFORMANCE:

Time complexity: O(nlogn)

Space complexity: O(n)

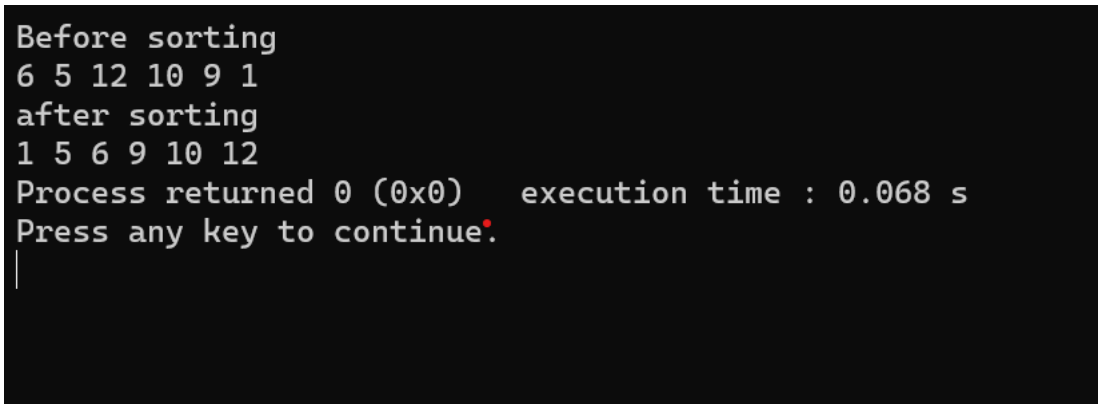
PROGRAM:

```
#include<stdio.h>
void mergesort(int a[],int l,int h){
    if(l<h){
        int mid = (l+h)/2;
        mergesort(a,l,mid);
        mergesort(a,mid+1,h);
        merge(a,l,mid,h);
    }
}
void merge(int arr[],int low,int mid,int high){
    int n1 = mid-low+1;
    int n2 = high-mid;
    int a[n1],b[n2];
    for(int i=0;i<n1;i++)
```

```
        a[i] = arr[low+i];
    for(int i=0;i<n2;i++)
        b[i] = arr[mid+i+1];
     int i=0,j=0,k=low;
    while(i<n1 && j<n2){
        if(a[i] > b[j]){
            arr[k] = b[j];
            k++;
            j++;
        }
        else{
arr[k] = a[i];
            k++;
            i++;
        }
    }
    while(i<n1){
        arr[k] = a[i];
        k++;
        i++;
    }
    while(j<n2){
        arr[k] = b[j];
            k++;
            j++;
```

```
    }
}
int main(){
    int a[] = {6,5,12,10,9,1};
    int n = sizeof(a)/sizeof(a[0]);
    printf("Before sorting\n");
    for(int i = 0;i<n;i++){
        printf("%d ",a[i]);
    }
    mergesort(a,0,n-1);
    printf("\nafter sorting\n");
    for(int i = 0;i<n;i++){
        printf("%d ",a[i]);
    }
}
```

OUTPUT:



QUICK SORT:

Quick sort is also known as partition-excahnge sort which uses the divide and conquer principle.

Quick sort is one of the fastest sorting algorithm because of it's average time complexity of O(nlogn).

The basic idea behind quick sort is to divide the unsorted list into two sublists, one with elements less than a chosen pivot element and the other with elements greater than the pivot. Then, recursively sort both sublists and combine them to form the sorted list.

PARTIONING:

Rearrange the elements in the array such that all elements less than or equal to the pivot are on the left side, and all elements greater than the pivot are on the right side. The pivot itself is in its final sorted position.

ALGORITHM :

1.Select the pivot element in the list.

2.construct two functions say qs() and part()

3. The part function sets the pivot element to the correct position as it was in a sorted array.
4. The qs function does the recursive function calls
5. qs() splits the array after the partition is done from the pivot +1 to highest index and lowest index to pivot -1

6. The sorting is done after the recursive function calls.

EXAMPLE:

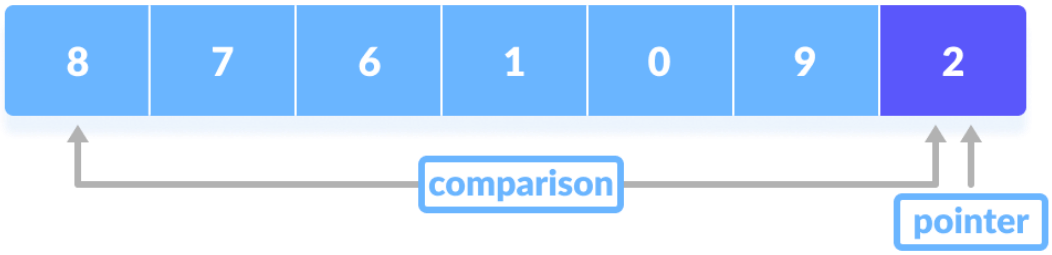
Consider an unsorted array

0	1	2	3	4	5	6
8	7	6	1	0	9	2

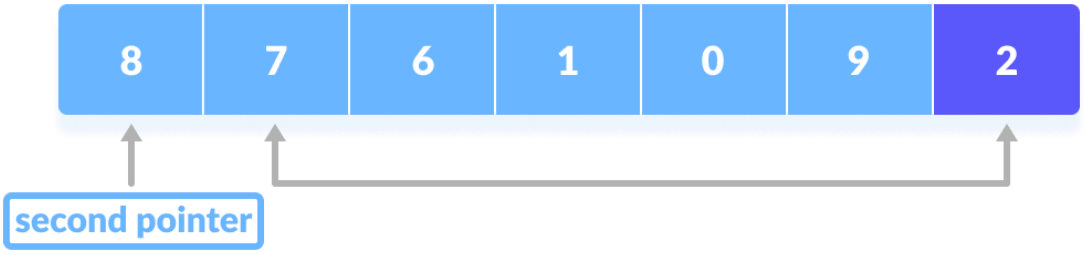
Select the right most element of an array as pivot element.

Elements of the array are arranged so that elements smaller than pivot are placed on the left and the elements greater than pivot are placed on the right.

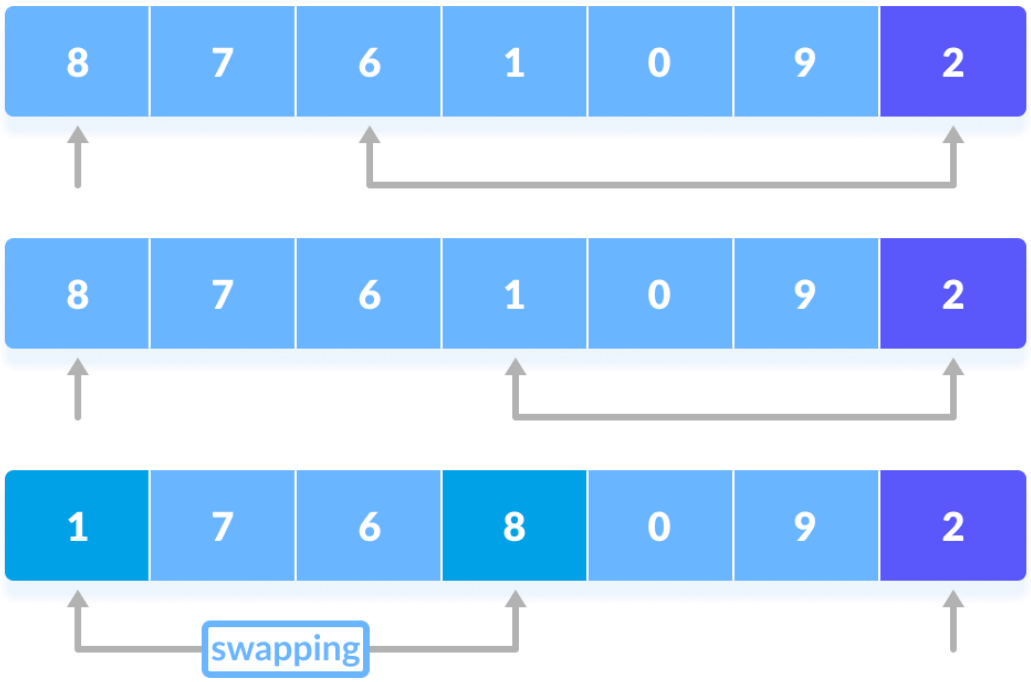
A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.



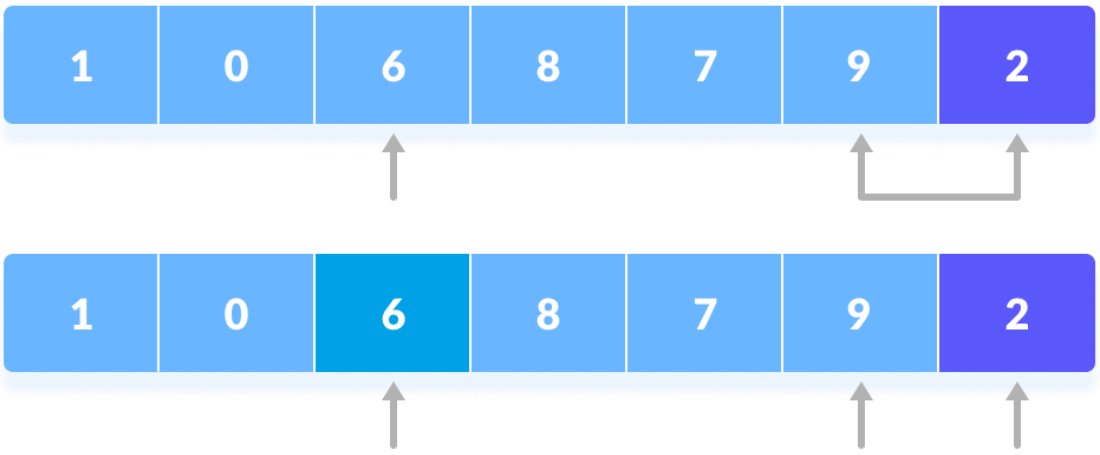
If the element is greater than the pivot element, a second pointer is set for that element.



Pivot is compared with other elements.



The process goes on until the second last element is reached.



Finally, the pivot element is swapped with the second pointer.

ADVANTAGES OF QUICK SORT:

1.It is an inplace sorting algorithm so it does not require additional memory for data storage.

2. Quick sort can be adaptive, meaning that it performs well on partially sorted data.

3. Quicksort is a cache-friendly algorithm.

4. It works rapidly and effectively.

DISADVANTAGES OF QUICK SORT:

1.it is difficult to implement when recursion does not takes place as it is a recursive process.

2.not efficient for small datasets.

3.It's not a stable sorting algorithm as it changes the relative position Of elements after sorting.

APPLICATIONS OF QUICK SORT:

1.Sorting large datasets.

2. Quick Sort is widely used for general-purpose sorting tasks, especially when efficiency is crucial.

3. Compilers may use quick sort in optimization stages for code generation and data manipulation.

4. It is tail -recursive and hence all the call optimization can be done.

PERFORMANCE:

1.TIME COMPLEXITY: O(nlogn)

Average time complexity: O(nlogn)

Worst case time complexity: o(n^2)

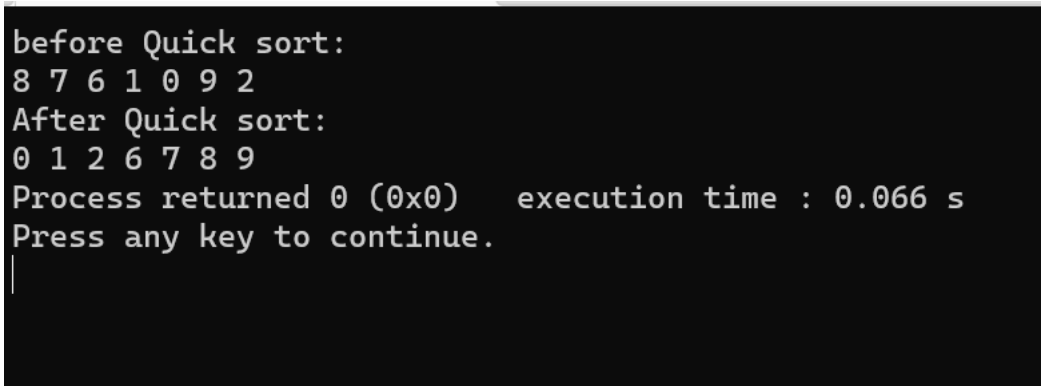
2.SPACE COMPLEXITY: O(logn)

PROGRAM:

```
#include<stdio.h>
int partition(int a[],int low,int high){
    int i=low-1,pivot = a[high];
    for(int j=low;j<high;j++){
        if(a[j] <= pivot){
            i++;
            int temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }
    int temp = a[high];
    a[high] = a[i+1];
    a[i+1] = temp;
    return i+1;
}
void quicksort(int a[], int l,int h){
    if(l<h){
        int p = partition(a,l,h);
        quicksort(a,l,p-1);
        quicksort(a,p+1,h);
    }
}
int main(){
```

```
    int a[] = {9,6,2,1};
    int n = sizeof(a)/sizeof(a[0]);
    printf("before Quick sort: \n");
    for(int i =0;i<n;i++)
        printf("%d ",a[i]);
    quicksort(a,0,n-1);
    printf("\nAfter Quick sort: \n");
    for(int i =0;i<n;i++)
        printf("%d ",a[i]);
}
```

OUTPUT:



COMPARISION OF 5 SORTING ALGORITHMS:

	Bubble sort	Selection sort	Insertion sort	Merge sort	Quick sort
No:of iterations	n-1	n-1	n-1	n-1	0 to n-1
No:of comparisions	N-1 - pass1 N-2 - pass-2 at most n-1+n-2+n-3 .....+1	N-1 - pass1 N-2 - pass-2 at most n-1+n-2+n-3 .....+1	Depends on the elements at most n-i comparisons For each pass i where 1<i<=n-1	n + (n – 1) + (n – 2) + (n – 3) + … + 1	n + (n – 1) + (n – 2) + (n – 3) + … + 1
No:of swaps	n-1	n-1	Depends on the elements at most n-i comparisons For each pass i where 1<i<=n-1	n/2+n/4+.. .	n-1
Time complexity	Best Case: 0(n) Avgcase: $0(n^2)$	Best Case: $0(n^2)$ Avg case: $0(n^2)$ Worst Case:	Best Case: 0(n) Avg case:$0(n^2)$	Best Case: 0(nlogn) Avg case: 0(nlogn) Worst Case:	BestCase: 0(nlogn) Avg case: 0(nlogn) Worst

	Worst Case: $0(n^2)$	$0(n^2)$	Worst Case: $0(n^2)$	0(nlogn)	Case: $0(n^2)$
Space complexity	O(1)	O(1)	O(1)	O(n)	O(1)

VIVA QUESTIONS:

Q. What is Bubble Sort?

A. Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

Q. What is the time complexity of Bubble Sort in the worst case?

A. The worst-case time complexity of Bubble Sort is O(n^2), where 'n' is the number of elements in the array.

Q. Is Bubble Sort stable or unstable?

A3 Bubble Sort is a stable sorting algorithm, as it preserves the relative order of equal elements.

Q. What is Selection Sort?

A. Selection Sort is a simple sorting algorithm that repeatedly selects the minimum element from the unsorted portion of the array and places it at the beginning.

Q. What is the time complexity of Selection Sort in the worst case?

A. The worst-case time complexity of Selection Sort is O(n^2).

Q. Is Selection Sort stable or unstable?

A. Selection Sort is an unstable sorting algorithm, as it may change the order of equal elements.

Q. What is Insertion Sort?

A. Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time by repeatedly moving larger elements to the right.

Q. What is the time complexity of Insertion Sort in the worst case?

A. The worst-case time complexity of Insertion Sort is O(n^2).

Q. Is Insertion Sort stable or unstable?

A. Insertion Sort is a stable sorting algorithm, as it preserves the relative order of equal elements.

Q What is Merge Sort?

A. Merge Sort is a divide-and-conquer sorting algorithm that divides an array into smaller subarrays, sorts them, and then merges the sorted subarrays.

Q. What is the time complexity of Merge Sort in the worst case?

A. The worst-case time complexity of Merge Sort is O(n log n).

Q. Is Merge Sort stable or unstable?

A. Merge Sort is a stable sorting algorithm, as it preserves the relative order of equal elements.

Q. What is Quick Sort?

A. Quick Sort is a divide-and-conquer sorting algorithm that selects a pivot element, partitions the array into subarrays, and recursively sorts them.

Q. Is Quick Sort stable or unstable?

A3. Quick Sort is an unstable sorting algorithm, as it may change the order of equal elements.

EXPERIMENT NO:1

RECURSION:

Recursion is used to divide a complex into simpler and similar subproblems that can be solved by calling the same function.

Recursion is a technique where the function calls itself inorder to solve the problem during the execution.

Each recursive call works on a smaller subproblem until the base case is reached and the results are combined to solve the original problem.

Recursion is based on the concept of divide and conquer.

ESSENTIAL COMPONENTS OF RECURSION:

The two components for recursion are :

1.Base Case.

2.Recursive Case.

BASE CASE:

The Base case acts as the stopping condition for the recursion. it does not allow recurion for indefinite number of times and provide the solution for the most basic version of the problem.

RECURSIVE CASE:

the recursive case(s) determine how the problem is divided and solved in a recursive fashion.

When a function encounters a recursive case, it calls itself with a smaller or modified version of the original problem.

The return values from each level of recursion are combined to obtain the final result.

DIRECT RECURSIVE ALGORITHM:

Recursion initiated by the function calling itself directly within the body.

Here base case and recursive case are present in the same function.

It is simpler to write and understand the direct recursive algorithm easily.

INDIRECT RECURSIVE ALGORITHM:

In indirect recursion, a function calls another function which then calls the first function again.

Here the base case and recursive case are defined in the separate functions.

LIMITATIONS OF RECURSION:

If recursive solution does not run with in space and time complexities we should not use it.

Recursion should always be used when it is suitable for the program.

It should be shorter and understandable.

In some cases, iterative solutions might be more efficient than their recursive counterparts. Iterative solutions often use loops, which have less overhead compared to function calls.

Recursion may leads to inefficiency as it may involve repeated calculations of the same subproblems.

Memory usage: Each recursive call adds a new entry to the stack,it may lead to stack overflow if there are two many nested calls

Example:

num	num*fact(num-1)
5	
4	4 * fact(3)
3	3 * fact(2)
2	2 * fact(1)
1	1 * fact(0)

FACTORIAL OF A GIVEN NUMBER:

ALGORITHM:

Step 1: Read the number

Step 2: Start the calculation of factorial for the given number.

Step 3: check whether the number is 0.

Step 4: if number is 0 return 1.(base case)

Step 5: if number not equal to 0,calculate the factorial of (number-1) recursively.

Step 6: multiply the number with the result obtained in step 3.

Step 7:return the result.

PROGRAM FOR THE FACTORIAL OF GIVEN NUMBER USING RECURSION:

```
#include <stdio.h>
int factorial(int n) {
 if (n == 0) {
 printf("factorial(%d) = 1\n", n);
 return 1;
 }
 int result = n * factorial(n - 1);
 printf("factorial(%d) = %d * factorial(%d) = %d\n", n, n,
n - 1, result);
 return result;
}
int main() {
 int num;
 printf("Enter a non-negative integer: ");
 scanf("%d", &num);
 if (num < 0) {
 printf("Factorial is not defined for negative
numbers.\n");
 } else {
```

```
 int result = factorial(num);
 printf("Factorial of %d is %d\n", num, result);
}
return 0
}
```

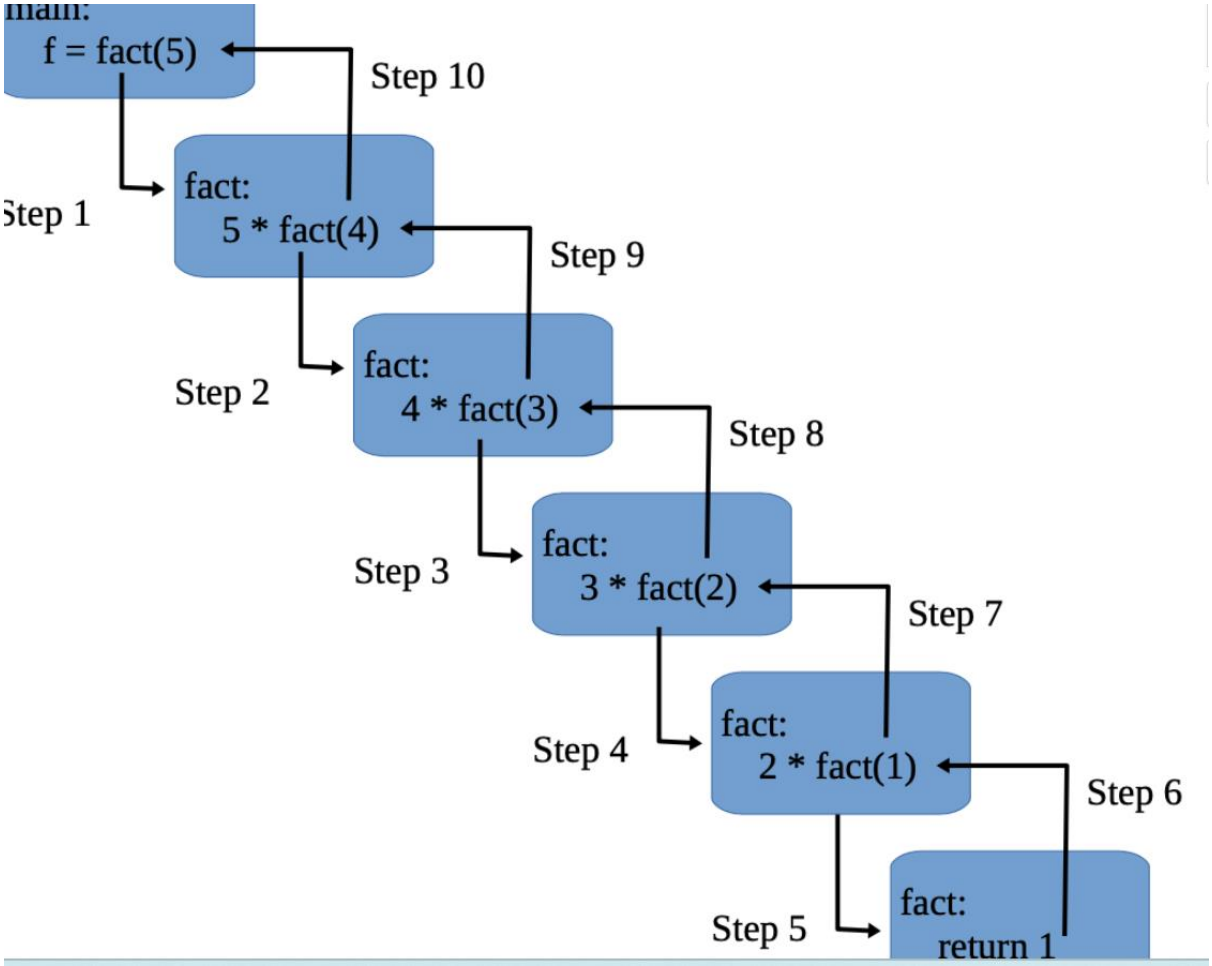
OUTPUT:

```
Enter a non-negative integer: 5
factorial(0) = 1
factorial(1) = 1 * factorial(0) = 1
factorial(2) = 2 * factorial(1) = 2
factorial(3) = 3 * factorial(2) = 6
factorial(4) = 4 * factorial(3) = 24
factorial(5) = 5 * factorial(4) = 120
Factorial of 5 is 120
```

OUTPUT:

```
Enter a non-negative integer: 5
factorial(0) = 1
factorial(1) = 1 * factorial(0) = 1
factorial(2) = 2 * factorial(1) = 2
factorial(3) = 3 * factorial(2) = 6
factorial(4) = 4 * factorial(3) = 24
factorial(5) = 5 * factorial(4) = 120
Factorial of 5 is 120

Process returned 0 (0x0)   execution time : 7.630 s
Press any key to continue.
```

FLOWCHART:



BINARY SEARCH USING RECURSION:

ALGORITHM:

STEP-1: Define a function binarySearchRecursive that takes an array,

left index, right index, and the target element as parameters.

STEP-2: Calculate the middle index: mid = (left + right) / 2.

STEP-3: Check if the middle element is equal to the target element:

• If yes, return the middle index.

• If no, proceed to the next steps.

STEP-4: Check if the middle element is greater than the target element:

• If yes, recursively call binarySearchRecursive with parameters:

array, left, mid - 1, and the target element.

• If no, recursively call binarySearchRecursive with parameters:

array, mid + 1, right, and the target element.

STEP-5: If the left index becomes greater than the right index, return a

value indicating that the target element is not found in the array.

EXAMPLE:

Consider an array

0	1	2	3	4	5	6
10	20	30	40	50	60	70

.

Let the search element be 20

Search=20

Low=0 and high=n-1=7-1=6

Step 1:

The search element (20) is compared with the middle element(40).

0	1	2	3	4	5	6
10	20	30	40	50	60	70
			20			

If(low<=high) (0<6) (true)

Mid=(low+high)/2=3

a[Mid]==search (40==20) (false)

Both the elements are not matching and the element 20 is smaller than the middle element so the left subarray is searched.

High=mid-1=3-1=2

Step 2:

The search element (20) is compared is the middle element

0	1	2	3	4	5	6
10	20	30	40	50	60	70

20

If(Low<high) (0<2) (true)

Mid=(low+high)/2=1

a[Mid]==search (20==20)

both the elements are matching so return index 1

PROGRAM FOR BINARY SEARCH USING RECURSION:

```
#include <stdio.h>
int binarySearchRecursive(int a[], int low, int high, int search) {
 if (low <= high) {
 int mid = low+ (high - low) / 2;
 if (a[mid] == search) {
 return mid; // Target found
 }
 if (a[mid] > search) {
 return binarySearchRecursive(a, low, mid - 1, search);
 }
```

```
 return binarySearchRecursive(a, mid + 1, high, search);
 }
 return -1; // Target not found
}
int main() {
 int length;
 printf("Enter the number of elements in the array: ");
 scanf("%d", &length);
int a[length];
 printf("Enter the sorted array elements:\n");
 for (int i = 0; i < length; ++i) {
 scanf("%d", &a[i]);
 }
 int search;
 printf("Enter the target value to search for: ");
 scanf("%d", &search);
 int result = binarySearchRecursive(array, 0, length - 1,
target);
 if (result != -1) {
 printf("Target found at index %d.\n", result);
 } else {
```

```
 printf("Target not found in the array.\n");
 }
 return 0;
}
```

OUTPUT:

Here element is found

```
Enter the number of elements in the array: 7
Enter the sorted array elements:
10
20
30
40
50
60
70
Enter the target value to search for: 20
Target found at index 1.

Process returned 0 (0x0)   execution time : 23.203 s
Press any key to continue.
```

OUTPUT 2:

Here element is not found

```
Enter the number of elements in the array: 5
Enter the sorted array elements:
10
20
30
40
50
Enter the target value to search for: 45
Target not found in the array.

Process returned 0 (0x0)   execution time : 14.866 s
Press any key to continue.
```

TOWERS OF HANOI:

Tower of Hanoi is a mathematical puzzle where we have three rods (A, B, and C) and N disks.

the smallest disk is placed on the top and finally they are on rod A.

The objective of the puzzle is to move the entire stack to another rod (here considered C)(destination).

RULES:

Only one disk can be moved at a time.

Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack

No disk may be placed on top of a smaller disk

ALGORITHM FOR TOWERS OF HANOI:

Assign labels A, B, and C to the three pegs. These labels represent the distinct positions where disks can be placed.

If the total number of disks is even:

- Move a disk between pegs A and B, complying with the rules.
- Move a disk between pegs A and C, adhering to the rules.

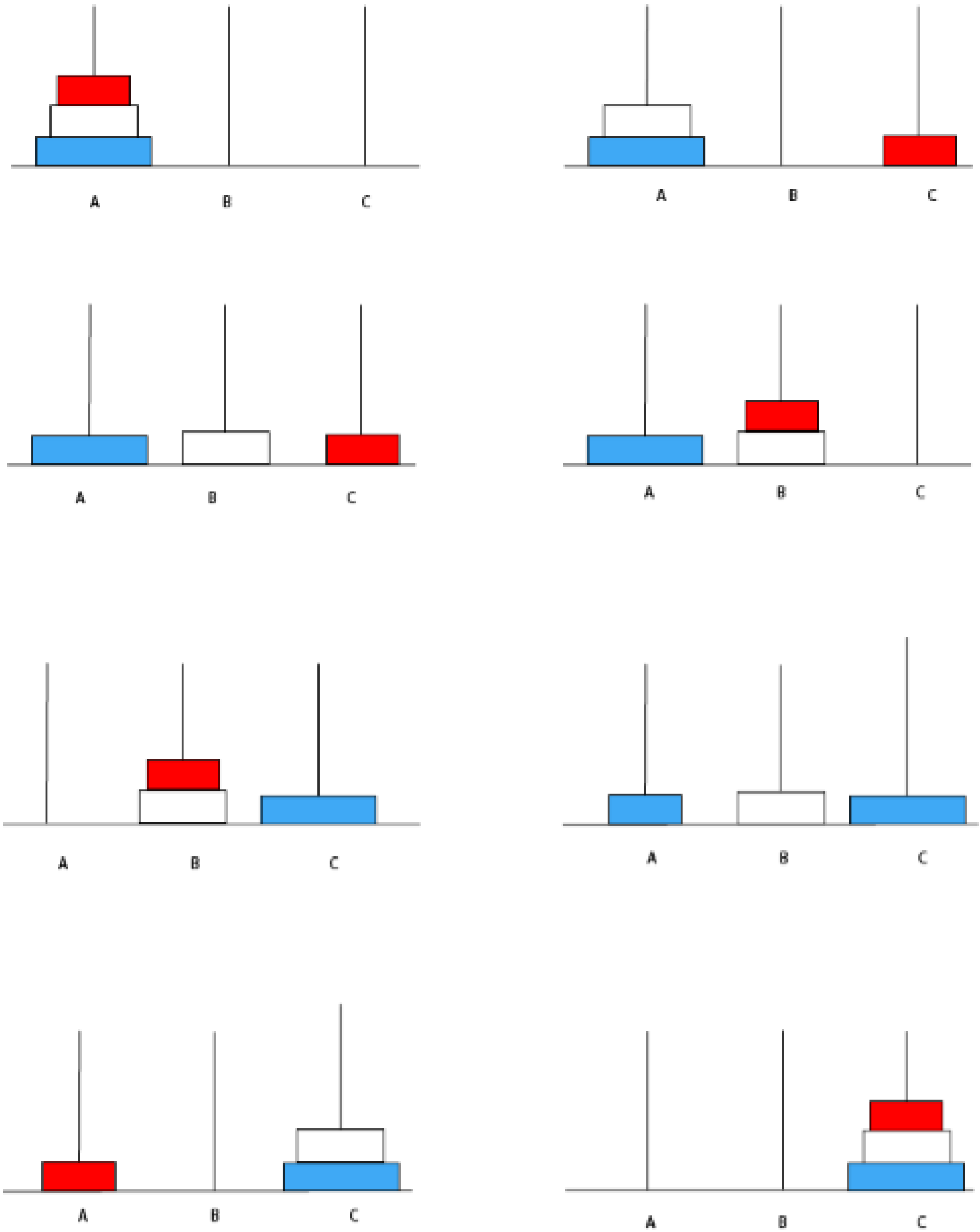
• Move a disk between pegs B and C, following the rules.

• Continue these three moves in a repeating pattern until the puzzle is solved.

EXAMPLE:

Assuming we have three pegs named A, B, and C, and n disks

Let us we have three disks stacked on a peg



PASS 1: Move the top disk from peg A to peg C.

PASS 2: Move the second disk from peg A to peg B.

PASS 3: Move the top disk from peg C to peg B.

PASS 4: Move the top disk from peg A to peg C.

PASS 5: Move the disk from peg B to peg A.

PASS 6: Move the top disk from peg B to peg C.

PASS 7: Move the top disk from peg A to peg C.

the Towers of Hanoi problem with n disks can be calculated using the formula: 2 ** n – 1

PROGRAM FOR TOWERS OF HANOI:

```
#include <stdio.h>

void towersOfHanoi(int n, char source, char auxiliary,
char target) {

 if (n == 1) {
```

```
 printf("Move disk 1 from %c to %c\n", source, target);
 return;
 }
 towersOfHanoi(n - 1, source, target, auxiliary);
 printf("Move disk %d from %c to %c\n", n, source,
target);
 towersOfHanoi(n - 1, auxiliary, source, target);
}
int main() {
 int num_disks;
 printf("Enter the number of disks: ");
 scanf("%d", &num_disks);
 if (num_disks < 1) {
 printf("Number of disks should be at least 1.\n");
 } else {
 towersOfHanoi(num_disks, 'A', 'B', 'C');
}
return 0;
}
```

OUTPUT:

```
Enter the number of disks: 3
```

Move disk 1 from A to C

Move disk 2 from A to B

Move disk 1 from C to B

Move disk 3 from A to C

Move disk 1 from B to A

Move disk 2 from B to C

Move disk 1 from A to C

OUTPUT:

```
Enter the number of disks: 3
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C

Process returned 0 (0x0)   execution time : 2.530 s
Press any key to continue.
```

Difference between Recursion and Iteration:

Iteration	**Recursion**
Recursion uses the selection structure.	Iteration uses the repetition structure.

Infinite recursion occurs if the step in recursion doesn't reduce the problem to a smaller problem. It also becomes infinite recursion if it doesn't convert on a specific condition. This specific condition is known as the base case.	An infinite loop occurs when the condition in the loop doesn't become False ever.
The system crashes when infinite recursion is encountered.	Iteration uses the CPU cycles again and again when an infinite loop occurs.
Recursion terminates when the base case is met.	Iteration terminates when the condition in the loop fails.
Recursion is slower than iteration since it has the overhead of maintaining and updating the stack.	Iteration is quick in comparison to recursion. It doesn't utilize the stack.
Recursion uses more memory in comparison to iteration.	Iteration uses less memory in comparison to recursion.
Recursion reduces the size of the code.	Iteration increases the size of the code.

VIVA QUESTIONS:

1. **What is recursion?**

- Recursion is a programming technique where a function calls itself in order to solve a problem. It involves breaking down a problem into smaller instances of the same problem until a base case is reached.

2. **What is a base case in recursion?**

- A base case is the condition that determines when the recursive function stops calling itself. It provides a way to exit the recursion and prevent infinite loops.

3. **Explain the two main components of a recursive function.**

- Base case: It defines the simplest scenario where the function does not call itself and returns a result directly.
- Recursive case: It defines how the function calls itself with modified arguments to break down the problem into smaller subproblems.

4. **What is the difference between direct recursion and indirect recursion?**

- Direct recursion occurs when a function directly calls itself.
- Indirect recursion occurs when there is a chain of function calls where the last function eventually calls the first function to complete the cycle.

5. **What are the advantages of using recursion?**

- Recursion can lead to elegant and concise code for problems that have a recursive structure. It's particularly useful for tasks like traversing tree structures.

6. What are the disadvantages of using recursion?

- Recursive solutions can be less efficient in terms of memory and execution time compared to iterative solutions due to the overhead of function calls and maintaining the call stack.

PROGRAM ON SINGLY LINKED LIST:

```
#include<stdio.h>
#include<stdlib.h>
struct node{
int data;
struct node *next;
};
struct node *head;
int count=0;
void insert_begin(){
   int value;
   printf("enter data to be inserted");
   scanf("%d",&value);
   struct node *temp;
   temp = (struct node *)malloc(sizeof(struct node *));
   if(temp==NULL)
      printf("memory insufficient\n");
   else
   {
       temp->data=value;
    if(head==NULL){
          temp->next=NULL;
```

```
        }


    else{
            temp->next=head;
        }
        head=temp;
        count++;
    }
}
void display(){
if(head==NULL)
    printf("list is empty\n");
else{
    struct node *p;
    p=head;
    while(p->next!=NULL)
    {
        printf("%d->",p->data);
        p=p->next;
    }
    printf("%d\n",p->data);
}
```

```
}
void insert_end(){
    int value;
  printf("enter data to be inserted");
    scanf("%d",&value);
    struct node *temp;
    temp = (struct node *)malloc(sizeof(struct node *));
    if(temp==NULL)
        printf("memory insufficient\n");
    else{
        temp->data=value;
        temp->next=NULL;
      if(head==NULL){
            head=temp;
      }
      else{
          struct node *p;
          p=head;
          while(p->next!=NULL)
              p = p->next;
          p->next=temp;
      }
```

```
      count++;
    }
}
void insert_position(){
    int value,loc,i;
    printf("enter which location data to be inserted");
    scanf("%d",&loc);
    if(loc>0 && loc<=count+1){
    struct node *temp;
    temp = (struct node *)malloc(sizeof(struct node *));
    if(temp==NULL)
        printf("memory insufficient\n");
    else
    {
      if(loc==1)
          insert_begin();
      else if(loc==count+1)
            insert_end();
      else
      {
          printf("enter data to be inserted");
          scanf("%d",&value);
```

```
            struct node *p;
            p=head;
            for(i=1;i<=loc-2;i++)
              p = p->next;
            temp->data=value;
          temp->next = p->next;
            p->next=temp;
            count++;
          }
       }
       }
       else
          printf("cant insert in the given location");
}
void delete_begin(){
if(head==NULL)
    printf("list is empty can not perofrm delete operation");
else{
    struct node *p;
    p=head;
    head = head->next;
    free(p);
```

```
    count--;
    printf("first node is deleted successfully");
}
}


void delete_end(){
if(head==NULL)
    printf("list is empty can not perform delete operation\n");
else{
    struct node *p,*q;
     p = head;
     while(p->next->next != NULL)
        p=p->next;
     q=p;
     p=p->next;
     q->next=NULL;
     free(p);
     count--;
     printf("last node is deleted successfully\n");
}
}
```

```
void delete_position(){
    int loc,i;
    printf("enter which location data to be inserted");
    scanf("%d",&loc);
    if(loc>=1 && loc<=count){
        if(loc==1)


       delete_begin();
        else if(loc==count)
            delete_end();
        else{
        struct node *p,*q;
        p=head;
        for(i=1;i<=loc-2;i++)
            p=p->next;
        q=p;
        p=p->next;
        q->next=p->next;
        free(p);
        count--;
        printf("deletion is performed at given location");
        }
```

```
    }
    else
        printf("deletion cannot be performed in given location\n");
}
void search(){
int value,loc=1,found=0;
printf("enter the data to search:");
scanf("%d",&value);
struct node *p;
p=head;
while(p->next!= NULL){
    if(p->data==value)
    {
         printf("found the data at position %d",loc);
         found=1;
        break;
    }
    loc++;
    p=p->next;
}
if(p->data==value && found==0)
    {
```

```
        printf("found the data at position %d",loc);
        found=1;
    }
if(found==0)
    printf("data is not found\n");
}
int main(){
int choice;
while(1){
   printf("\n1. insert at begining\n");
   printf("2. display\n");
   printf("3. print number of nodes\n");
   printf("4.insert at end\n");
   printf("5.insert at position\n");
   printf("6. delete at the begining\n");
   printf("7. delete at the end\n");
   printf("8. delete at position\n");
   printf("9. Search for data\n");
   printf("default. exit\n");
   printf("enter the choice:");
   scanf("%d",&choice);
   switch(choice){
```

```
case 1: insert_begin();
        break;
case 2: display();
        break;
case 3: printf("no of nodes=%d\n",count);
        break;
case 4: insert_end();
        break;
case 5: insert_position();
        break;

case 6: delete_begin();
        break;
case 7: delete_end();
         break;
case 8: delete_position();
          break;
case 9: search();
         break;
default: exit(0);
  }
}
```

```
return 0;

}
```

OUTPUT:

1. insert at begining

2. display

3. print number of nodes

4.insert at end

5.insert at position

6. delete at the begining

7. delete at the end

8. delete at position

9. Search for data

default. exit

enter the choice:1

enter data to be inserted5

1. insert at begining

2. display

3. print number of nodes

4.insert at end

5.insert at position

6. delete at the begining

7. delete at the end

8. delete at position

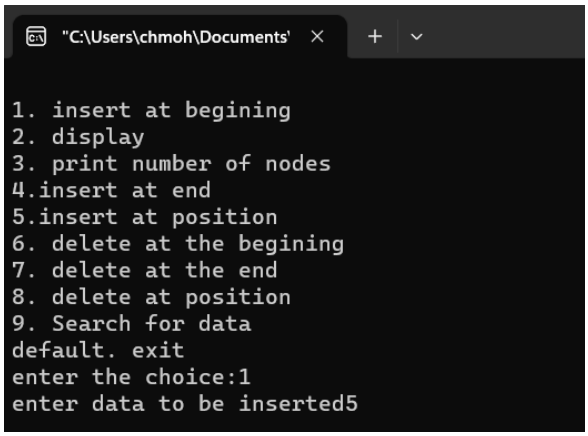
9. Search for data

default. exit

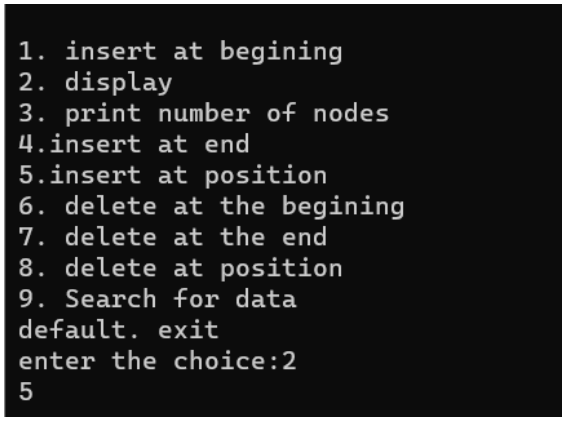
enter the choice:3

no of nodes=1

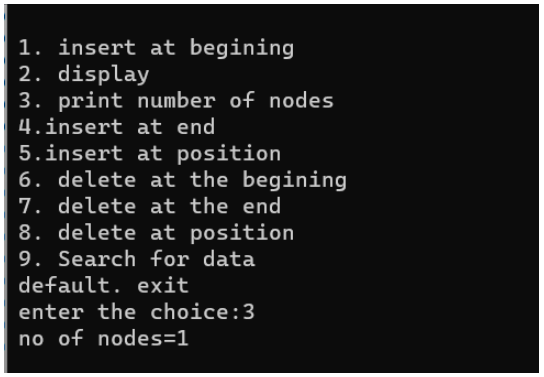
1.Data insert at beginning:



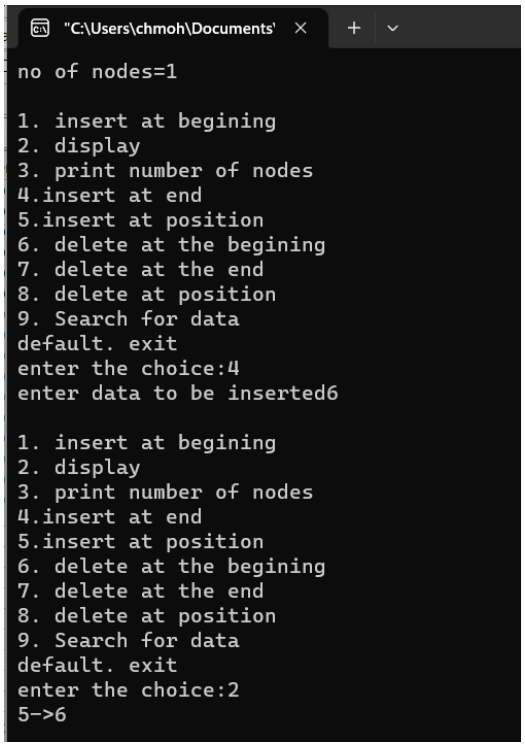
2.Display:



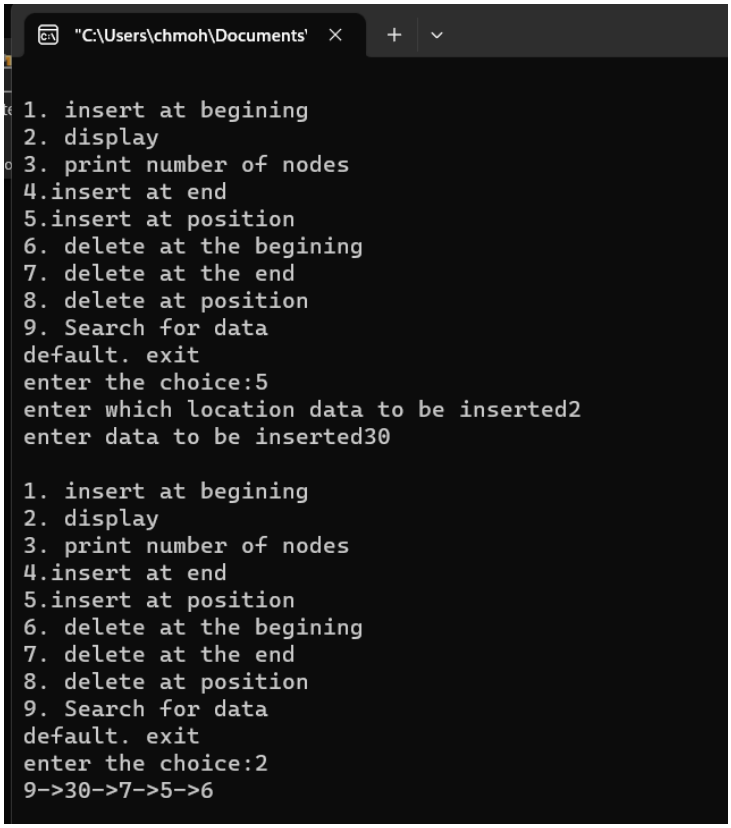
3.Print number of nodes:



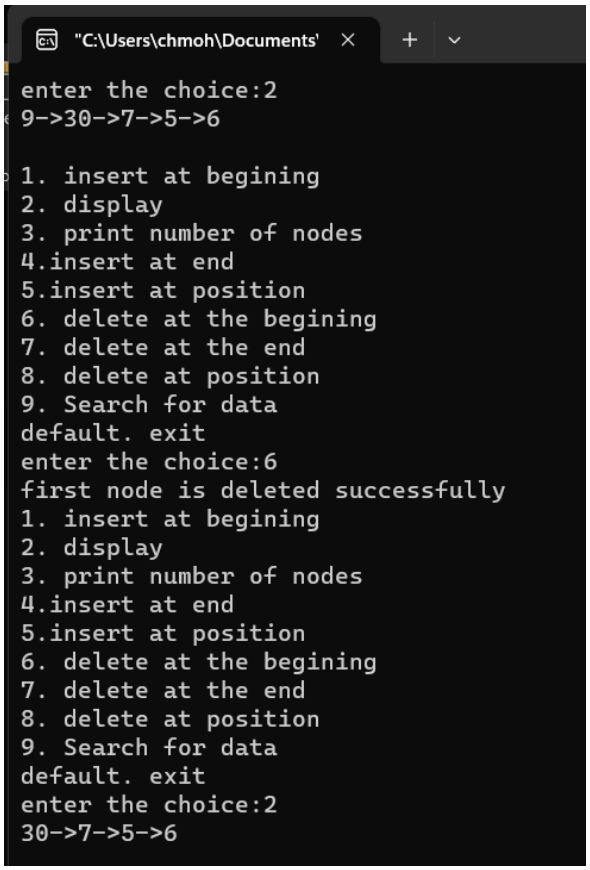
4.insert at end:



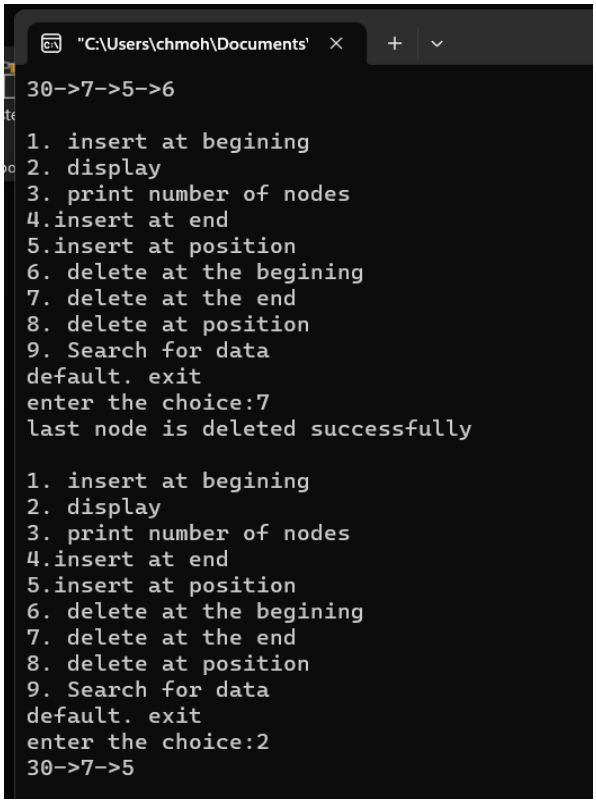
5.Insert at position:



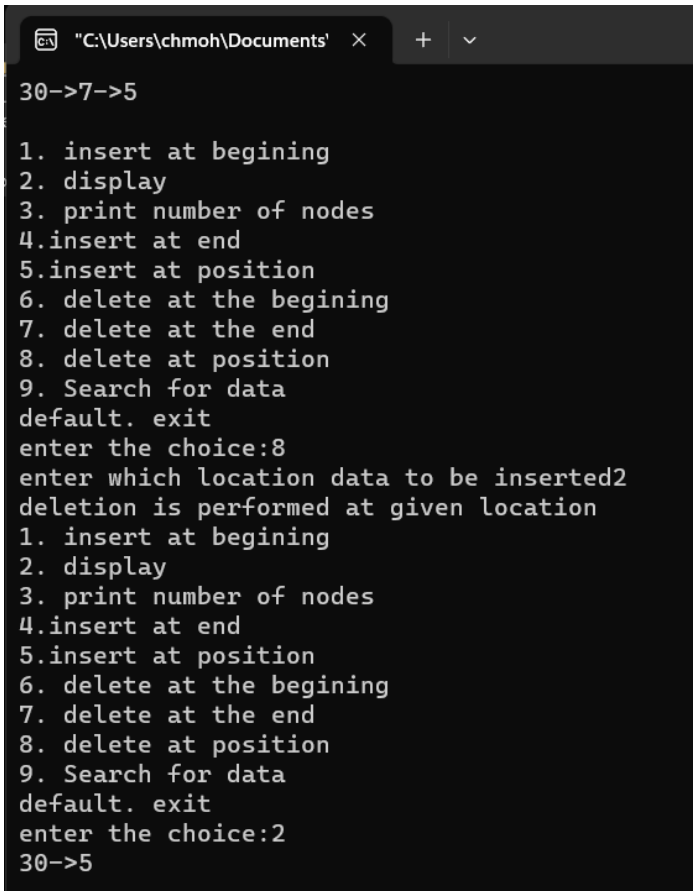
6.Delete at the beginning:



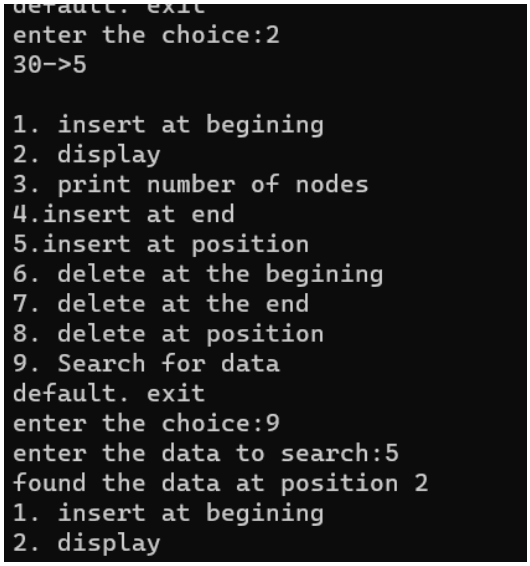
7.Delete at the end:



8.Delete at the position:



9.Search for data:



PROGRAM ON DOUBLY LINKED LIST:

```
#include <stdio.h>
#include <stdlib.h>
struct node {
    int data;
    struct node* next;
    struct node* prev;
};
struct node* head = NULL;
int count = 0;
void insert_begin() {
    int value;
    printf("Enter data to be inserted: ");
    scanf("%d", &value);
    struct node* temp;
    temp = (struct node*)malloc(sizeof(struct node));
    if (temp == NULL)
        printf("Memory insufficient\n");
    else {
        temp->data = value;
        if (head == NULL) {
            temp->next = NULL;
            temp->prev = NULL;
        } else {
            temp->next = head;
            temp->prev = NULL;
            head->prev = temp;
```

```
        }
        head = temp;
        count++;
    }
}
void display() {
    if (head == NULL)
        printf("List is empty\n");
    else {
        struct node* p;
        p = head;
        while (p->next != NULL) {
            printf("%d <-> ", p->data);
            p = p->next;
        }
        printf("%d\n", p->data);
    }
}
void insert_end() {
    int value;
    printf("Enter data to be inserted: ");
    scanf("%d", &value);
    struct node* temp;
    temp = (struct node*)malloc(sizeof(struct node));
    if (temp == NULL)
        printf("Memory insufficient\n");
    else {
        temp->data = value;
```

```
        temp->next = NULL;
        if (head == NULL) {
            temp->prev = NULL;
            head = temp;
        } else {
            struct node* p;
            p = head;
            while (p->next != NULL)
                p = p->next;
            p->next = temp;
            temp->prev = p;
        }
        count++;
    }
}
void insert_position() {
    int value, loc, i;
    printf("Enter which location data to be inserted: ");
    scanf("%d", &loc);
    if (loc > 0 && loc <= count + 1) {
        struct node* temp;
        temp = (struct node*)malloc(sizeof(struct node));
        if (temp == NULL)
            printf("Memory insufficient\n");
        else {
            if (loc == 1)
                insert_begin();
            else if (loc == count + 1)
```

```
            insert_end();
        else {
            printf("Enter data to be inserted: ");
            scanf("%d", &value);
            struct node* p;
            p = head;
            for (i = 1; i <= loc - 2; i++)
                p = p->next;
            temp->data = value;
            temp->next = p->next;
            p->next->prev = temp;
            temp->prev = p;
            p->next = temp;
            count++;
        }
    }
  } else
    printf("Can't insert in the given location\n");
}
void delete_begin() {
  if (head == NULL)
    printf("List is empty, can not perform delete operation\n");
  else {
    struct node* p;
    p = head;
    head = head->next;
    if (head != NULL)
      head->prev = NULL;
```

```
        free(p);
        count--;
        printf("First node is deleted successfully\n");
    }
}
void delete_end() {
    if (head == NULL)
        printf("List is empty, can not perform delete operation\n");
    else {
        struct node* p;
        p = head;
        while (p->next->next != NULL)
            p = p->next;
        struct node* q;
        q = p;
        p = p->next;
        q->next = NULL;
        free(p);
        count--;
        printf("Last node is deleted successfully\n");
    }
}
void delete_position() {
    int loc, i;
    printf("Enter which location data to be inserted: ");
    scanf("%d", &loc);
    if (loc >= 1 && loc <= count) {
        if (loc == 1)
```

```
            delete_begin();
        else if (loc == count)
            delete_end();
        else {
            struct node* p, * q;
            p = head;
            for (i = 1; i <= loc - 2; i++)
                p = p->next;
            q = p;
            p = p->next;
            q->next = p->next;
            p->next->prev = q;
            free(p);
            count--;
            printf("Deletion is performed at given location\n");
        }
    } else
        printf("Deletion cannot be performed in given location\n");
}
void search() {
    int value, loc = 1, found = 0;
    printf("Enter the data to search: ");
    scanf("%d", &value);
    struct node* p;
    p = head;
    while (p->next != NULL) {
        if (p->data == value) {
            printf("Found the data at position %d\n", loc);
```

```
            found = 1;
            break;
        }
        loc++;
        p = p->next;
    }
    if (p->data == value && found == 0) {
        printf("Found the data at position %d\n", loc);
        found = 1;
    }
    if (found == 0)
        printf("Data is not found\n");
}
void print_reverse() {
    if (head == NULL) {
        printf("List is empty\n");
    } else {
        struct node* p = head;
        while (p->next != NULL) {
            p = p->next;
        }

        printf("Reversed List: ");
        while (p != NULL) {
            printf("%d <-> ", p->data);
            p = p->prev;
        }
        printf("\n");
```

```
    }
}

int main() {
    int choice;
    while (1) {
        printf("\n1. Insert at beginning\n");
        printf("2. Display\n");
        printf("3. Print number of nodes\n");
        printf("4. Insert at end\n");
        printf("5. Insert at position\n");
        printf("6. Delete at the beginning\n");
        printf("7. Delete at the end\n");
        printf("8. Delete at position\n");
        printf("9. Search for data\n");
        printf("10. Print elements in reverse order\n");
        printf("0. Exit\n");
        printf("Enter the choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                insert_begin();
                break;
            case 2:
                display();
                break;
            case 3:
```

```
            printf("Number of nodes = %d\n", count);
            break;
        case 4:
            insert_end();
            break;
        case 5:
            insert_position();
            break;
        case 6:
            delete_begin();
            break;
        case 7:
            delete_end();
            break;
        case 8:
            delete_position();
            break;
        case 9:
            search();
            break;
        case 10:
            print_reverse();
            break;
        case 0:
            exit(0);
        default:
            printf("Invalid choice, please try again\n");
    }
```

```
    }
    return 0;
}
```

OUTPUT:

1. Insert at beginning

2. Display

3. Print number of nodes

4. Insert at end

5. Insert at position

6. Delete at the beginning

7. Delete at the end

8. Delete at position

9. Search for data

10. Print elements in reverse order

0. Exit

Enter the choice: 1

Enter data to be inserted: 5

1. Insert at beginning

2. Display

3. Print number of nodes

4. Insert at end

5. Insert at position

6. Delete at the beginning

7. Delete at the end

8. Delete at position

9. Search for data

10. Print elements in reverse order

0. Exit

Enter the choice: 2 5 <-> 6

1. Insert at beginning

2. Display

3. Print number of nodes

4. Insert at end

5. Insert at position

6. Delete at the beginning

7. Delete at the end

8. Delete at position

9. Search for data

10. Print elements in reverse order

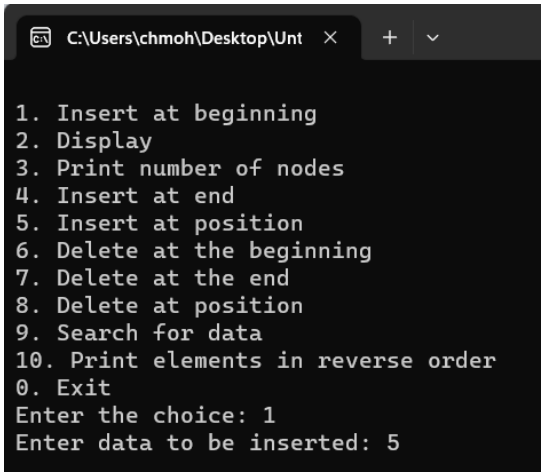
0. Exit

Enter the choice: 4

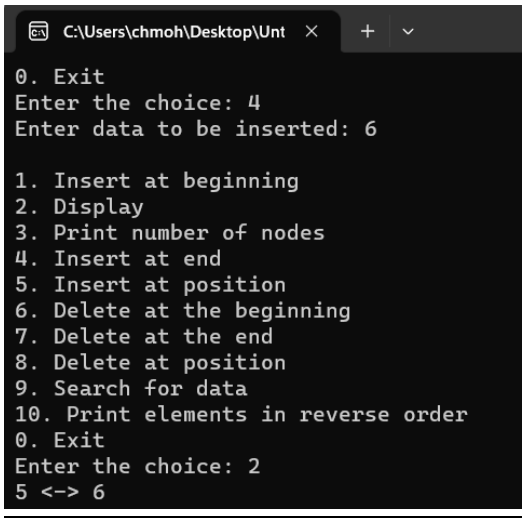
Enter data to be inserted: 6

OUTPUT:

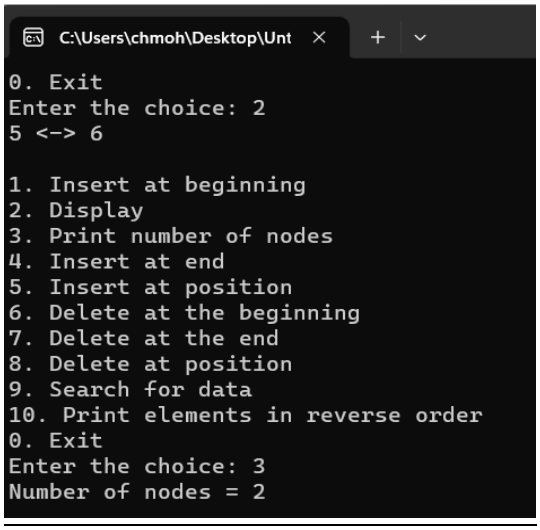
1. **Insert at beginning:**



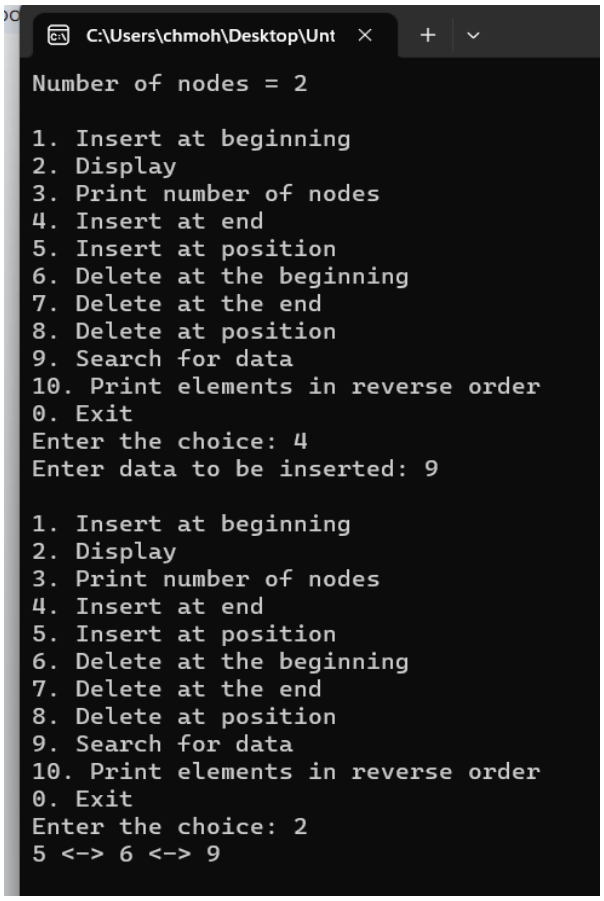
2.Display the list:



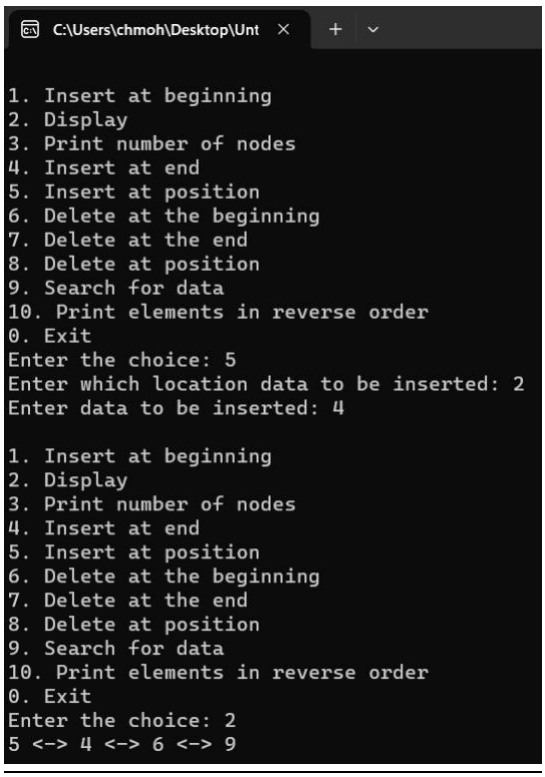
3.Print number of nodes:



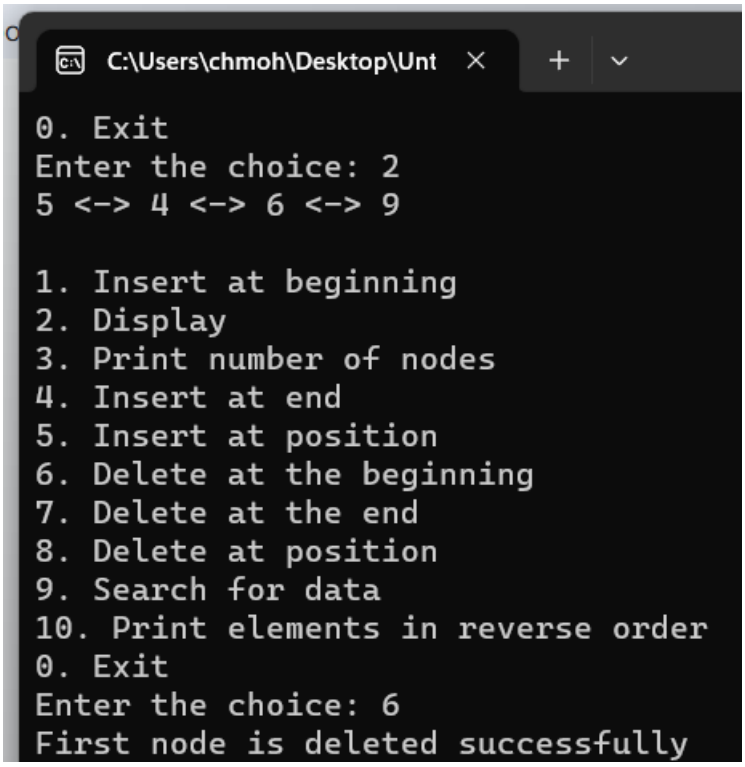
4.Insert at end:



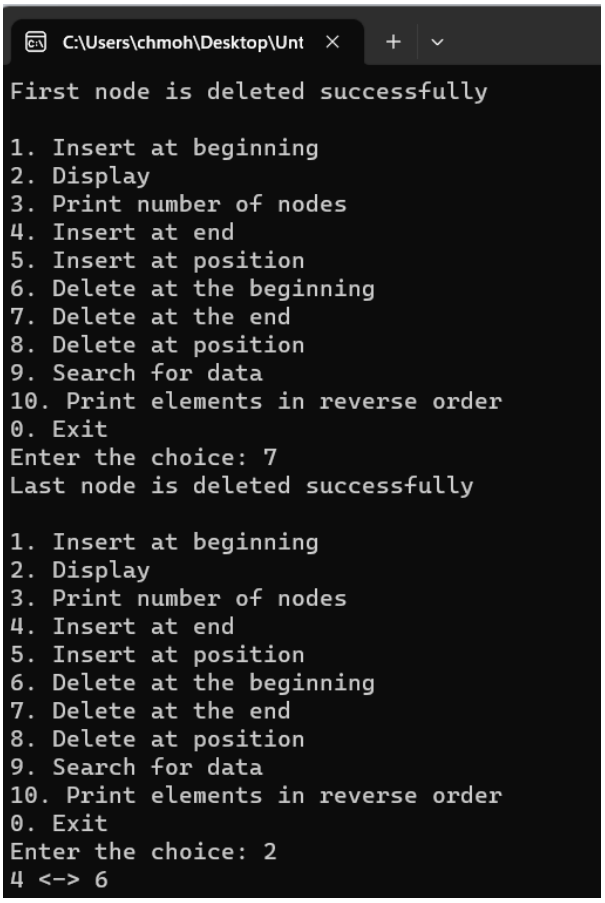
5.Insert at position:



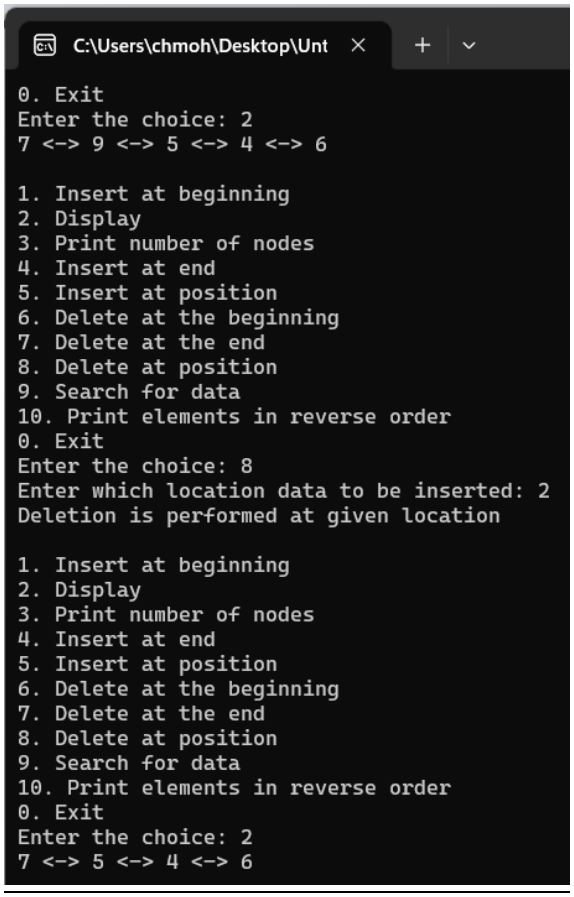
6.Delete at the beginning:



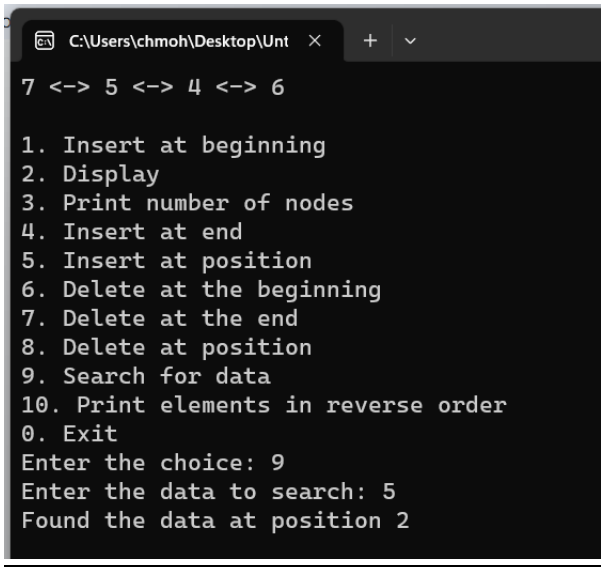
7.Delete at the end:



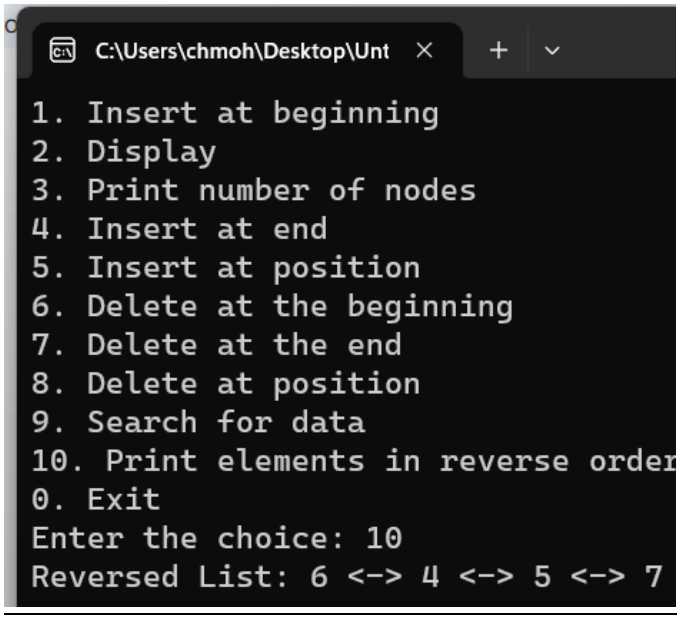
8.Delete at the position:



9.Search for data:



10.Print elements in reverse order:



PROGRAM ON CIRCULAR LINKED LIST:

PROGRAM TO PERFORM OPERATIONS ON CLL:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node* tail = NULL;
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
void insertAtBeginning(int data) {
    struct Node* newNode = createNode(data);
    if (tail == NULL) {
        newNode->next = newNode;
        tail = newNode;
    } else {
        newNode->next = tail->next;
        tail->next = newNode;
    }
}
void displayList() {
    if (tail == NULL) {
```

```
        printf("List is empty\n");
        return;
    }
    struct Node* current = tail->next;
    do {
        printf("%d ", current->data);
        current = current->next;
    } while (current != tail->next);
    printf("\n");
}
int countNodes() {
    if (tail == NULL)
        return 0;
    int count = 0;
    struct Node* current = tail->next;
    do {
        count++;
        current = current->next;
    } while (current != tail->next);
    return count;
}
void insertAtEnd(int data) {
    struct Node* newNode = createNode(data);
    if (tail == NULL) {
        tail = newNode;
        tail->next = newNode;
    } else {
        newNode->next = tail->next;
```

```
        tail->next = newNode;
        tail = newNode;
    }
}
void insertAtPosition(int data, int position) {
    if (position < 0 || (position > 0 && tail == NULL)) {
        printf("Invalid position\n");
        return;
    }

    if (position == 0) {
        insertAtBeginning(data);
        return;
    }
    struct Node* current = tail->next;
    int i;
    for (i = 0; i < position - 1; i++) {
        if (current == tail) {
            printf("Invalid position\n");
            return;
        }
        current = current->next;
    }
    struct Node* newNode = createNode(data);
    newNode->next = current->next;
    current->next = newNode;
}
void deleteAtBeginning() {
```

```
    if (tail == NULL) {
        printf("List is empty\n");
        return;
    }
    if (tail->next == tail) {
        free(tail);
        tail = NULL;
    } else {
        struct Node* temp = tail->next;
        tail->next = temp->next;
        free(temp);
    }
}
void deleteAtEnd() {
    if (tail == NULL) {
        printf("List is empty\n");
        return;
    }
    if (tail->next == tail) {
        free(tail);
        tail = NULL;
    } else {
        struct Node* current = tail->next;
        while (current->next != tail) {
            current = current->next;
        }
        current->next = tail->next;
        free(tail);
```

```
        tail = current;
    }
}
void deleteAtPosition(int position) {
    if (position < 0 || (position > 0 && tail == NULL)) {
        printf("Invalid position\n");
        return;
    }

    if (position == 0) {
        deleteAtBeginning();
        return;
    }
    struct Node* current = tail->next;
    int i;
    for (i = 0; i < position - 1; i++) {
        if (current->next == tail) {
            printf("Invalid position\n");
            return;
        }
        current = current->next;
    }

    struct Node* temp = current->next;
    current->next = temp->next;
    free(temp);
}
```

```
void search(int data) {
    if (tail == NULL) {
        printf("List is empty\n");
        return;
    }
    struct Node* current = tail->next;
    do {
        if (current->data == data) {
            printf("Data found\n");
            return;
        }
        current = current->next;
    } while (current != tail->next);
    printf("Data not found\n");
}
void deleteDuplicates() {
    if (tail == NULL || tail->next == tail) {
        return; // Nothing to do if list is empty or has only one element
    }
    struct Node* current = tail->next;
    do {
        struct Node* runner = current;
        do {
            if (runner->next->data == current->data) {
                struct Node* temp = runner->next;
                runner->next = temp->next;
                if (temp == tail) {
                    tail = runner;
```

```
            }
            free(temp);
        } else {
            runner = runner->next;
        }
    } while (runner->next != current);
    current = current->next;
  } while (current != tail);
}
int main() {
  int choice, data, position;
  do {
    printf("\n1. Insert at Beginning\n2. Display List\n3. Count Nodes\n");
    printf("4. Insert at End\n5. Insert at Position\n6. Delete at Beginning\n");
    printf("7. Delete at End\n8. Delete at Position\n9. Search for Data\n10. Delete
Duplicates\n0. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
      case 1:
        printf("Enter data to insert at the beginning: ");
        scanf("%d", &data);
        insertAtBeginning(data);
        break;
      case 2:
        printf("List: ");
        displayList();
```

```
            break;
        case 3:
            printf("Number of nodes: %d\n", countNodes());
            break;
        case 4:
            printf("Enter data to insert at the end: ");
            scanf("%d", &data);
            insertAtEnd(data);
            break;
        case 5:
            printf("Enter data and position to insert: ");
            scanf("%d %d", &data, &position);
            insertAtPosition(data, position);
            break;
        case 6:
            deleteAtBeginning();
            break;
        case 7:
            deleteAtEnd();
            break;
        case 8:
            printf("Enter position to delete: ");
            scanf("%d", &position);
            deleteAtPosition(position);
            break;
        case 9:
            printf("Enter data to search: ");
            scanf("%d", &data);
```

```
            search(data);
            break;
        case 10:
            deleteDuplicates();
            break;
        case 0:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice! Please try again.\n");
    }

} while (choice != 0);

return 0;
}
```

OUTPUT:

1. Insert at Beginning
2. Display List
3. Count Nodes
4. Insert at End
5. Insert at Position
6. Delete at Beginning
7. Delete at End
8. Delete at Position
9. Search for Data
10. Delete Duplicates

0. Exit

Enter your choice: 1

Enter data to insert at the beginning: 6

1. Insert at Beginning

2. Display List

3. Count Nodes

4. Insert at End

5. Insert at Position

6. Delete at Beginning

7. Delete at End

8. Delete at Position

9. Search for Data

10. Delete Duplicates

0. Exit

Enter your choice: 2

List: 6 5

1. Insert at Beginning

2. Display List

3. Count Nodes

4. Insert at End

5. Insert at Position

6. Delete at Beginning

7. Delete at End

8. Delete at Position

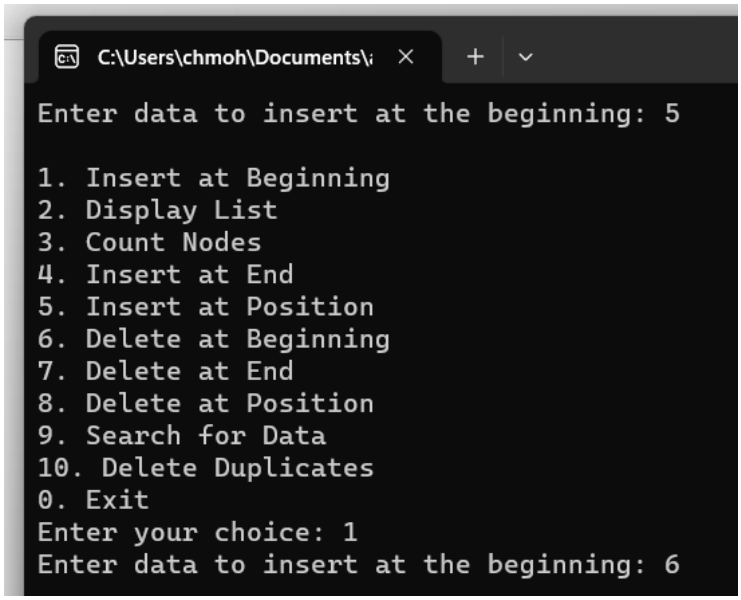
9. Search for Data

10. Delete Duplicates

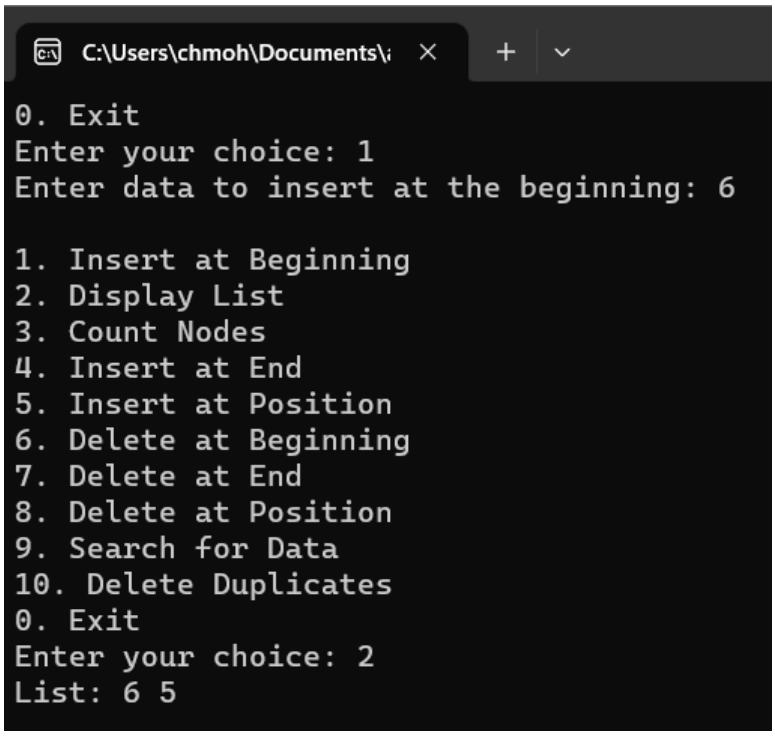
0. Exit

Enter your choice: 6

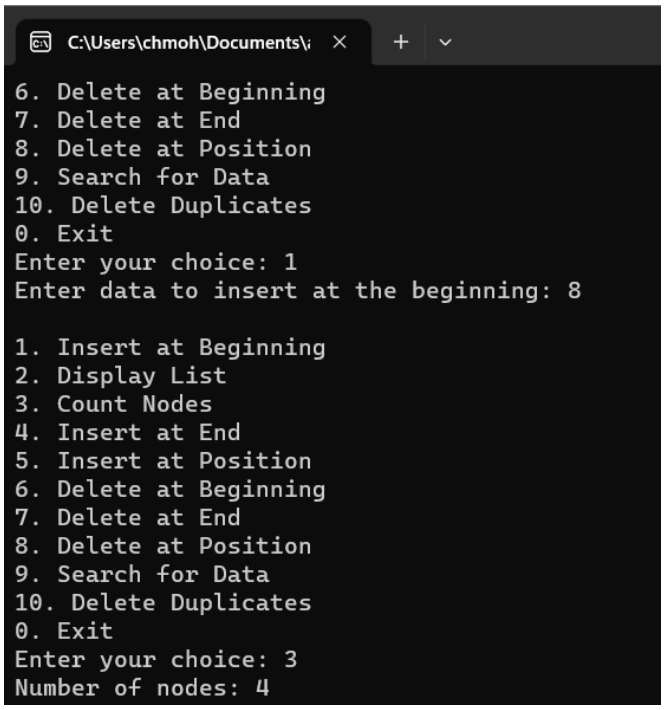
1. Insert at Beginning:



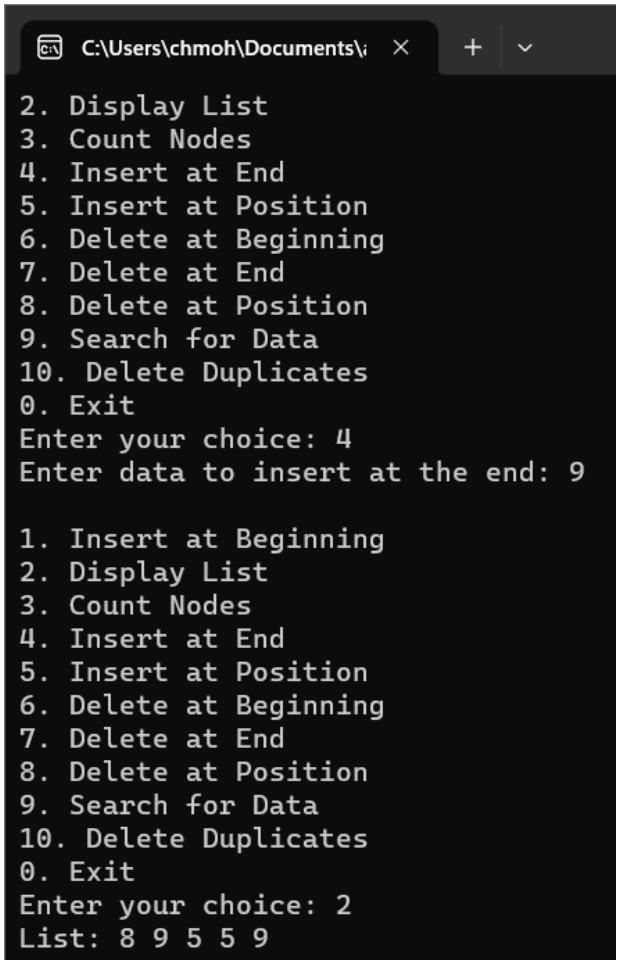
2. Display List



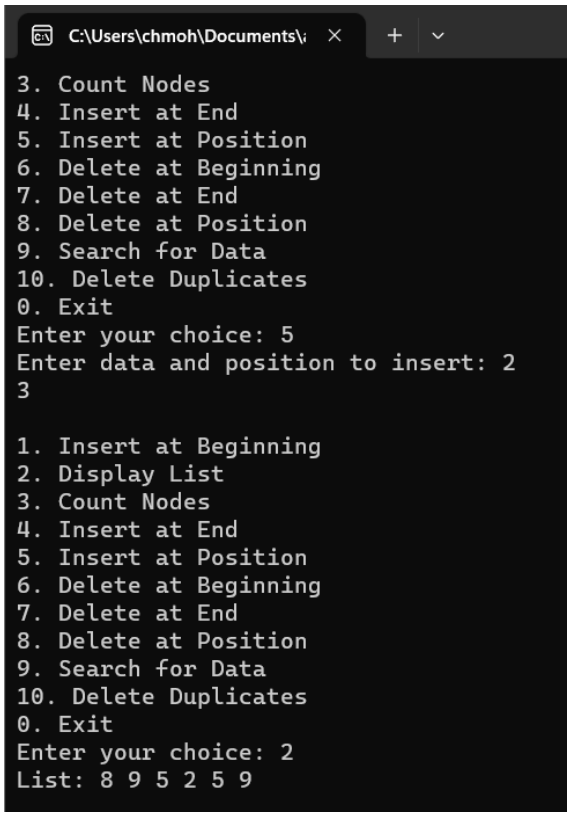
3. Count Nodes



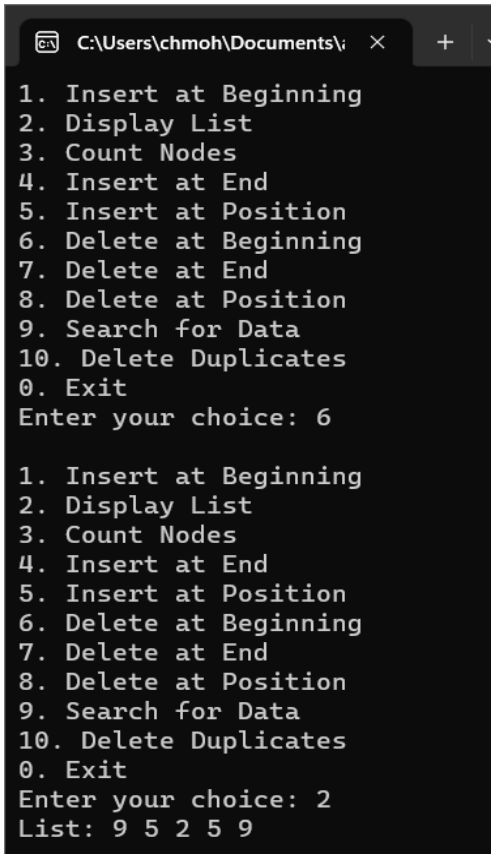
4. Insert at End



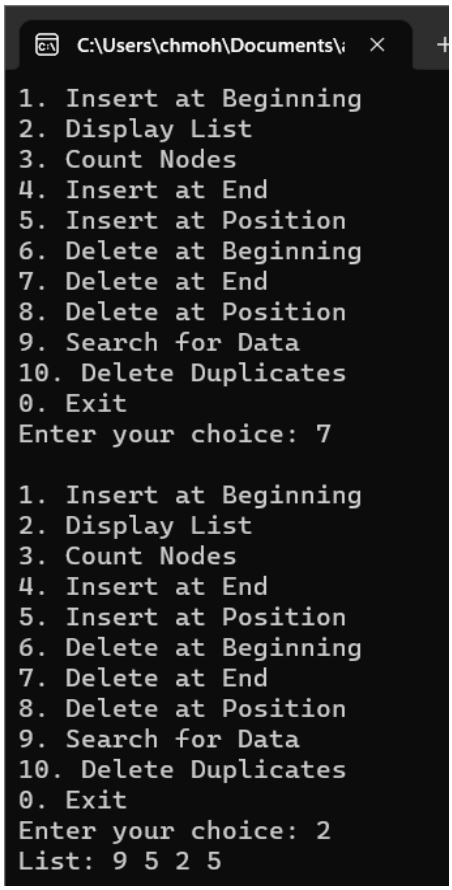
5. Insert at Position



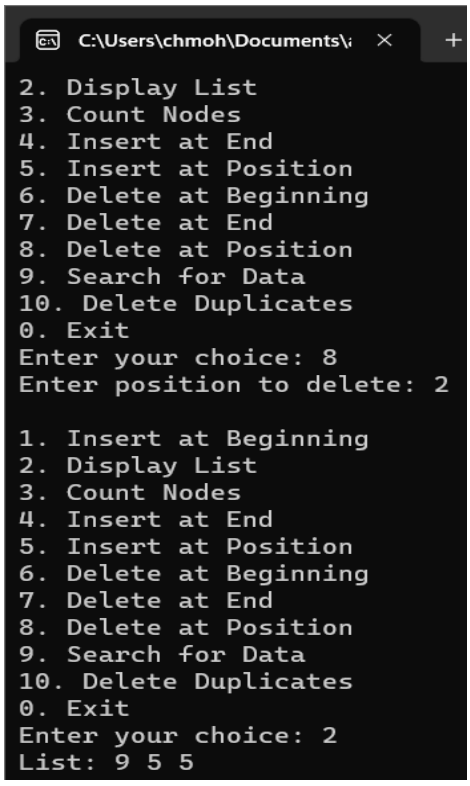
6. Delete at Beginning



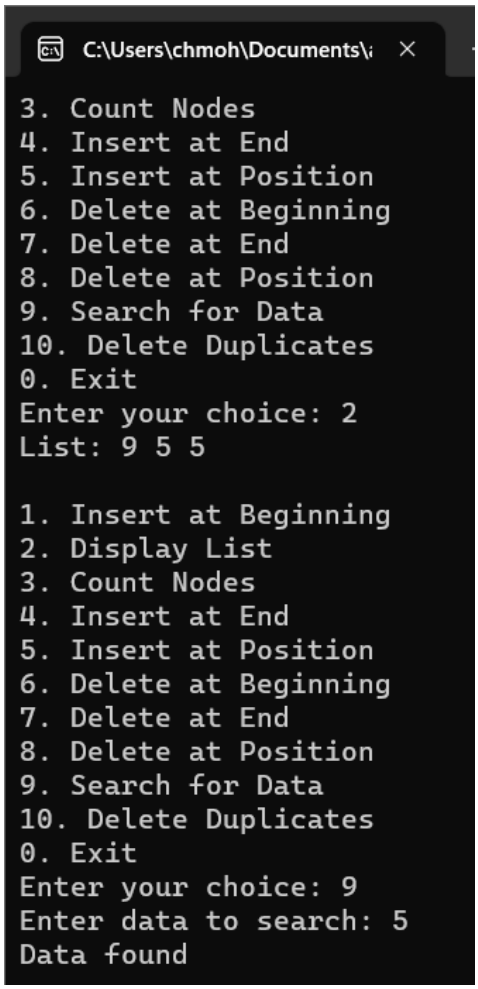
7. Delete at End



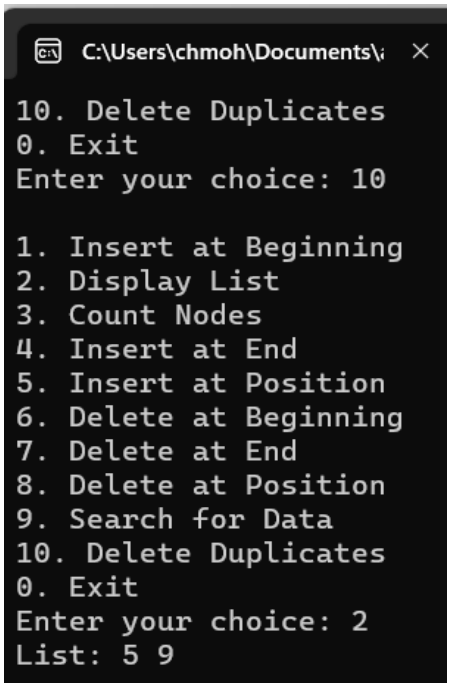
8. Delete at Position



9. Search for Data



10.Delete duplicates



Comparison of Linked Lists with Arrays and Dynamic Arrays:

Parameter	Linked List	Array	Dynamic array
Indexing	O(n)	O(1)	O(1)
Insertion/Deletion at beginning	O(1)	O(n) ,if array is not full	O(n)
Insertion at ending	O(n)	O(1) ,if array is not full	O(1) ,if array is not full O(n) ,if array is full
Deletion at ending	O(n)	O(1)	O(n)
Insertion at middle	O(n)	O(n) ,if array is not full	O(n)
Deletion in middle	O(n)	O(n) ,if array is not full	O(n)
Wasted space	O(n)(for pointers)	0	O(n)

Viva Questions:

1. Which node contains a NULL pointer in the above linked lists?

A: The last node in the linked list contains a NULL pointer, indicating the end of the list.

2. How do arrays differ from linked lists?

A: Arrays are static data structures with a fixed size that needs to be declared before use. On the other hand, linked lists are dynamic data structures that can grow or shrink during runtime. Arrays store elements in contiguous memory locations, allowing for direct access using indices, while linked lists store elements in nodes with each node containing data and a pointer to the next node, requiring traversal for access.

3. Compare the complexities for inserting/deleting elements at various positions in the above linked lists.

A:

- Singly Linked List (SLL):

Insertion at the Beginning:

Complexity: O(1)

Explanation: Since we only need to change the pointers of the new node to point to the current head node and update the head pointer, the operation can be done in constant time.

Insertion at the End:

Complexity: O(n)

Explanation: To insert at the end, we need to traverse the entire list to find the last node. This takes linear time.

Insertion at a Given Position:

Complexity: O(n)

Explanation: Similar to insertion at the end, we might need to traverse the list to find the desired position.

Deletion at the Beginning:

Complexity: O(1)

Explanation: Like insertion at the beginning, this operation involves changing pointers and updating the head pointer, which can be done in constant time.

Deletion at the End:

Complexity: O(n)

Explanation: Similar to insertion at the end, we need to traverse the list to find the second-to-last node. This takes linear time.

Deletion at a Given Position:

Complexity: O(n)

Explanation: Similar to insertion at a given position, we might need to traverse the list to find the desired position.

- Doubly Linked List (DLL):

Insertion at the Beginning:

Complexity: O(1)

Explanation: Similar to SLL, we only need to change the pointers of the new node and the current head node, and update the head pointer.

Insertion at the End:

Complexity: O(1)

Explanation: Since we have a tail pointer in a DLL, insertion at the end can be done in constant time.

Insertion at a Given Position:

Complexity: O(n)

Explanation: Even though we can traverse the list in either direction, we might still need to traverse to find the desired position.

Deletion at the Beginning:

Complexity: O(1)

Explanation: Similar to SLL, we only need to change pointers and update the head pointer.

Deletion at the End:

Complexity: O(1)

Explanation: Since we have a tail pointer, deletion at the end can be done in constant time.

Deletion at a Given Position:

Complexity: O(n)

Explanation: Similar to insertion at a given position, we might need to traverse the list to find the desired position.

- **Circular Linked List (CLL):**

Insertion:

Complexity: O(n)

Explanation: Depending on the specific position (beginning, end, or given), we might need to traverse the list to find the desired position.

Deletion:

Complexity: O(n)

Explanation: Similar to insertion, we might need to traverse the list to find the desired position.

In summary, the complexities for inserting and deleting elements at various positions in SLL, DLL, and CLL can vary, with some operations being more efficient in certain types of linked lists compared to others.

Program to perform operations on stack using arrays

```
#include <stdio.h>
#include <stdlib.h>
int size, top = -1;
int *a;
void push() {
   int data;
   printf("Enter the data to insert into the stack: ");
   scanf("%d", &data);
   if (top == size - 1)
      printf("Stack is full\n");
   else
      a[++top] = data;
}

void pop() {
   if (top == -1)
      printf("Stack is empty\n");
   else {
      printf("%d data is removed from stack", a[top]);
      top--;
   }
}

void peek() {
   if (top == -1)
```

```
        printf("Stack is empty\n");
    else {
        printf("%d is the top element in the stack", a[top]);
    }
}

void isEmpty() {
    if (top == -1)
        printf("Stack is empty\n");
    else
        printf("Stack is not empty\n");
}

void isFull() {
    if (top == size - 1)
        printf("Stack is full\n");
    else
        printf("Stack is not full\n");
}

void display() {
    int i;
    if (top == -1)
        printf("Stack is empty\n");
    else {
        printf("Elements in the stack:\n");
```

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

void count() {
    printf("Number of elements in the stack = %d", top + 1);
}

int main() {
    int choice;
    printf("Enter the size of the stack: ");
    scanf("%d", &size);
    a = (int *)malloc(sizeof(int) * size);

    while (1) {
        printf("\n1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. IsEmpty\n");
        printf("5. IsFull\n");
        printf("6. Display\n");
        printf("7. Count\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
        switch (choice) {
            case 1: push(); break;
            case 2: pop(); break;
            case 3: peek(); break;
            case 4: isEmpty(); break;
            case 5: isFull(); break;
            case 6: display(); break;
            case 7: count(); break;
            default: exit(0);
        }
    }
    return 0;
}
```

OUTPUT:

enter the size of stack5

1. push

2. pop

3. peek

4. isEmpty

5. isFull

6. Display

7. count

enter your choice1

enter the data to insert into the stack6

1. push

2. pop

3. peek

4. isEmpty

5. isFull

6. Display

7. count

enter your choice1

enter the data to insert into the stack52

1. push

2. pop

3. peek

4. isEmpty

5. isFull

6. Display

7. count

enter your choice6

6

52

1. push

2. pop

3. peek

4. isEmpty

5. isFull

6. Display

7. count

enter your choice2

52 data is removed from stack

1. push

2. pop

3. peek

4. isEmpty

5. isFull

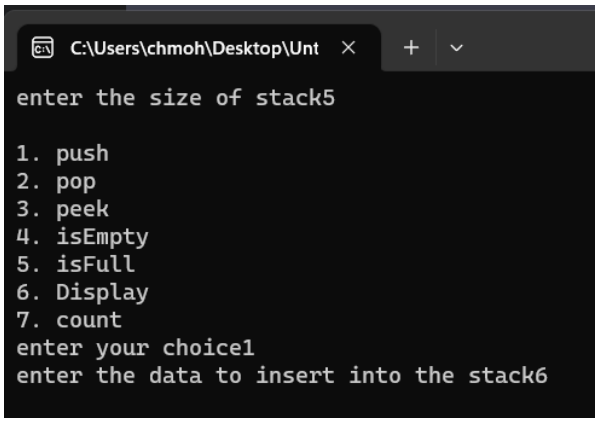
6. Display

7. count

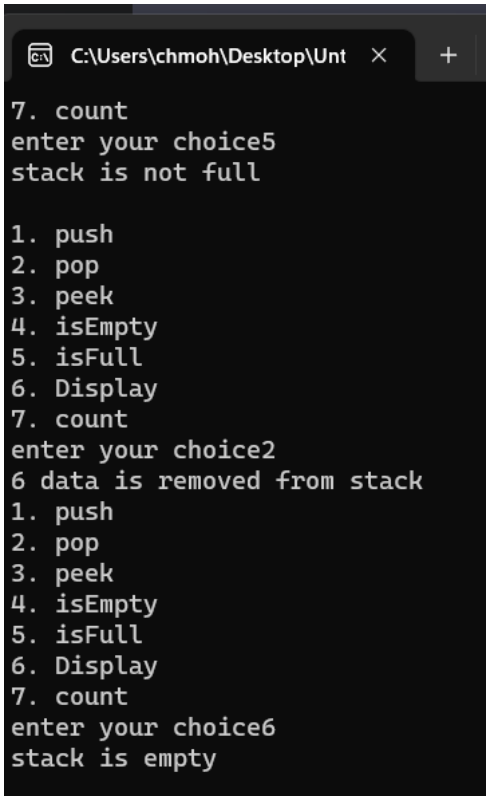
enter your choice5

stack is not full

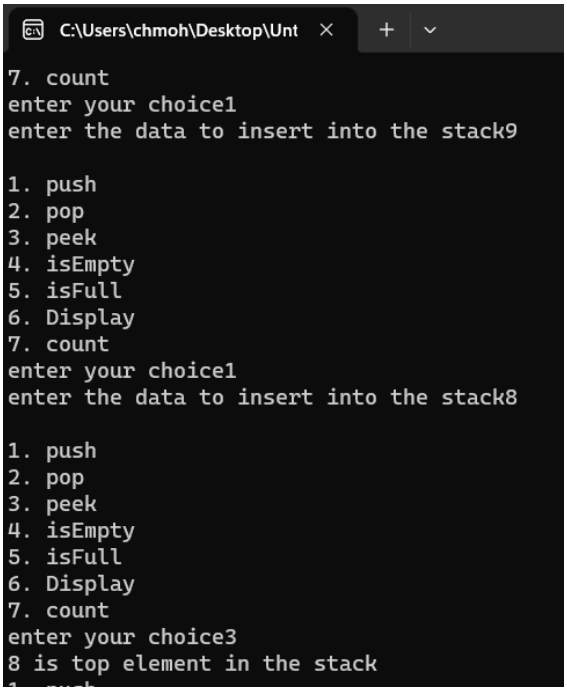
1. push:



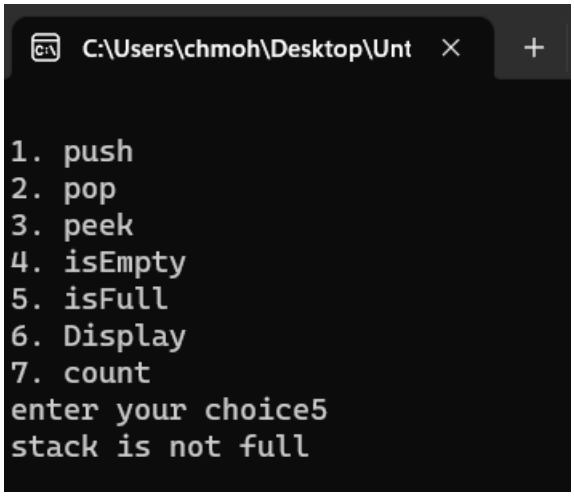
2. pop:



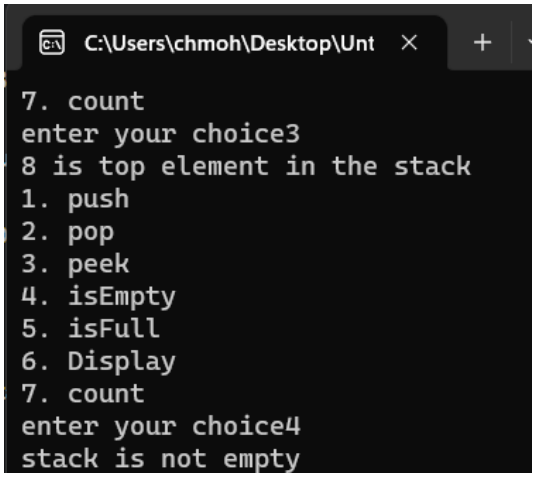
3.peek:



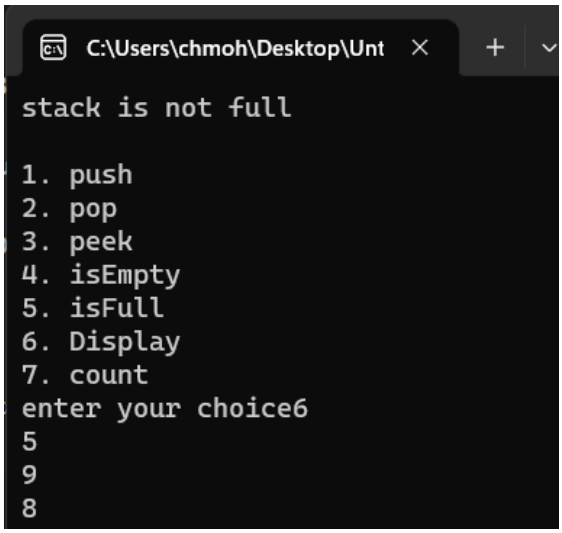
4.isEmpty



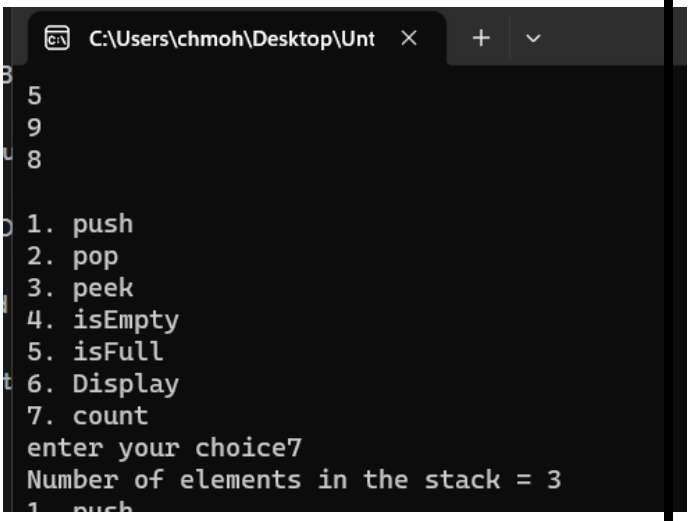
5.isFull:



6.Display:



7.count:



Program to perform operations on stack using linked lists:

```
#include <stdio.h>
#include <stdlib.h>
struct node {
    int data;
    struct node *next;
};
struct node *top;
int count;
void push() {
    int value;
    struct node *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    printf("Enter the data to insert in the stack: ");
    scanf("%d", &value);
    temp->data = value;
    if (top == NULL)
        temp->next = NULL;
    else
        temp->next = top;

    top = temp;
    count++;
}
void pop() {
    if (top == NULL)
```

```
        printf("Stack is empty\n");
    else if (top->next != NULL) {
        struct node *p;
        p = top;
        printf("%d is removed from stack\n", top->data);
        top = top->next;
        free(p);
        count--;
    } else {
        struct node *p;
        p = top;
        printf("%d is removed from stack\n", top->data);
        top = NULL;
        free(p);
        count--;
    }
}
void peek() {
    if (top == NULL)
        printf("Stack is empty\n");
    else
        printf("%d is the top element in the stack", top->data);
}
void isEmpty() {
    if (top == NULL)
        printf("Stack is empty");
```

```
    else
        printf("Stack is not empty");
}
void display() {
    if (top == NULL)
        printf("Stack is empty");
    else {
        struct node *p;
        p = top;
        while (p->next != NULL) {
            printf("%d->", p->data);
            p = p->next;
        }
        printf("%d", p->data);
    }
}
void total() {
    if (top == NULL)
        printf("Stack is empty");
    else
        printf("Count = %d", count);
}
int main() {
    int choice;
    while (1) {
        printf("\n1. Push\n");
```

```
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. IsEmpty\n");
        printf("5. Display\n");
        printf("6. Count\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: push(); break;
            case 2: pop(); break;
            case 3: peek(); break;
            case 4: isEmpty(); break;
            case 5: display(); break;
            case 6: total(); break;
            default: exit(0);
        }
    }
    return 0;
}
```

OUTPUT:

1. Push

2. Pop

3. Peek

4. IsEmpty

5. Display

6. Count

Enter your choice: 1

Enter the data to insert in the stack: 9

1. Push

2. Pop

3. Peek

4. IsEmpty

5. Display

6. Count

Enter your choice: 5

9->7->5

1. Push

2. Pop

3. Peek

4. IsEmpty

5. Display

6. Count

Enter your choice: 2

9 is removed from stack

1. Push

2. Pop

3. Peek

4. IsEmpty

5. Display

6. Count

Enter your choice: 5

7->5

1. Push

2. Pop

3. Peek

4. IsEmpty

5. Display

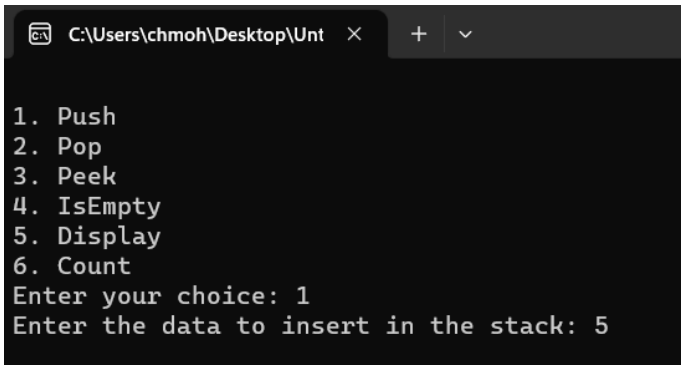
6. Count

Enter your choice: 6

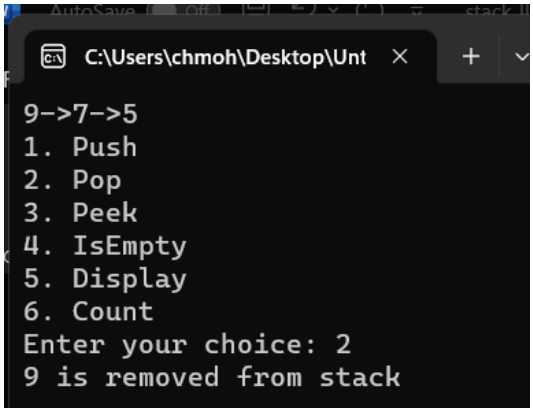
Count = 2

22501A0504

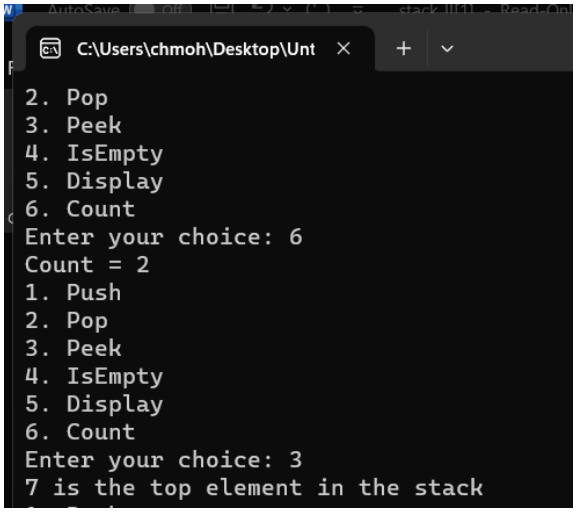
1.push:



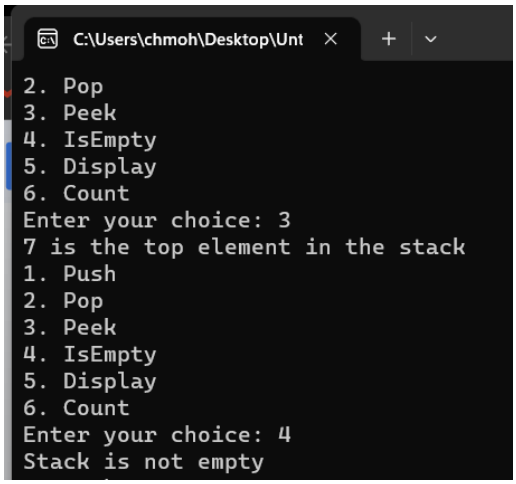
2.pop:



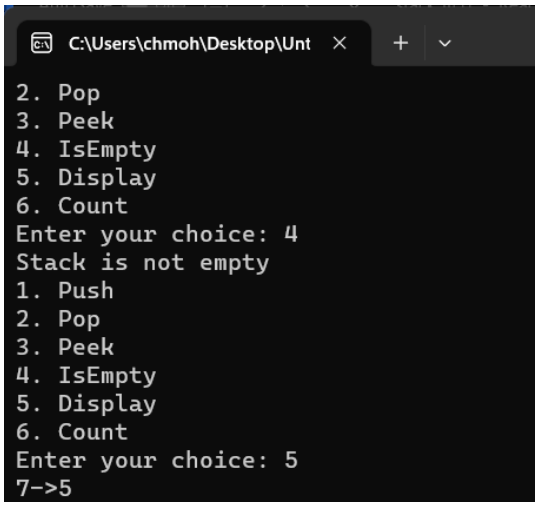
3.peek:



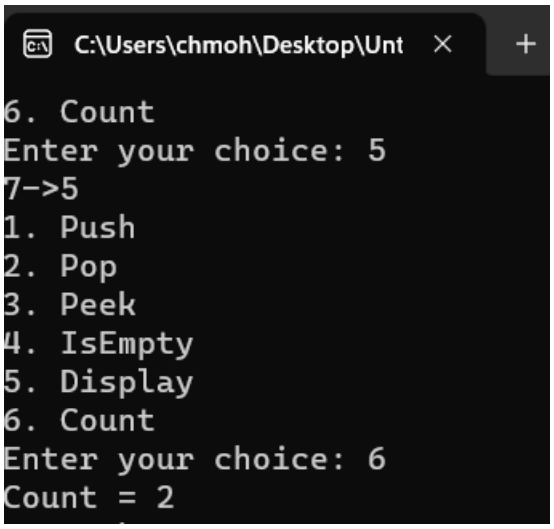
4.IsEmpty:



5.Display:



6.count:



VIVA QUESTIONS:

1. **Define Stack**

A. A stack is a fundamental data structure in computer science that follows the Last-In, First-Out (LIFO) principle. It is an ordered collection of elements where the addition of new elements (push operation) and the removal of existing elements (pop operation) occur from the same end, traditionally referred to as the top. Stacks have limited access, allowing operations only on the topmost element.

2. Explain why stack is a recursive data structure

A. A stack is considered a recursive data structure because it exhibits recursive behavior in its operations. When an element is pushed onto the stack, it forms a relationship with the previous element, creating a chain of references. The push and pop operations operate on this chain in a recursive manner. The push operation involves creating a new element that references the current top element, and the pop operation involves removing the current top element, revealing the previously referenced element. This recursive behavior mirrors the concept of function calls and returns in recursion.

3.List the applications of stacks

A.

1. Function Calls: Stacks are used by programming languages to keep track of function calls and returns.
2. Expression Evaluation: They are used in compilers to evaluate arithmetic expressions.
3. Backtracking: In algorithms, stacks are used to facilitate backtracking, like in depth-first search (DFS) algorithms.
4. Browser History: Web browsers use a stack to keep track of visited pages.
5. Undo Mechanism: Stacks can be used to implement the undo functionality in software applications.

4. How is a stack different from an array?

A: While both stacks and arrays can store elements, the key difference lies in how elements are added and removed. In a stack, elements are added and removed from the same end, following the LIFO principle. In an array, elements can be added and removed from any position, and the order of insertion does not determine the order of retrieval.

5. Give an example of a real-world scenario where stacks are used.

A: A common real-world example is the use of a web browser's back button. When a user visits multiple web pages, the URLs are pushed onto a stack. Clicking the back button pops the last visited URL, allowing the user to navigate back through the history.

PROBLEM:

You are given a stack of N integers such that the first element represents the top of the stack and the last element represents the bottom of the stack. You need to pop at least one element from the stack. At any one moment, you can convert stack into a queue. The bottom of the stack represents the front of the queue. You cannot convert the queue back into a stack. Your task is to remove exactly K elements such that the sum of the K removed elements is maximised in c

PROGRAM:

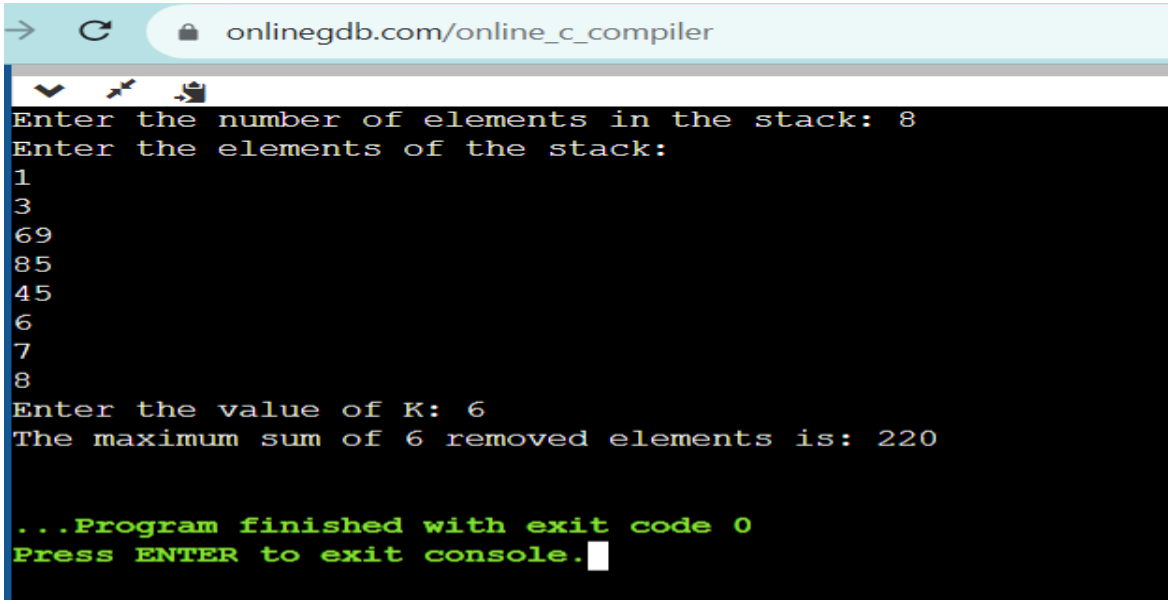
```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
struct Queue {
   int front, rear, size;
   unsigned capacity;
   int* array;
};
struct Queue* createQueue(unsigned capacity) {
   struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
   queue->capacity = capacity;
   queue->front = 0;
   queue->size = 0;
   queue->rear = capacity - 1;
   queue->array = (int*)malloc(queue->capacity * sizeof(int));
   return queue;
}
int isFull(struct Queue* queue) {
```

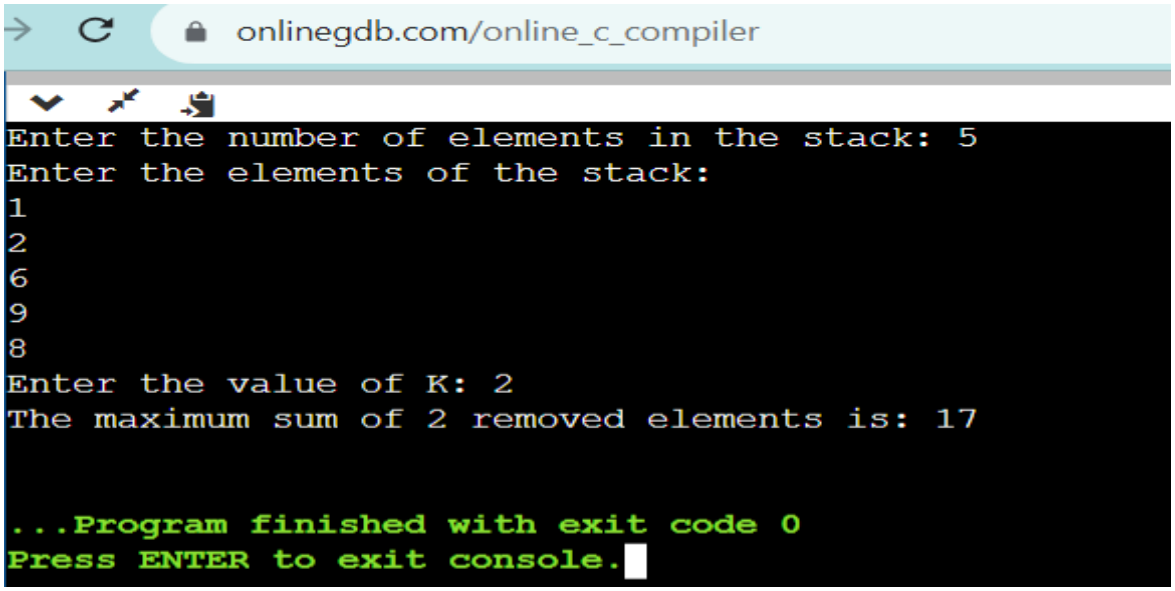
```
    return (queue->size == queue->capacity);
}
int isEmpty(struct Queue* queue) {
    return (queue->size == 0);
}
void enqueue(struct Queue* queue, int item) {
    if (isFull(queue))
        return;
    queue->rear = (queue->rear + 1) % queue->capacity;
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
}
int dequeue(struct Queue* queue) {
    if (isEmpty(queue))
        return -1;
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1) % queue->capacity;
    queue->size = queue->size - 1;
    return item;
}
int max_sum_of_removed_elements(int stack[], int N, int K) {
    if (K >= N) {
        int sum = 0;
        for (int i = 0; i < N; i++) {
            sum += stack[i];
        }
```

```
        return sum;
    }

    struct Queue* queue = createQueue(N);
 for (int i = 0; i < K; i++) {
        enqueue(queue, stack[N - 1 - i]);
    while (queue->size > K) {
 dequeue(queue);
    }
 int sum = 0;
    for (int i = 0; i < queue->size; i++) {
        sum += queue->array[i];
    }
return sum;
}
int main() {
    int stack[MAX];
    int N, K;
    printf("Enter the number of elements in the stack: ");
    scanf("%d", &N);
 printf("Enter the elements of the stack:\n");
    for (int i = 0; i < N; i++) {
        scanf("%d", &stack[i]);
    }
 printf("Enter the value of K: ");
    scanf("%d", &K);
```

```
 int max_sum = max_sum_of_removed_elements(stack, N, K);
    printf("The maximum sum of %d removed elements is: %d\n", K, max_sum);
}
```

OUTPUT:





PROBLEM:

Humpy likes to jump from one building to another. But he only jumps to next higher building and stops when no higher building is available. Stamina required for a journey is **xor** of all the heights on which humpy jumps until he stops.

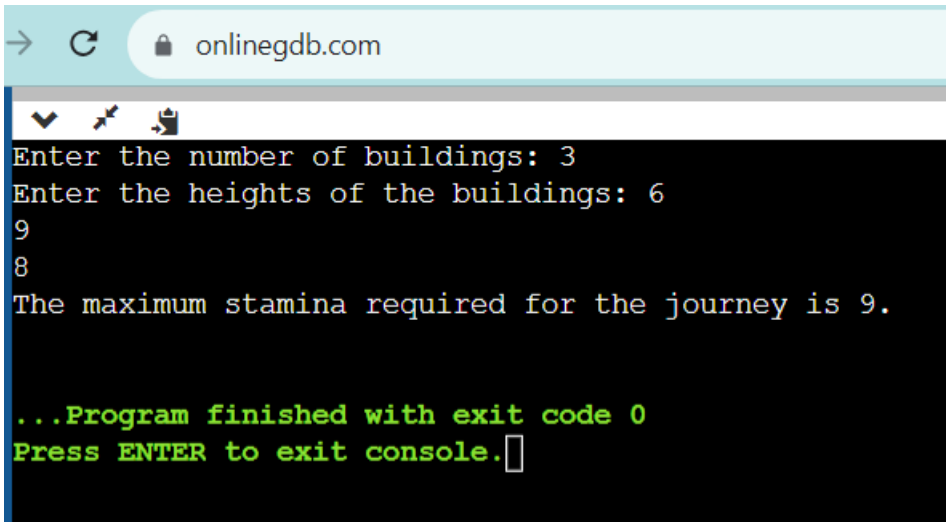
If heights are [1 2 4], and he starts from 1, goes to 2 stamina required is 1⊕2=3, then from 2 to 3. Stamina for the entire journey is 1⊕2⊕4=7. Find the maximum stamina required if can start his journey from any building.

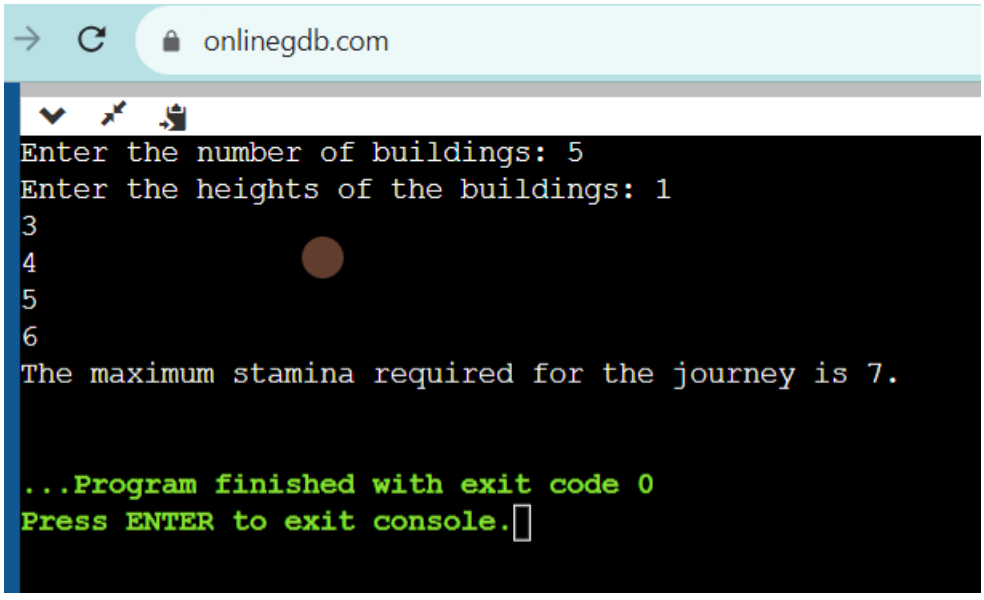
PROGRAM:

```
#include <stdio.h>

int maxStamina(int heights[], int n) {
    int max_stamina = 0;
    for (int i = 0; i < n; i++) {
        int curr_stamina = heights[i];
        int j = i + 1;
        while (j < n && heights[j] > heights[i]) {
            curr_stamina ^= heights[j];
            j++;
        }
        if (curr_stamina > max_stamina) {
            max_stamina = curr_stamina;
        }
    }
    return max_stamina;
}

int main() {
    int n;
    printf("Enter the number of buildings: ");
    scanf("%d", &n);
```

```
    int heights[n];
    printf("Enter the heights of the buildings: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &heights[i]);
    }
    int result = maxStamina(heights, n);
    printf("The maximum stamina required for the journey is %d.\n", result);
    return 0;
}
```

Output:





EXP NO: **APPLICATIONS OF QUEUES AND STACKS**

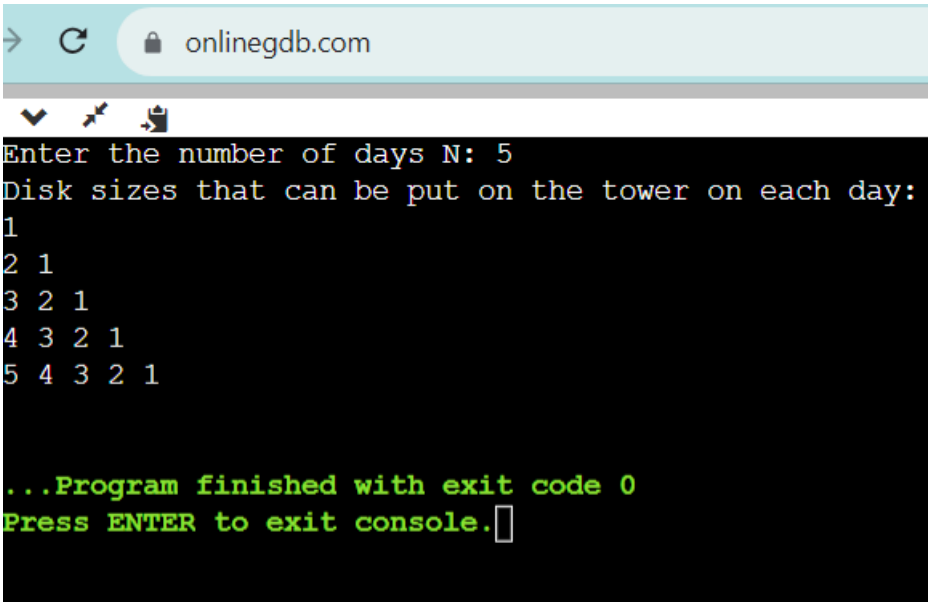
PROBLEM:

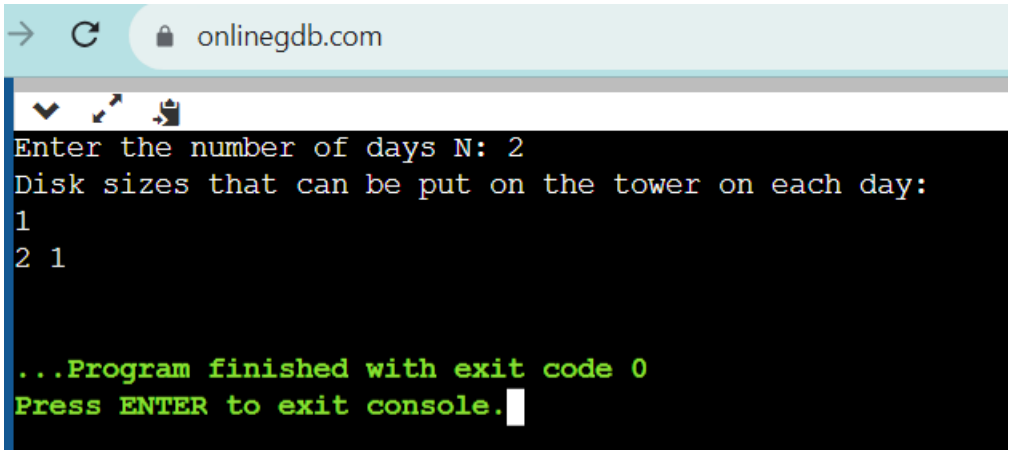
Your task is to construct a tower in N days by following these conditions: Every day you are provided with one disk of distinct size. The disk with larger sizes should be placed at the bottom of the tower. The disk with smaller sizes should be placed at the top of the tower. The order in which tower must be constructed is as follows: You cannot put a new disk on the top of the tower until all the larger disks that are given to you get placed. Print N lines denoting the disk sizes that can be put on the tower on the i^th day**.**

PROGRAM:

```
#include <stdio.h>
int main() {
   int N;
   printf("Enter the number of days N: ");
   scanf("%d", &N);
   int tower[N];
   int currentTop = 0;
   printf("Disk sizes that can be put on the tower on each day:\n");
   for (int i = 1; i <= N; i++) {
      tower[currentTop] = i;
      for (int j = currentTop; j >= 0; j--) {
         printf("%d ", tower[j]);
      }
      printf("\n");
      currentTop++;
   }
   return 0;
}
```

OUTPUT:


onlinegdb.com
Enter the number of days N: 5
Disk sizes that can be put on the tower on each day:
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
...Program finished with exit code 0
Press ENTER to exit console.


onlinegdb.com
Enter the number of days N: 2
Disk sizes that can be put on the tower on each day:
1
2 1
...Program finished with exit code 0
Press ENTER to exit console.

PROBLEM:

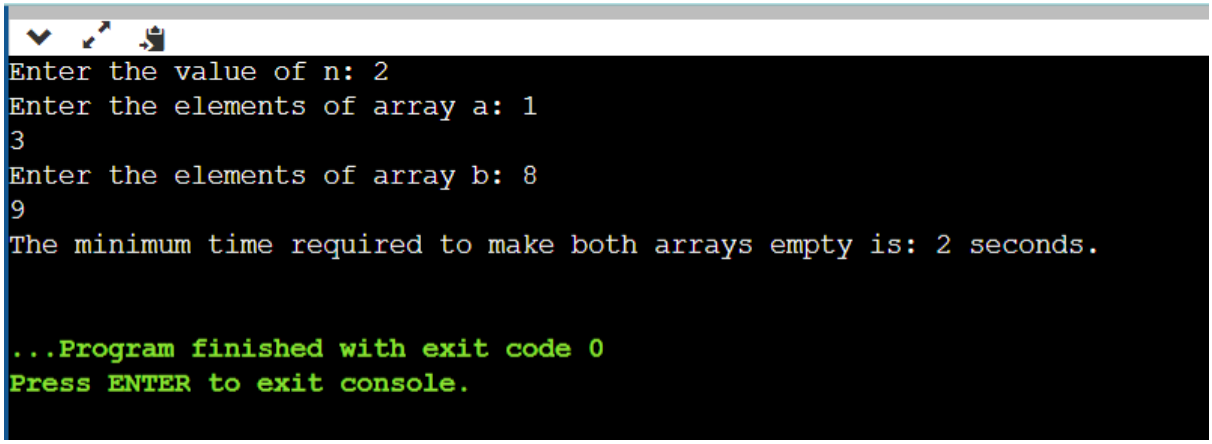
You are given two arrays each of size n , a and b consisting of the first n positive integers each exactly once, that is, they are permutations. Your task is to find the minimum time required to make both the arrays empty. The following two types of operations can be performed any number of times each taking 1 second: In the first operation, you are allowed to rotate the first array clockwise. In the second operation, when the first element of both the arrays is the same, they are removed from both the arrays and the process continues.

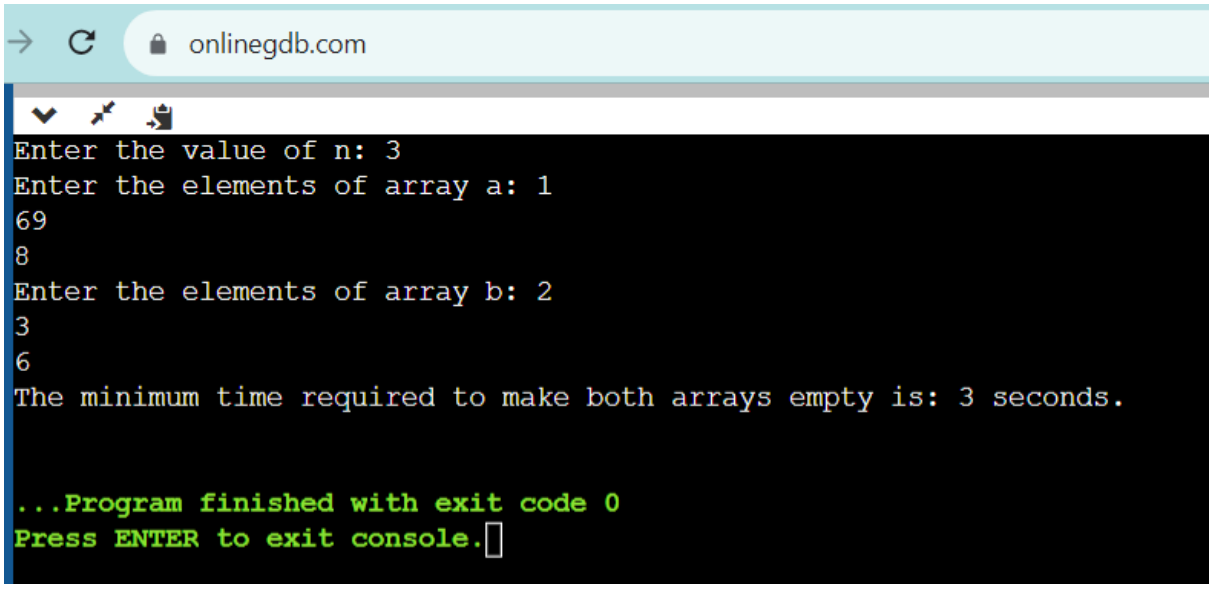
Program:

```
#include <stdio.h>

int minTimeToEmptyArrays(int a[], int b[], int n) {
   int i = 0, j = 0, rotations = 0;
   while (i < n && j < n) {
      if (a[i] == b[j]) {
         i++;
         j++;
      } else {
         i++;
         rotations++;
      }
   }
   while (i < n) {
      i++;
      rotations++;
   }
   return rotations;
}
```

```
int main() {
   int n;
   printf("Enter the value of n: ");
   scanf("%d", &n);

   int a[n], b[n];
   printf("Enter the elements of array a: ");
   for (int i = 0; i < n; i++) {
      scanf("%d", &a[i]);
   }
 printf("Enter the elements of array b: ");
   for (int i = 0; i < n; i++) {
      scanf("%d", &b[i]);
   }
int result = minTimeToEmptyArrays(a, b, n);
   printf("The minimum time required to make both arrays empty is: %d
seconds.\n", result);
   return 0;
}
```

OUTPUT:


Enter the value of n: 2
Enter the elements of array a: 1
3
Enter the elements of array b: 8
9
The minimum time required to make both arrays empty is: 2 seconds.
...Program finished with exit code 0
Press ENTER to exit console.


onlinegdb.com
Enter the value of n: 3
Enter the elements of array a: 1
69
8
Enter the elements of array b: 2
3
6
The minimum time required to make both arrays empty is: 3 seconds.
...Program finished with exit code 0
Press ENTER to exit console.

EXPERIMENT NO: 9 BINARY TREES

AIM: Implement Binary trees using Arrays and LinkedLists

DEFINITION:

A binary tree is a hierarchical data structure composed of nodes, where each node has at most two children, referred to as the left child and the right child. The topmost node in a binary tree is called the root.

The binary tree structure is defined by the following characteristics:

1. **Root**: The top node in the binary tree is known as the root. It serves as the starting point for traversing the tree.
2. **Node:** Each element or entity within the binary tree is called a node. Nodes contain data or values and have zero, one, or two child nodes.
3. **Child Nodes**: A node in a binary tree can have zero, one, or two child nodes. These child nodes are the nodes directly connected to the parent node.
4. **Leaf Nodes**: Leaf nodes are nodes in the binary tree that do not have any child nodes. They are the endpoints of the tree branches.
5. **Parent Node**: A node in a binary tree that has one or more child nodes is called a parent node. It is the node from which other nodes originate.
6. **Depth**: The depth of a node in a binary tree is the length of the path from the root to that node.

Applications of Binary Trees:

1. Search algorithms
2. Database systems
3. File systems
4. Compression algorithms
5. Decision trees
6. Game AI
7. Sorting algorithms

Advantages of Binary Tree:

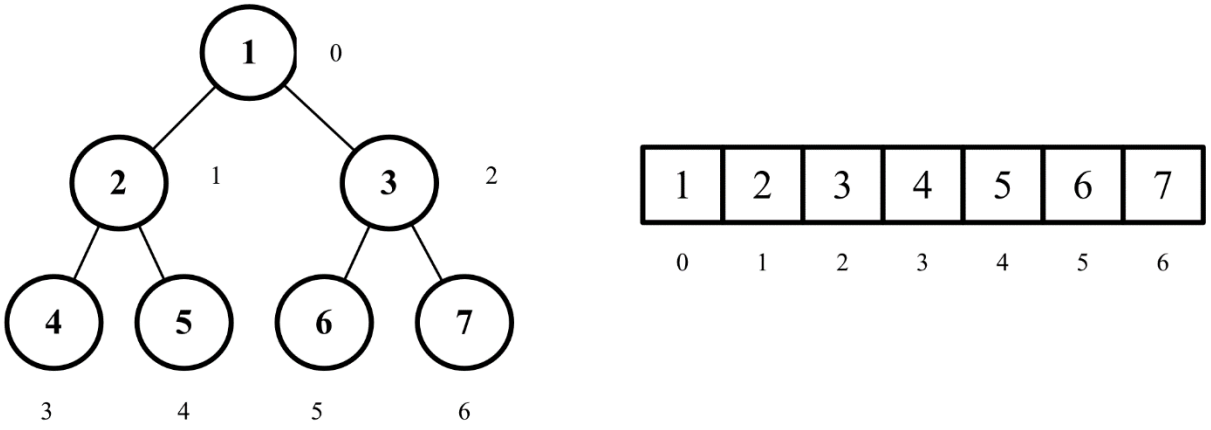
1. Efficient searching
2. Ordered traversal

3. Memory efficient
4. Fast insertion and deletion
5. Easy to implement
6. Useful for sorting

Disadvantages of Binary Tree:

1. Limited structure
2. Unbalanced trees
3. Space inefficiency
4. Slow performance in worst-case scenarios
5. Complex balancing algorithms

OPERATIONS ON BINARY TREES USING ARRAYS:



Initialize the Binary Tree Array:

Create an array to store the elements of the binary tree.

You can use a 1-dimensional array and represent each node at index i with its left child at index 2*i+1 and right child at index 2*i+2.

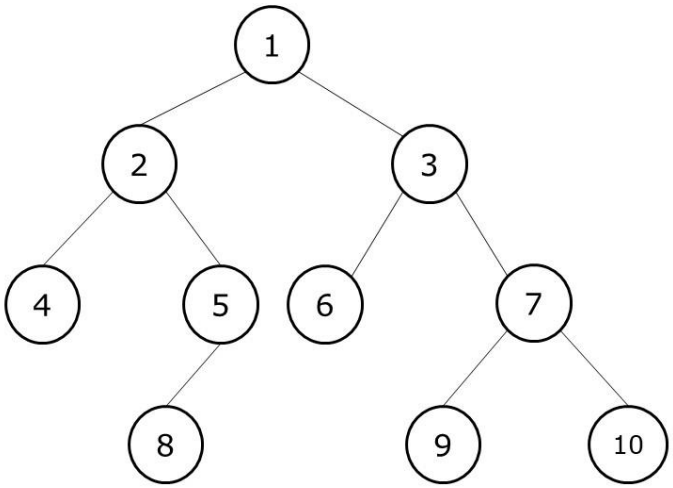
Insert Elements:

To insert elements into the binary tree, you'll need to traverse the array in a sequential manner and assign values to each element.

Operations:

INORDER TRAVERSAL:

1. Traverse the left subtree in inorder.
2. Visit the current node.
3. Traverse the right subtree in inorder.

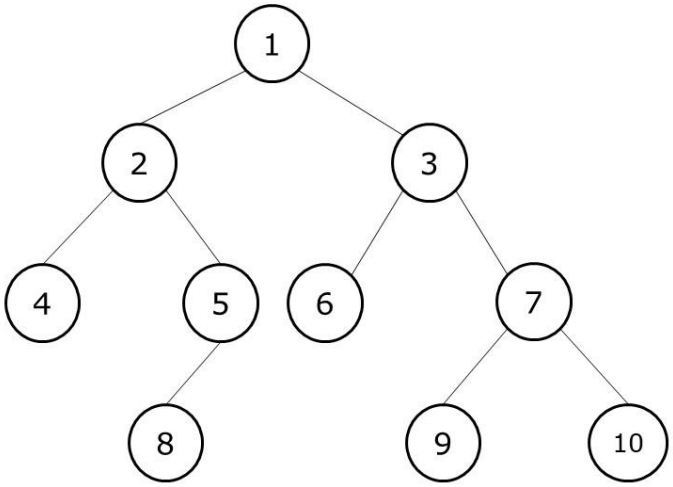


Inorder Traversal:
[left,root,right]

4	2	8	5	1	6	3	9	7	10

PREORDER TRAVERSAL:

1. Start from the root.
2. Visit the current node.
3. Traverse the left subtree in preorder.
4. Traverse the right subtree in preorder.

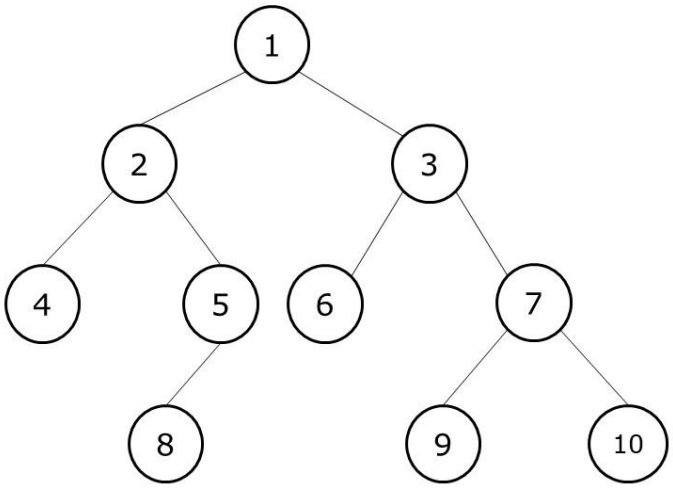


Preorder Traversal:
[root, left, right]

1	2	4	5	8	3	6	7	9	10

POSTORDER TRAVERSAL:

1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the current node.

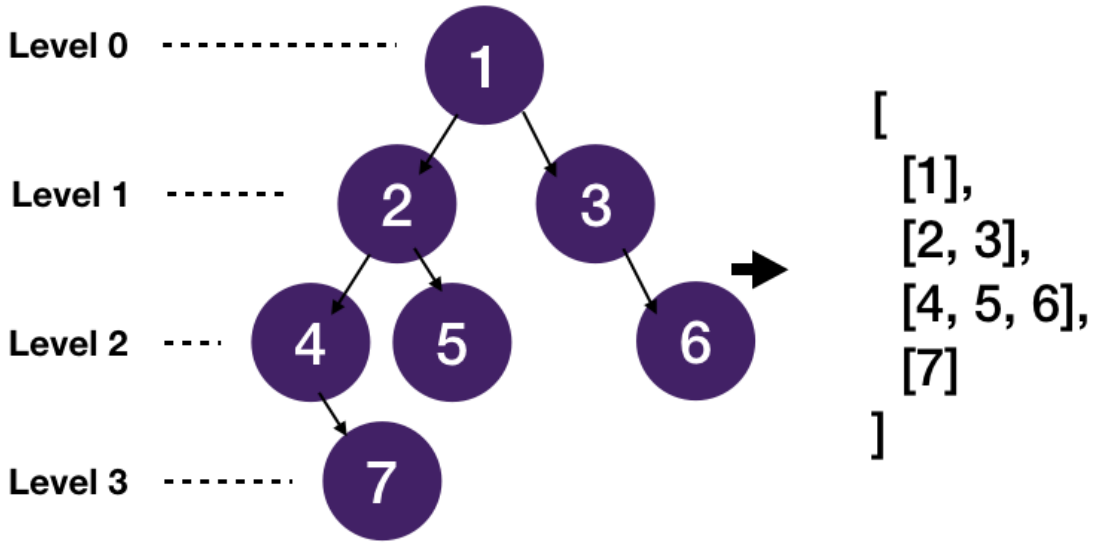


Postorder Traversal:
[left, right, root]

4	8	5	2	6	9	10	7	3	1

LEVELORDER TRAVERSAL:

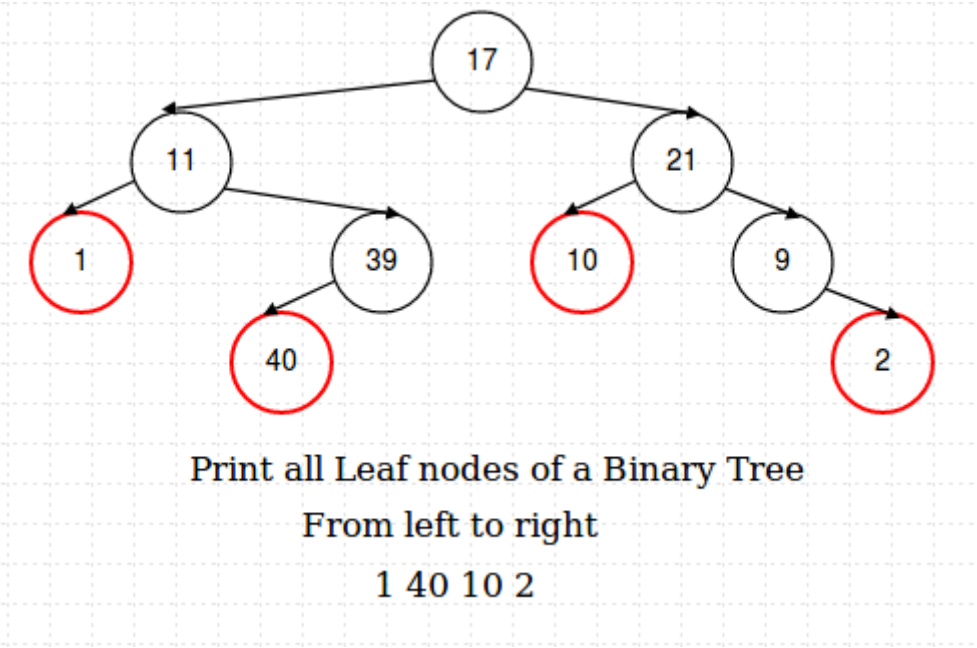
1. Use a queue data structure.
2. Enqueue the root node.
3. While the queue is not empty, dequeue a node, visit it, and enqueue its children.



PRINT LEAF NODES:

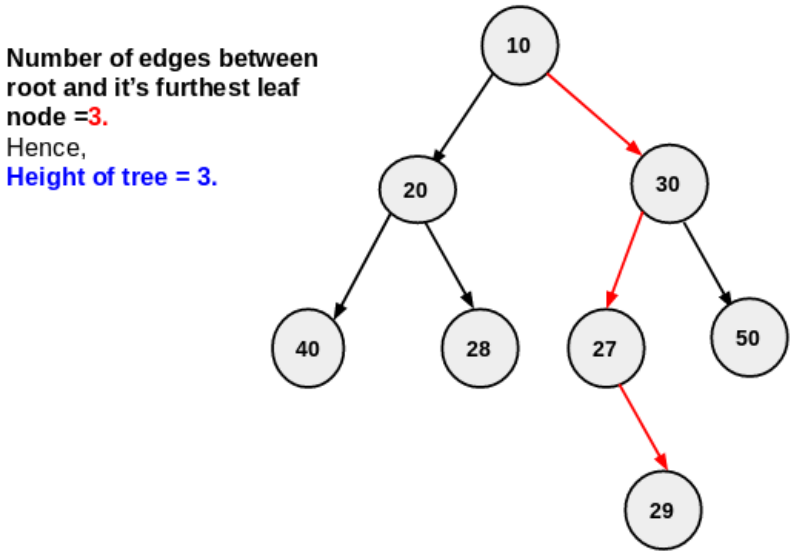
1. Certainly! Here are the steps described in words.
2. Define a variable min that will initially hold the data of the root node.
3. Traverse through the left sub-tree.
4. Compare the smallest node on the left with min.
5. Traverse through the right subtree.
6. In the end, min will hold the smallest node.
7. If you want to print the elements of the tree in an in-order traversal, you can use the print Elements function provided in the previous code

22501A0504



HEIGHT OF A TREE:

1. Recursively calculate the height of the left and right subtrees.
2. The height of the tree is the maximum of the heights of the left and right subtrees, plus 1.



Find the Min in Tree:

1. Define a variable min that will hold the root's data.
2. Traverse through the left sub-tree to find the smallest node.

3. Compare it with min and store the minimum of the two in the variable min.
4. Traverse through the right subtree to find the smallest node and compare it with min.
5. In the end, min will have the smallest node.

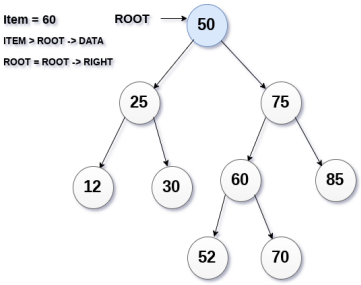
Find the Max in Tree:

1. Define variable max that will hold root's data.
2. Then, we traverse through the left sub-tree to find the largest node.
3. Compare it with max and store the maximum of two in a variable max.
4. Then, we traverse through the right subtree to find the largest node and compare it with max.
5. In the end, max will have the largest node.

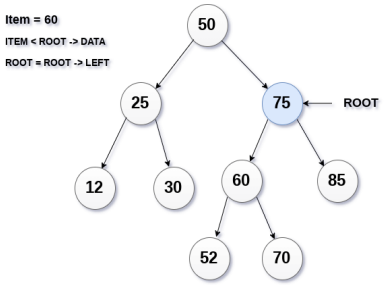
Searching an Element in the Tree:

1. Define a variable target that represents the element you want to search for.
2. Initialize a variable found to keep track of whether the element is found (initially set to false).
3. Start at the root node.
4. If the current node's data matches the target element, set found to true and exit the loop.
5. If the target element is less than the current node's data, move to the left child (if it exists).
6. If the target element is greater than the current node's data, move to the right child (if it exists).
7. Repeat steps 4-6 until either the element is found or you reach a leaf node with no more children.
8. If found is true, the element has been found.
9. If found is false, the element is not present in the tree

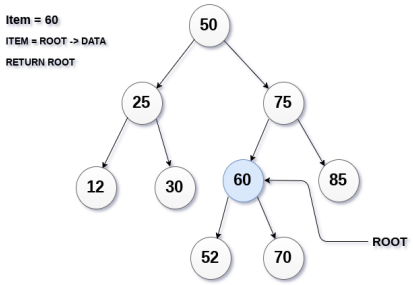
22501A0504



STEP 1



STEP 2



STEP 3

PROGRAM:

/*implement a binary tree using arrays perform tree traversals print leaf nodes print height of tree*/

```
#include<stdio.h>

int complete_node = 15;

int tree[15]={15,16,10,45,20,'\0',24,34,54,64,74};

int max,min,sum;

int print_tree() {

  printf("\n");

  for (int i = 0; i < 15; i++) {

    if (tree[i] != '\0')

      printf(" %d ",tree[i]);

    else

      printf(" - ");

  }

  return 0;
```

```
}
int get_right_child(int index)
{
    if(tree[index]!='\0' && ((2*index)+2)<complete_node)
        return (2*index)+2;
    return -1;
}
int get_left_child(int index)
{
    if(tree[index]!='\0' && (2*index)+1<complete_node)
        return (2*index)+1;
    return -1;
}
void preorder(int index)
{
    if(index>=0 && tree[index]!='\0')
    {
        printf(" %d ",tree[index]);
        preorder(get_left_child(index));
        preorder(get_right_child(index));
    }
}
void postorder(int index)
{
    if(index>=0 && tree[index]!='\0')
    {
```

```
        postorder(get_left_child(index));
        postorder(get_right_child(index));
        printf(" %d ",tree[index]);
    }
}
void inorder(int index)
{
    if(index>=0 && tree[index]!='\0')
    {
        inorder(get_left_child(index));
        printf(" %d ",tree[index]);
        {
            if(max<tree[index])
                max=tree[index];
            if(min>tree[index])
                min=tree[index];
            sum=sum+tree[index];
        }
        inorder(get_right_child(index));
    }
}
void levelorder()
{
    int j;
    for(j=0;j<complete_node;j++)
    {
```

```
        if(tree[j]!='\0')
            printf(" %d ",tree[j]);
    }
}
int is_leaf(int index)
{
    if(!get_left_child(index) && !get_right_child(index)){
        return 1;
    }
    if(tree[get_left_child(index)]=='\0' && tree[get_right_child(index)]=='\0'){
        return 1;
    }
    return 0;
}
int get_max(int a, int b)
{
    return (a>b) ? a : b;
}
int get_height(int index)
{
    if(tree[index]=='\0' || index<0 || is_leaf(index))
        return 0;
    return(get_max(get_height(get_left_child(index)), get_height(get_right_child(index)))+1);
}
int main()
{
```

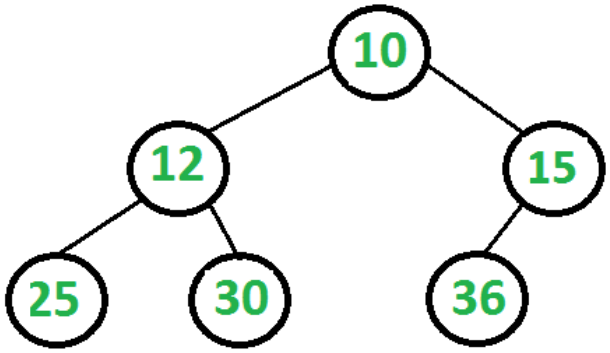
```
    print_tree();
    max=tree[0];
    min=tree[0];
    sum=0;
    printf("\nPreorder:\n");
    preorder(0);
    printf("\nPostorder:\n");
    postorder(0);
    printf("\nInorder:\n");
    inorder(0);
    printf("\n max=%d",max);
    printf("\n min=%d",min);
    printf("\n sum=%d",sum);
    printf("\nLevelorder:\n");
    levelorder();
    printf("\n");
    printf("\nheight of the tree is:%d\n",get_height(0));
    for(int i=0;i<complete_node;i++)
    {
      if(tree[i]!='\0'){
        if(is_leaf(i))
          printf("\n%d is a leaf node",tree[i]);
      }
    }
}
```

22501A0504

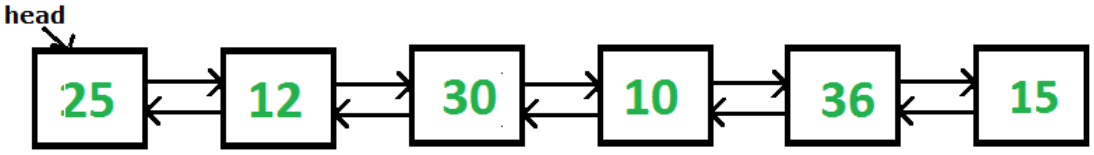
OUTPUT:

```
 15  16  10  45  20  -  24  34  54  64  74  -  -  -  -
Preorder:
 15  16  45  34  54  20  64  74  10  24
Postorder:
 34  54  45  64  74  20  16  24  10  15
Inorder:
 34  45  54  16  64  20  74  15  10  24
 max=74
 min=10
 sum=356
Levelorder:
 15  16  10  45  20  24  34  54  64  74

height of the tree is:3

24 is a leaf node
34 is a leaf node
54 is a leaf node
64 is a leaf node
74 is a leaf node
Process returned 0 (0x0)   execution time : 0.039 s
Press any key to continue.
```

OPERATIONS ON BINARY TREES USING LINKED LISTS:



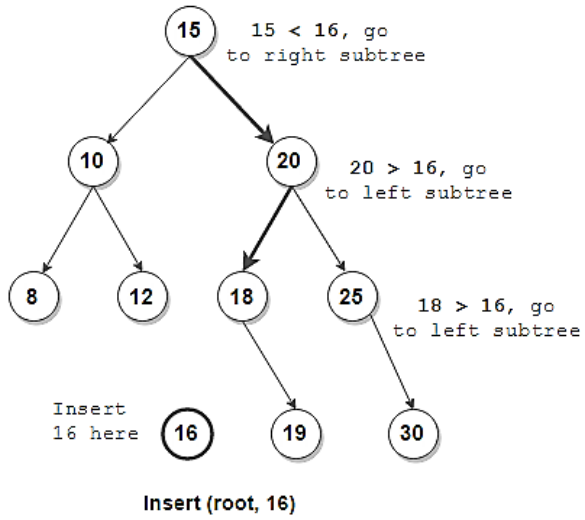
The above tree should be in-place converted to following Doubly Linked List(DLL).



Insertion:

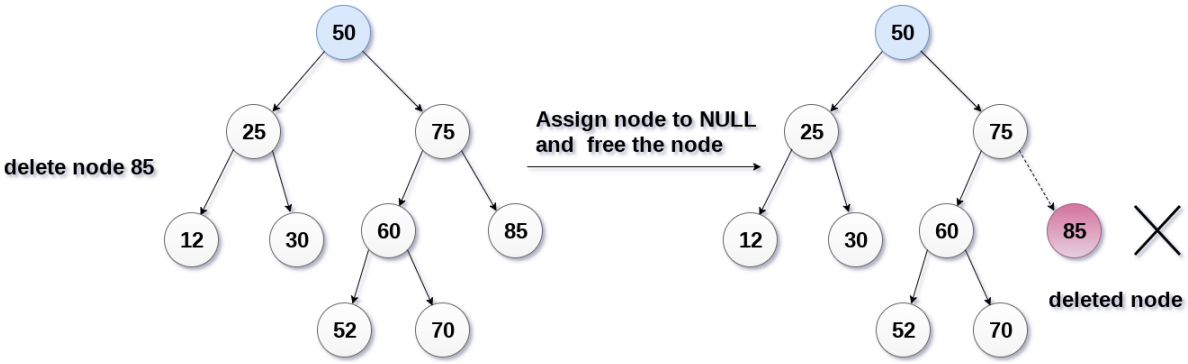
- Create a new node with the data to be inserted.
- If the tree is empty, set the root to the new node and return.
- Otherwise, start from the root and traverse down the tree.

- At each node, if the data to be inserted is less than the current node's data, move to the left child. If greater, move to the right child.
- Repeat steps 4 until you reach a null pointer.
- Set the left or right child of the null pointer to the new node, depending on whether the data is less or greater than the current node's data.



Deletion:

1. Search for the node containing the data to be deleted.
2. If the node has no children (leaf node), simply remove it.
3. If the node has one child, replace the node with its child.
4. If the node has two children, find the node's in-order successor (smallest node in its right subtree) or in-order predecessor (largest node in its left subtree).
5. Replace the node to be deleted with the in-order successor/predecessor.
6. Delete the in-order successor/predecessor from its original position.



Preorder Traversal:

- Start at the root node.
- Visit the current node.
- Recursively traverse the left subtree.
- Recursively traverse the right subtree

Inorder Traversal:

- Start at the root node.
- Recursively traverse the left subtree.
- Visit the current node.
- Recursively traverse the right subtree

Postorder Traversal:

- Start at the root node.
- Recursively traverse the left subtree.
- Recursively traverse the right subtree.
- Visit the current node.

Levelorder Traversal:

- Enqueue the root node.
- While the queue is not empty:

a. Dequeue a node and visit it.

b. Enqueue its left child.

c. Enqueue its right child

Print the Leaf Nodes:

- Start at the root node.
- If the current node is a leaf node (both left and right children are null), print its data.
- Recursively check the left and right subtrees.

Print the Height of the Tree:

- If the tree is empty, the height is -1.
- Otherwise, recursively find the height of the left and right subtrees.
- Return the maximum height of the left and right subtrees, plus 1.

Find the Max in Tree:

- Start at the root node.
- If the current node's data is greater than the maximum value, update the maximum value.
- Recursively check the left and right subtrees.

Find the Min in Tree:

- Start at the root node.
- If the current node's data is less than the minimum value, update the minimum value.
- Recursively check the left and right subtrees.

Searching an Element in the Tree:

- Start at the root node.
- If the current node's data matches the target value, return true.
- If the target value is less than the current node's data, recursively search the left subtree.

- If the target value is greater than the current node's data, recursively search the right subtree.
- If a null pointer is reached, return false.

Find Height and Breadth of Each Node:

1. Start at the root node with height and breadth values initialized to 0.
2. For each node, update its height as the height of its parent node plus 1.
3. For each node, update its breadth as the breadth of its parent node plus 1 if it is a right child, or the same breadth if it is a left child.
4. Recursively apply steps 2 and 3 to the left and right subtrees.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
struct node{
   int data;
   struct node *left;
   struct node *right;
};
struct node *root = NULL;
struct node* createNode(int data){
   struct node *newNode = (struct node*)malloc(sizeof(struct node));
   newNode->data = data;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
```

```
}
struct queue {
    int front, rear, size;
    struct node* *arr;
};
struct queue* createQueue() {
    struct queue* newQueue = (struct queue*)malloc(sizeof(struct queue));
    newQueue->front = -1;
    newQueue->rear = 0;
    newQueue->size = 0;
    newQueue->arr = (struct node**)malloc(100 * sizeof(struct node*));
    return newQueue;
}
void enqueue(struct queue* queue, struct node *temp) {
    queue->arr[queue->rear++] = temp;
    queue->size++;
}
struct node *dequeue(struct queue* queue) {
    queue->size--;
    return queue->arr[++queue->front];
}
void insertNode(int data) {
    struct node *newNode = createNode(data);
    if(root == NULL){
        root = newNode;
```

```
        return;
    }
    else {
        struct queue* queue = createQueue();
        enqueue(queue, root);
        while(true) {
            struct node *node = dequeue(queue);
            if(node->left != NULL && node->right != NULL) {
                enqueue(queue, node->left);
                enqueue(queue, node->right);
            }
            else {
                if(node->left == NULL) {
                    node->left = newNode;
                    enqueue(queue, node->left);
                }
                else {
                    node->right = newNode;
                    enqueue(queue, node->right);
                }
                break;
            }
        }
    }
}
```

```
void inorderTraversal(struct node *node) {
    if(root == NULL){
        printf("Tree is empty\n");
        return;
    }
    else {
        if(node->left != NULL)
            inorderTraversal(node->left);
        printf("%d ", node->data);
        if(node->right != NULL)
            inorderTraversal(node->right);
    }
}
void preorderTraversal(struct node *node) {
    if(root == NULL){
        printf("Tree is empty\n");
        return;
    }
    else {
        printf("%d ", node->data);
        if(node->left != NULL)
            preorderTraversal(node->left);
        if(node->right != NULL)
            preorderTraversal(node->right);
    }
```

```
}
void postorderTraversal(struct node *node) {
    if(root == NULL){
        printf("Tree is empty\n");
        return;
    }
    else {
        if(node->left != NULL)
            postorderTraversal(node->left);
        if(node->right != NULL)
            postorderTraversal(node->right);
        printf("%d ", node->data);
    }
}
void levelorderTraversal(struct node *node) {
    if(root == NULL){
        printf("Tree is empty\n");
        return;
    }
    else {
        struct queue* queue = createQueue();
        enqueue(queue, root);
        while(queue->size > 0) {
            struct node *current = dequeue(queue);
            printf("%d ", current->data);
```

```
            if(current->left != NULL)
                enqueue(queue, current->left);
            if(current->right != NULL)
                enqueue(queue, current->right);
        }
    }
}
void printLeafNodes(struct node *node) {
    if(root == NULL){
        printf("Tree is empty\n");
        return;
    }
    else {
        if (node == NULL)
            return;
        if (node->left == NULL && node->right == NULL)
            printf("%d ", node->data);
        else {
            printLeafNodes(node->left);
            printLeafNodes(node->right);
        }
    }
}
int findMax(struct node *node) {
    if(root == NULL){
```

```
        printf("Tree is empty\n");
        return -1;
    }
    else {
        if(node == NULL)
            return INT_MIN;
        int max = node->data;
        int leftMax = findMax(node->left);
        int rightMax = findMax(node->right);
        if (leftMax > max)
            max = leftMax;
        if (rightMax > max)
            max = rightMax;
        return max;
    }
}
int findMin(struct node *node) {
    if(root == NULL){
        printf("Tree is empty\n");
        return -1;
    }
    else {
        if(node == NULL)
            return INT_MAX;
        int min = node->data;
```

```
        int leftMin = findMin(node->left);
        int rightMin = findMin(node->right);
        if (leftMin < min)
            min = leftMin;
        if (rightMin < min)
            min = rightMin;

        return min;
    }
}
bool searchElement(struct node *node, int target) {
    if(root == NULL){
        printf("Tree is empty\n");
        return false;
    }
    else {
        if (node == NULL)
            return false;
        if (node->data == target)
            return true;
        return searchElement(node->left, target) || searchElement(node->right, target);
    }
}
int getHeight(struct node *node) {
    if(root == NULL){
```

```
        printf("Tree is empty\n");
        return -1;
    }
    else {
        if(node == NULL)
            return -1;
        int leftHeight = getHeight(node->left);
        int rightHeight = getHeight(node->right);
        return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);
    }
}
void getBreadth(struct node *node, int height, int *breadth) {
    if(root == NULL){
        printf("Tree is empty\n");
        return;
    }
    else {
        if(node == NULL)
            return;
        if(height == 0)
            (*breadth)++;
        else {
            getBreadth(node->left, height - 1, breadth);
            getBreadth(node->right, height - 1, breadth);
        }
```

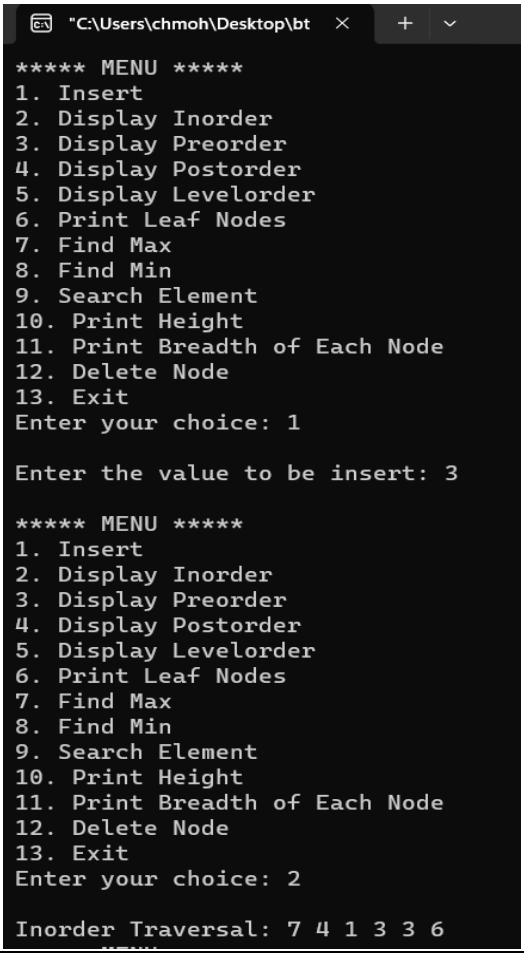
```
    }
}
void breadthOfEachNode(struct node *node) {
    if(root == NULL){
        printf("Tree is empty\n");
        return;
    }
    else {
        int height = getHeight(node);
        for (int i = 0; i <= height; i++) {
            int breadth = 0;
            getBreadth(node, i, &breadth);
            printf("Node at height %d: %d\n", i, breadth);
        }
    }
}
int main(){
    int choice, value;
    printf("\n----- Binary Tree -----\n");
    while(1){
        printf("\n***** MENU *****\n");
        printf("1. Insert\n2. Display Inorder\n3. Display Preorder\n4. Display
Postorder\n");
        printf("5. Display Levelorder\n6. Print Leaf Nodes\n7. Find Max\n8. Find Min\n");
```

```
        printf("9. Search Element\n10. Print Height\n11. Print Breadth of Each
Node\n12. Exit\n");
        printf("Enter your choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1:
                printf("\nEnter the value to be insert: ");
                scanf("%d", &value);
                insertNode(value);
                break;
            case 2:
                printf("\nInorder Traversal: ");
                inorderTraversal(root);
                break;
            case 3:
                printf("\nPreorder Traversal: ");
                preorderTraversal(root);
                break;
            case 4:
                printf("\nPostorder Traversal: ");
                postorderTraversal(root);
                break;
            case 5:
                printf("\nLevelorder Traversal: ");
                levelorderTraversal(root);
```

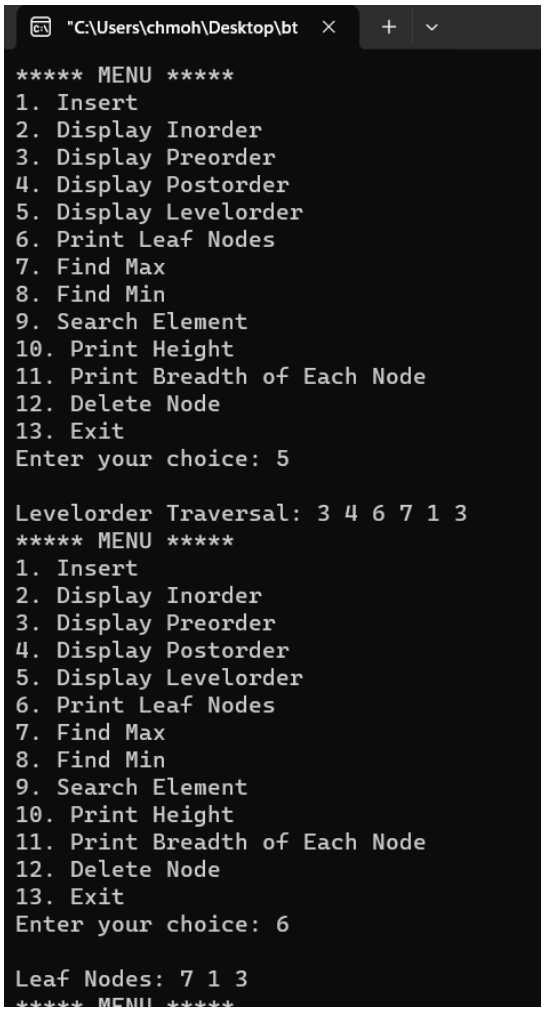
```
            break;
        case 6:
            printf("\nLeaf Nodes: ");
            printLeafNodes(root);
            break;
        case 7:
            printf("\nMax Element: %d", findMax(root));
            break;
        case 8:
            printf("\nMin Element: %d", findMin(root));
            break;
        case 9:
            printf("\nEnter element to be searched: ");
            scanf("%d", &value);
            if (searchElement(root, value))
                printf("Element found in the tree.\n");
            else
                printf("Element not found in the tree.\n");
            break;
        case 10:
            printf("\nHeight of the Tree: %d", getHeight(root));
            break;
        case 11:
            breadthOfEachNode(root);
            break;
```

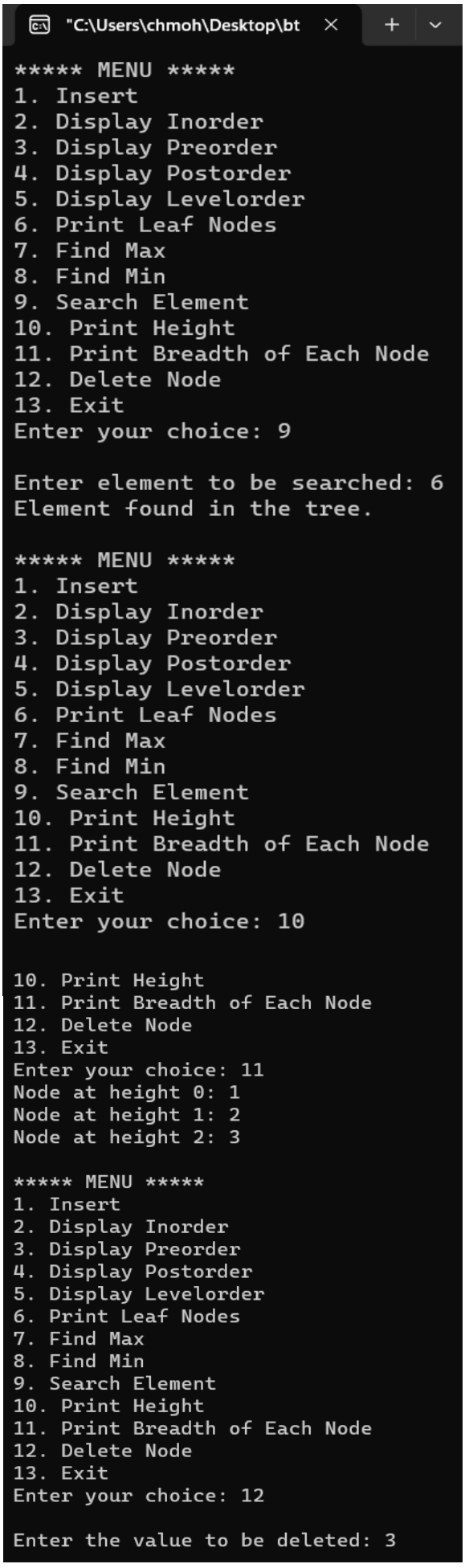
```
            case 12:
                exit(0);
            default:
                printf("\nPlease select correct operations!!!\n");
        } }
    return 0;
}
```

OUTPUT:



22501A0504


"C:\Users\chmoh\Desktop\bt
***** MENU *****
1. Insert
2. Display Inorder
3. Display Preorder
4. Display Postorder
5. Display Levelorder
6. Print Leaf Nodes
7. Find Max
8. Find Min
9. Search Element
10. Print Height
11. Print Breadth of Each Node
12. Delete Node
13. Exit
Enter your choice: 5
Levelorder Traversal: 3 4 6 7 1 3
***** MENU *****
1. Insert
2. Display Inorder
3. Display Preorder
4. Display Postorder
5. Display Levelorder
6. Print Leaf Nodes
7. Find Max
8. Find Min
9. Search Element
10. Print Height
11. Print Breadth of Each Node
12. Delete Node
13. Exit
Enter your choice: 6
Leaf Nodes: 7 1 3


"C:\Users\chmoh\Desktop\bt
***** MENU *****
1. Insert
2. Display Inorder
3. Display Preorder
4. Display Postorder
5. Display Levelorder
6. Print Leaf Nodes
7. Find Max
8. Find Min
9. Search Element
10. Print Height
11. Print Breadth of Each Node
12. Delete Node
13. Exit
Enter your choice: 9
Enter element to be searched: 6
Element found in the tree.
***** MENU *****
1. Insert
2. Display Inorder
3. Display Preorder
4. Display Postorder
5. Display Levelorder
6. Print Leaf Nodes
7. Find Max
8. Find Min
9. Search Element
10. Print Height
11. Print Breadth of Each Node
12. Delete Node
13. Exit
Enter your choice: 10
10. Print Height
11. Print Breadth of Each Node
12. Delete Node
13. Exit
Enter your choice: 11
Node at height 0: 1
Node at height 1: 2
Node at height 2: 3
***** MENU *****
1. Insert
2. Display Inorder
3. Display Preorder
4. Display Postorder
5. Display Levelorder
6. Print Leaf Nodes
7. Find Max
8. Find Min
9. Search Element
10. Print Height
11. Print Breadth of Each Node
12. Delete Node
13. Exit
Enter your choice: 12
Enter the value to be deleted: 3

22501A0504

```
Enter the value to be deleted: 3

***** MENU *****
1. Insert
2. Display Inorder
3. Display Preorder
4. Display Postorder
5. Display Levelorder
6. Print Leaf Nodes
7. Find Max
8. Find Min
9. Search Element
10. Print Height
11. Print Breadth of Each Node
12. Delete Node
13. Exit
Enter your choice: 2

Inorder Traversal: 7 4 1 3 6
```

```
"C:\Users\chmoh\Desktop\bt
***** MENU *****
1. Insert
2. Display Inorder
3. Display Preorder
4. Display Postorder
5. Display Levelorder
6. Print Leaf Nodes
7. Find Max
8. Find Min
9. Search Element
10. Print Height
11. Print Breadth of Each Node
12. Delete Node
13. Exit
Enter your choice: 1

Enter the value to be insert: 3

***** MENU *****
1. Insert
2. Display Inorder
3. Display Preorder
4. Display Postorder
5. Display Levelorder
6. Print Leaf Nodes
7. Find Max
8. Find Min
9. Search Element
10. Print Height
11. Print Breadth of Each Node
12. Delete Node
13. Exit
Enter your choice: 2

Inorder Traversal: 7 4 1 3 3 6
```