	MGM's COLLEGE OF ENGINEERING & TECHNOLOGY, NOIDA	LABORATORY MANUAL (DS)
	EXPERIMENT TITLE: WAP TO IMPLEMENT SORTING TECHNIQUE	
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING		
EXPERIMENT NO.: MGMCOET/CSE/ST(CS/AIML)/DSL-PROGEAM-1	SEMESTER : III(ST)	PAGE:1

AIM: Implementing Sorting Techniques: Bubble Sort, Insertion Sort, Selection Sort, Quick sort.

THEORY:

1. Bubble Sort:

Bubble sort is a simple and intuitive sorting algorithm. It repeatedly swaps adjacent elements if they are in the wrong order until the array is sorted. In this algorithm, the largest element "bubbles up" to the end of the array in each iteration. Bubble sort is inefficient for large data sets, but it is useful for educational purposes and small data sets. In this article, we will implement the bubble sort algorithm in C programming language.

The first step is to define the bubble sort function. This function takes an integer array and the size of the array as its parameters. The function returns nothing as it modifies the original array.

The function has two loops. The outer loop runs from the first element to the second-last element of the array. The inner loop runs from the first element to the second-last element of the unsorted part of the array. The condition of the inner loop is $n - i - 1$ because the last i elements of the array are already sorted.

In each iteration of the inner loop, we compare adjacent elements. If the left element is greater than the right element, we swap them. After the inner loop completes, the largest element is guaranteed to be at the end of the unsorted part of the array.

The main function creates an integer array `arr` of size 7 and initializes it with random numbers. We then calculate the size of the array by dividing the size of the array by the size of an integer element. Next, we call the `bubble_sort` function to sort the array. Finally, we print the sorted array using a for loop. This command compiles the `bubble_sort.c` file and produces an executable file named `bubble_sort`.

In summary, the bubble sort algorithm repeatedly swaps adjacent elements until the array is sorted. The algorithm has a time complexity of $O(n^2)$, which makes it inefficient for large data sets. However, it is useful for educational purposes and small data sets. We implemented the bubble sort algorithm in C programming language and tested it using a simple example.

Characteristics:

- Bubble sort is a simple sorting algorithm.
- It works by repeatedly swapping adjacent elements if they are in the wrong order.
- The algorithm sorts the array in ascending or descending order.
- It has a time complexity of $O(n^2)$ in the worst case, where n is the size of the array.

Usage:

- Bubble sort is useful for educational purposes and small data sets.
- It is not suitable for large data sets because of its time complexity.

Advantages:

- Bubble sort is easy to understand and implement.
- It requires minimal additional memory space to perform the sorting.

Disadvantages:

- It is not efficient for large data sets because of its time complexity.
- It has poor performance compared to other sorting algorithms, such as quicksort and mergesort

Bubble sort is a simple and intuitive sorting algorithm that is useful for educational purposes and small data sets. However, its time complexity makes it inefficient for large data sets. Therefore, it is not commonly used in real-world applications. Other sorting algorithms, such as quicksort and mergesort, are more efficient for large data sets.

PROGRAM:

```
// C program for implementation of Bubble sort
```

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

```
// Swap function
```

```
void swap(int* arr, int i, int j)
```

```
{
```

```
    int temp = arr[i];
```

```
    arr[i] = arr[j];
```

```
    arr[j] = temp;
```

```
}
```

```
// A function to implement bubble sort
```

```
void bubbleSort(int arr[], int n)
```

```
{
```

```
    int i, j;
```

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

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

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

// Driver code
int main()
{
    int arr[] = { 5, 1, 4, 2, 8 };
    int N = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, N);
    printf("Sorted array: ");
    printArray(arr, N);
    return 0;
}
```

OUTPUT:

Output

```
/tmp/8AxvJ8puyG.o
Sorted array: 1 2 4 5 8
```

2. Insertion Sort:

The working procedure of insertion sort is also simple. This article will be very helpful and interesting to students as they might face insertion sort as a question in their examinations. So, it is important to discuss the topic.

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected

unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is $O(n^2)$, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as –

Simple implementation

- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Now, let's see the algorithm of insertion sort.

Algorithm

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step 2 - Pick the next element, and store it separately in a **key**.

Step 3 - Now, compare the **key** with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

Working of Insertion sort Algorithm

Now, let's see the working of the insertion sort Algorithm.

To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.

Let the elements of array are -

12	31	25	8	32	17
----	----	----	---	----	----

Initially, the first two elements are compared in insertion sort.

12	31	25	8	32	17
----	----	----	---	----	----

Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.

12	31	25	8	32	17
----	----	----	---	----	----

Now, move to the next two elements and compare them.

12	31	25	8	32	17
----	----	----	---	----	----

12	31	25	8	32	17
----	----	----	---	----	----

Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

12	25	31	8	32	17
----	----	----	---	----	----

Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

12	25	31	8	32	17
----	----	----	---	----	----

12	25	31	8	32	17
----	----	----	---	----	----

Both 31 and 8 are not sorted. So, swap them.

12	25	8	31	32	17
----	----	---	----	----	----

After swapping, elements 25 and 8 are unsorted.

12	25	8	31	32	17
----	----	---	----	----	----

So, swap them.

12	8	25	31	32	17
----	---	----	----	----	----

Now, elements 12 and 8 are unsorted.

12	8	25	31	32	17
----	---	----	----	----	----

So, swap them too.

8	12	25	31	32	17
---	----	----	----	----	----

Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.

8	12	25	31	32	17
---	----	----	----	----	----

Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

8	12	25	31	32	17
---	----	----	----	----	----

Move to the next elements that are 32 and 17.

8	12	25	31	32	17
---	----	----	----	----	----

17 is smaller than 32. So, swap them.

8	12	25	31	17	32
---	----	----	----	----	----

8	12	25	31	17	32
---	----	----	----	----	----

Swapping makes 31 and 17 unsorted. So, swap them too.

8	12	25	17	31	32
---	----	----	----	----	----

8	12	25	17	31	32
---	----	----	----	----	----

Now, swapping makes 25 and 17 unsorted. So, perform swapping again.

8	12	17	25	31	32
---	----	----	----	----	----

Now, the array is completely sorted.

PROGRAM;

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

```
void insert(int a[], int n) /* function to sort an aay with insertion sort */
```

```
{
```

```
    int i, j, temp;
```

```
    for (i = 1; i < n; i++) {
```

```
        temp = a[i];
```

```
        j = i - 1;
```

```
        while(j >= 0 && temp <= a[j]) /* Move the elements greater than temp to one position ahead from their current position*/
```

```
        {
```

```
            a[j+1] = a[j];
```

```
            j = j-1;
```

```
        }
```

```
        a[j+1] = temp;
```

```
    }
```

```
}
```

```
void printArr(int a[], int n) /* function to print the array */
```

```
{
```

```
    int i;
```

```
    for (i = 0; i < n; i++)
```

```
        printf("%d ", a[i]);
```

```
}
```

```
int main()
```

```
{
```

```
    int a[] = { 12, 31, 25, 8, 32, 17 };
```

```
int n = sizeof(a) / sizeof(a[0]);
printf("Before sorting array elements are - \n");
printArr(a, n);
insert(a, n);
printf("\nAfter sorting array elements are - \n");
printArr(a, n);

return 0;
}
```

OUTPUT:

```
Output
/tmp/uEKkgdj2Rv.o
Before sorting array elements are -
12 31 25 8 32 17
After sorting array elements are -
8 12 17 25 31 32
```

3. Selection Sort:

The working procedure of selection sort is also simple. This article will be very helpful and interesting to students as they might face selection sort as a question in their examinations. So, it is important to discuss the topic.

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

The average and worst-case complexity of selection sort is $O(n^2)$, where n is the number of items. Due to this, it is not suitable for large data sets. Selection sort is generally used when -

- A small array is to be sorted
- Swapping cost doesn't matter
- It is compulsory to check all elements

Now, let's see the algorithm of selection sort.

1. SELECTION SORT(arr, n)
- 2.
3. Step 1: Repeat Steps 2 **and** 3 **for** i = 0 to n-1
4. Step 2: CALL SMALLEST(arr, i, n, pos)
5. Step 3: SWAP arr[i] with arr[pos]
6. [END OF LOOP]
7. Step 4: EXIT
- 8.
9. SMALLEST (arr, i, n, pos)
10. Step 1: [INITIALIZE] SET SMALL = arr[i]
11. Step 2: [INITIALIZE] SET pos = i
12. Step 3: Repeat **for** j = i+1 to n
13. **if** (SMALL > arr[j])
14. SET SMALL = arr[j]
15. SET pos = j
16. [END OF **if**]
17. [END OF LOOP]
18. Step 4: RETURN pos

Working of Selection sort Algorithm

Now, let's see the working of the Selection sort Algorithm.

To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

Let the elements of array are -

12	29	25	8	32	17	40
----	----	----	---	----	----	----

Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

12	29	25	8	32	17	40
----	----	----	---	----	----	----

So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

8	29	25	12	32	17	40
---	----	----	----	----	----	----

For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

8	29	25	12	32	17	40
---	----	----	----	----	----	----

Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.

8	12	25	29	32	17	40
---	----	----	----	----	----	----

The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.



Now, the array is completely sorted.

PROGRAM;

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

```
void selection(int arr[], int n)
```

```
{
```

```
    int i, j, small;
```

```
    for (i = 0; i < n-1; i++) // One by one move boundary of unsorted subarray
```

```
    {
```

```
        small = i; //minimum element in unsorted array
```

```
        for (j = i+1; j < n; j++)
```

```
            if (arr[j] < arr[small])
```

```
                small = j;
```

```
// Swap the minimum element with the first element
```

```
    int temp = arr[small];
```

```
    arr[small] = arr[i];
```

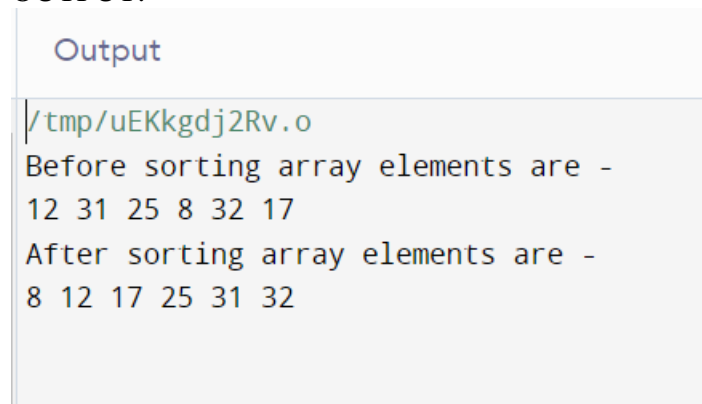
```
        arr[i] = temp;
    }
}

void printArr(int a[], int n) /* function to print the array */
{
    int i;

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

int main()
{
    int a[] = { 12, 31, 25, 8, 32, 17 };
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    selection(a, n);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);
    return 0;
}
```

OUTPUT:



```
Output
/tmp/uEKkgdj2Rv.o
Before sorting array elements are -
12 31 25 8 32 17
After sorting array elements are -
8 12 17 25 31 32
```

4. Quick Sort: The working procedure of Quicksort is also simple. This article will be very helpful and interesting to students as they might face quicksort as a question in their examinations. So, it is important to discuss the topic.

Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes **$n \log n$** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems,

then solving the subproblems, and combining the results back together to solve the original problem.

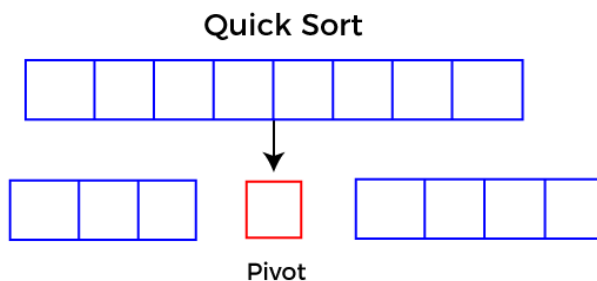
Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Algorithm

1. QUICKSORT (array A, start, end)
2. {
3. **if** (start < end)
4. {

```
5.  p = partition(A, start, end)
6.  QUICKSORT (A, start, p - 1)
7.  QUICKSORT (A, p + 1, end)
8.  }
9.  }
10. PARTITION (array A, start, end)
11. {
12.  pivot ← A[end]
13.  i ← start-1
14.  for j ← start to end -1 {
15.    do if (A[j] < pivot) {
16.      then i ← i + 1
17.      swap A[i] with A[j]
18.    } }
19.  swap A[i+1] with A[end]
20.  return i+1
21. }
```

Working of Quick Sort Algorithm

Now, let's see the working of the Quicksort Algorithm.

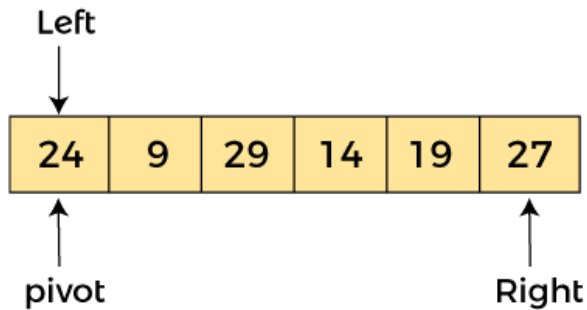
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

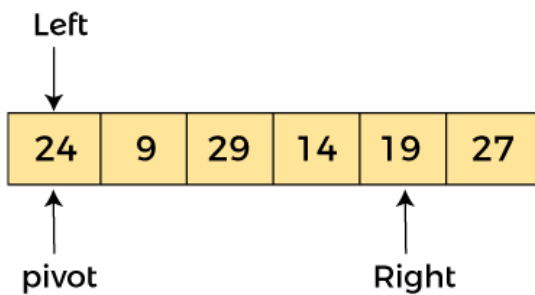
24	9	29	14	19	27
----	---	----	----	----	----

In the given array, we consider the leftmost element as pivot. So, in this case, $a[\text{left}] = 24$, $a[\text{right}] = 27$ and $a[\text{pivot}] = 24$.

Since, pivot is at left, so algorithm starts from right and move towards left.

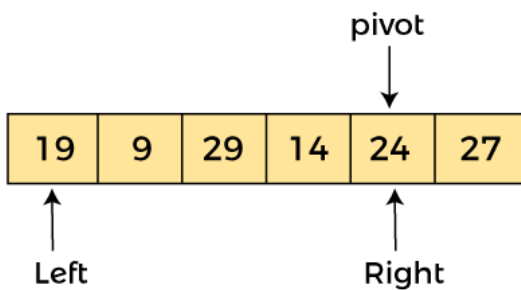


Now, $a[\text{pivot}] < a[\text{right}]$, so algorithm moves forward one position towards left, i.e. -



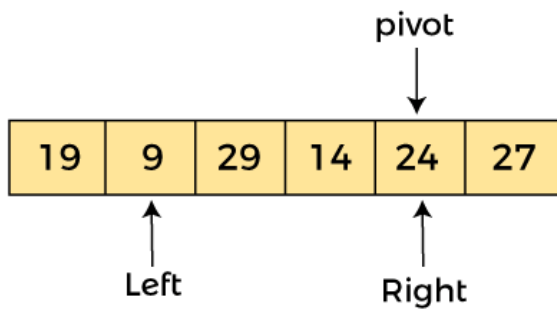
Now, $a[\text{left}] = 24$, $a[\text{right}] = 19$, and $a[\text{pivot}] = 24$.

Because, $a[\text{pivot}] > a[\text{right}]$, so, algorithm will swap $a[\text{pivot}]$ with $a[\text{right}]$, and pivot moves to right, as -

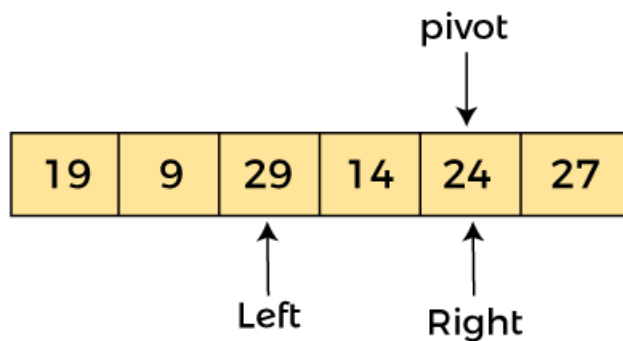


Now, $a[\text{left}] = 19$, $a[\text{right}] = 24$, and $a[\text{pivot}] = 24$. Since, pivot is at right, so algorithm starts from left and moves to right.

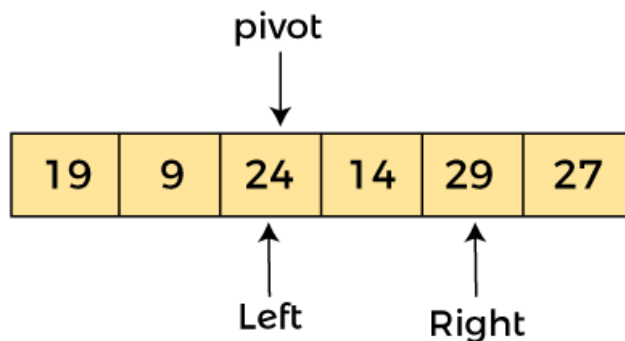
As $a[\text{pivot}] > a[\text{left}]$, so algorithm moves one position to right as -



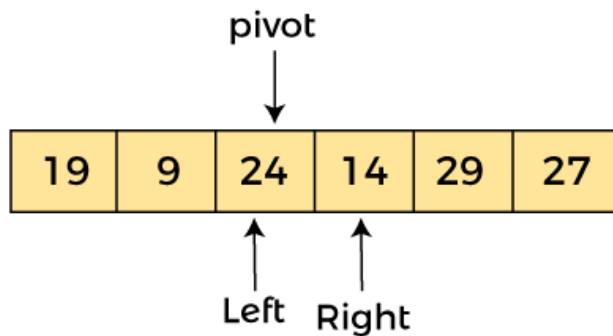
Now, $a[\text{left}] = 9$, $a[\text{right}] = 24$, and $a[\text{pivot}] = 24$. As $a[\text{pivot}] > a[\text{left}]$, so algorithm moves one position to right as -



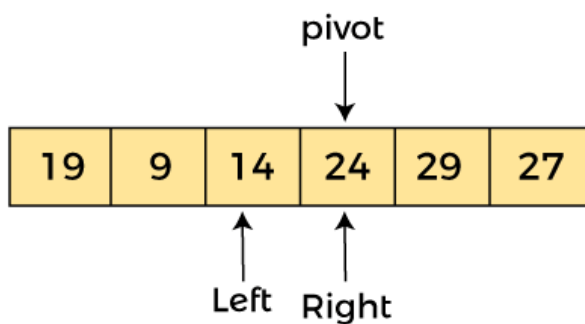
Now, $a[\text{left}] = 29$, $a[\text{right}] = 24$, and $a[\text{pivot}] = 24$. As $a[\text{pivot}] < a[\text{left}]$, so, swap $a[\text{pivot}]$ and $a[\text{left}]$, now pivot is at left, i.e. -



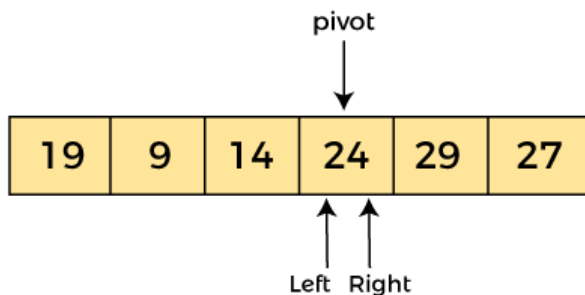
Since, pivot is at left, so algorithm starts from right, and move to left. Now, $a[\text{left}] = 24$, $a[\text{right}] = 29$, and $a[\text{pivot}] = 24$. As $a[\text{pivot}] < a[\text{right}]$, so algorithm moves one position to left, as -



Now, $a[\text{pivot}] = 24$, $a[\text{left}] = 24$, and $a[\text{right}] = 14$. As $a[\text{pivot}] > a[\text{right}]$, so, swap $a[\text{pivot}]$ and $a[\text{right}]$, now pivot is at right, i.e. -



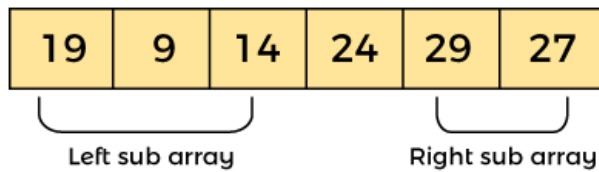
Now, $a[\text{pivot}] = 24$, $a[\text{left}] = 14$, and $a[\text{right}] = 24$. Pivot is at right, so the algorithm starts from left and move to right.



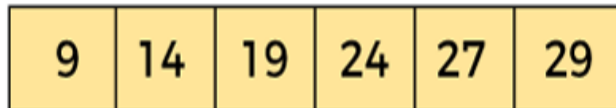
Now, $a[\text{pivot}] = 24$, $a[\text{left}] = 24$, and $a[\text{right}] = 24$. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



PROGRAM:

```
int partition (int a[], int start, int end)
{
    int pivot = a[end]; // pivot element
    int i = (start - 1);

    for (int j = start; j <= end - 1; j++)
    {
        // If current element is smaller than the pivot
        if (a[j] < pivot)
        {
            i++; // increment index of smaller element

            int t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
}
```

```
    }

    int t = a[i+1];

    a[i+1] = a[end];

    a[end] = t;

    return (i + 1);

}

/* function to implement quick sort */

void quick(int a[], int start, int end) /* a[] = array to be sorted, start = Starting index, end =
Ending index */

{
    if (start < end)
    {
        int p = partition(a, start, end); //p is the partitioning index

        quick(a, start, p - 1);

        quick(a, p + 1, end);
    }
}

/* function to print an array */

void printArr(int a[], int n)

{
    int i;


    for (i = 0; i < n; i++)
```

```
        printf("%d ", a[i]);  
    }  
  
int main()  
{  
    int a[] = { 24, 9, 29, 14, 19, 27 };  
    int n = sizeof(a) / sizeof(a[0]);  
    printf("Before sorting array elements are - \n");  
    printArr(a, n);  
    quick(a, 0, n - 1);  
    printf("\nAfter sorting array elements are - \n");  
    printArr(a, n);  
  
    return 0;  
}
```

OUTPUT:

Output

```
/tmp/uEKkgdj2Rv.o  
Before sorting array elements are -  
24 9 29 14 19 27  
After sorting array elements are -  
9 14 19 24 27 29 |
```

	MGM's COLLEGE OF ENGINEERING & TECHNOLOGY, NOIDA	LABORATORY MANUAL (DS)
	EXPERIMENT TITLE: WAP IMPLEMENTING SEARCHING AND HASHING TECHNIQUES:	
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING		
EXPERIMENT NO.: MGMCOET/CSE/ST(CS/AIML)/DSL-PROGEAM-2	SEMESTER : III(ST)	PAGE:21

AIM: Implementing Searching and Hashing Techniques: Linear search, Binary search

THEORY:

Linear search:

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element; otherwise, the search is called unsuccessful.

Two popular search methods are Linear Search and Binary Search. So, here we will discuss the popular searching technique, i.e., Linear Search Algorithm.

Linear search is also called as **sequential search algorithm**. It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted. The worst-case time complexity of linear search is **O(n)**.

The steps used in the implementation of Linear Search are listed as follows -

- First, we have to traverse the array elements using a **for** loop.
- In each iteration of **for loop**, compare the search element with the current array element, and -
 - If the element matches, then return the index of the corresponding array element.
 - If the element does not match, then move to the next element.
- If there is no match or the search element is not present in the given array, return **-1**.

Now, let's see the algorithm of linear search.

Algorithm

1. Linear_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search

2. Step 1: set pos = -1
3. Step 2: set i = 1
4. Step 3: repeat step 4 while i <= n
5. Step 4: if a[i] == val
6. set pos = i
7. print pos
8. go to step 6
9. [end of if]
10. set ii = i + 1
11. [end of loop]
12. Step 5: if pos = -1
13. print "value is not present in the array "
14. [end of if]
15. Step 6: exit

Working of Linear search

Now, let's see the working of the linear search Algorithm.

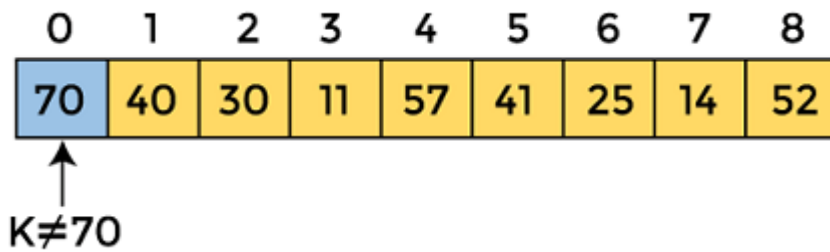
To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

Let the elements of array are -

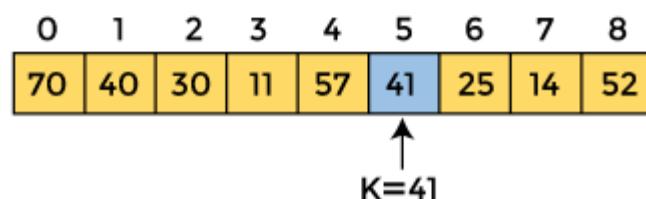
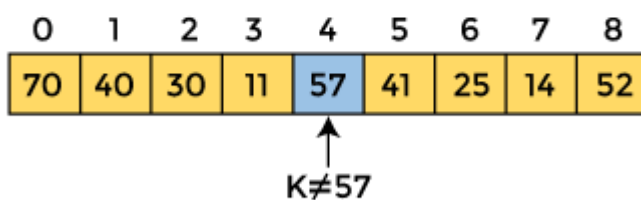
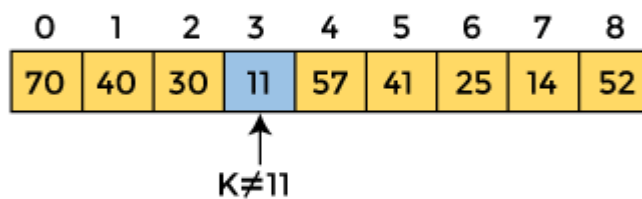
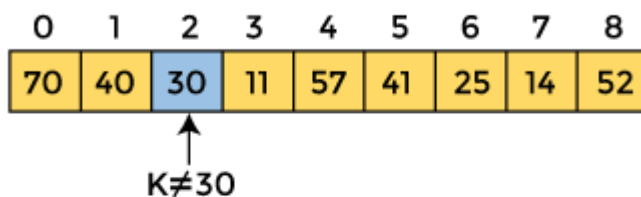
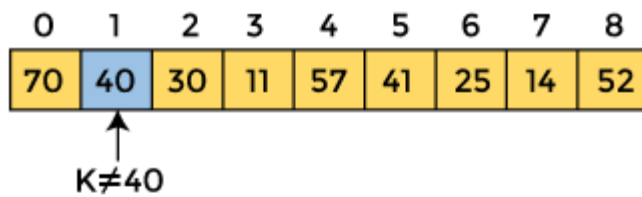
0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

Let the element to be searched is **K = 41**

Now, start from the first element and compare **K** with each element of the array.



The value of K , i.e., **41**, is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.



Now, the element to be searched is found. So algorithm will return the index of the element matched.

PROGRAM:

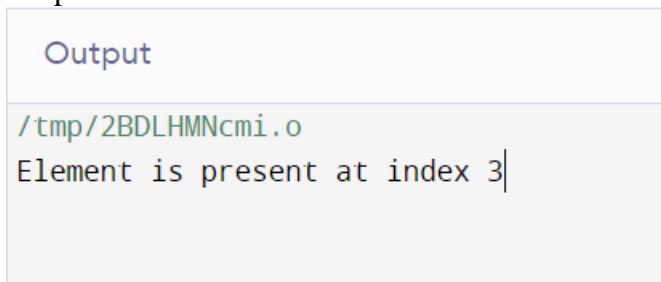
```
#include <stdio.h>

int search(int arr[], int N, int x)
{
    for (int i = 0; i < N; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

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

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

Output:



```
Output
/tmp/2BDLHMNcmi.o
Element is present at index 3|
```

Binary Search:

Binary Search Algorithm

In this article, we will discuss the Binary Search Algorithm. Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element. Otherwise, the search is called unsuccessful.

Linear Search and Binary Search are the two popular searching techniques. Here we will discuss the Binary Search Algorithm.

Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

Algorithm

1. Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array, 'lower_bound' is the index of the first array element, 'upper_bound' is the index of the last array element, 'val' is the value to search
2. Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
3. Step 2: repeat steps 3 and 4 while beg <=end
4. Step 3: set mid = (beg + end)/2
5. Step 4: if a[mid] = val
6. set pos = mid
7. print pos
8. go to step 6
9. else if a[mid] > val
10. set end = mid - 1
11. else
12. set beg = mid + 1
13. [end of if]
14. [end of loop]
15. Step 5: if pos = -1
16. print "value is not present in the array"
17. [end of if]
18. Step 6: exit

Working of Binary search

Now, let's see the working of the Binary Search Algorithm.

To understand the working of the Binary search algorithm, let's take a sorted array. It will be easy to understand the working of Binary search with an example.

There are two methods to implement the binary search algorithm -

- Iterative method
- Recursive method

The recursive method of binary search follows the divide and conquer approach.

Let the elements of array are -

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

Let the element to search is, **K = 56**

We have to use the below formula to calculate the **mid** of the array -

1. $\text{mid} = (\text{beg} + \text{end})/2$

So, in the given array -

beg = 0

end = 8

mid = $(0 + 8)/2 = 4$. So, 4 is the mid of the array.

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69



$A[mid] = 39$
 $A[mid] < K$ (or, $39 < 56$)
 So, $beg = mid + 1 = 5$, $end = 8$
 Now, $mid = (beg + end) / 2 = 13 / 2 = 6$

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69



$A[mid] = 51$
 $A[mid] < K$ (or, $51 < 56$)
 So, $beg = mid + 1 = 7$, $end = 8$
 Now, $mid = (beg + end) / 2 = 15 / 2 = 7$

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69



$A[mid] = 56$
 $A[mid] = K$ (or, $56 = 56$)
 So, $location = mid$
 Element found at 7th location of the array

Now, the element to search is found. So algorithm will return the index of the element matched.

PROGRAM :

// C program to implement iterative Binary Search

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

// An iterative binary search function.

```
int binarySearch(int arr[], int l, int r, int x)
```

```
{
```

```
    while (l <= r) {
        int m = l + (r - l) / 2;
```

```
        // Check if x is present at mid
```

```
        if (arr[m] == x)
```

```
    return m;

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

    // If x is smaller, ignore right half
    else
        r = m - 1;
}


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

// Driver code
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? printf("Element is not present"
                          " in array")
                  : printf("Element is present at "
                          "index %d",
                          result);

    return 0;
}
```

Output:

```
Output
/tmp/2BDLHMNcmi.o
Element is present at index 4
```

	MGM's COLLEGE OF ENGINEERING & TECHNOLOGY, NOIDA	LABORATORY MANUAL (DS)
	EXPERIMENT TITLE: WAP IMPLEMENTING STACKS:.	
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING		
EXPERIMENT NO.: MGMCOET/CSE/ST(CS/AIML)/DSL-PROGEAM-3	SEMESTER : III(ST)	PAGE:29

AIM: Implementing Stacks: Array implementation, Linked List implementation, Evaluation of postfix expression and balancing of parenthesis , Conversion of infix notation to postfix notation

Array implementation:

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refer to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:

1. begin
2. **if** top = n then stack full
3. top = top + 1
4. stack (top) := item;
5. end

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.

The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm:

1. begin
2. **if** top = 0 then stack empty;
3. item := stack(top);
4. top = top - 1;
5. end;

Visiting each element of the stack (Peek operation)

Peek operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm :

PEEK (STACK, TOP)

1. Begin
2. **if** top = -1 then stack empty
3. item = stack[top]
4. **return** item
5. End

Program:

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

```
int stack[100],i,j,choice=0,n,top=-1;
```

```
void push();
```

```
void pop();
```

```
void show();

void main ()
{

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

    printf("*****Stack operations using array*****");

    printf("\n-----\n");

    while(choice != 4)
    {
        printf("Choose one from the below options...\n");
        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
        printf("\n Enter your choice \n");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
```

```
        break;
    }
    case 3:
    {
        show();
        break;
    }
    case 4:
    {
        printf("Exiting....");
        break;
    }
    default:
    {
        printf("Please Enter valid choice ");
    }
};
}
```

```
void push ()
{
    int val;
    if (top == n )
        printf("\n Overflow");
}
```



```
else
{
    printf("Enter the value?");
    scanf("%d",&val);
    top = top +1;
    stack[top] = val;
}
}
```

```
void pop ()
{
    if(top == -1)
        printf("Underflow");
    else
        top = top -1;
}
```

```
void show()
{   printf("The elements are:");
    for (i=top;i>=0;i--)
    {
        printf("%d\n",stack[i]);
    }
    if(top == -1)
    {
        printf("Stack is empty");
    }
}
```

```
}  
  
}
```

OUTPUT:

Enter the number of elements in the stack 3

*****Stack operations using array*****

Choose one from the below options...

1.Push

2.Pop

3.Show

4.Exit

Enter your choice

1

Enter the value?3

Choose one from the below options...

1.Push

2.Pop

3.Show

4.Exit

Enter your choice

1

Enter the value?6

Choose one from the below options...

1.Push

2.Pop

3.Show

4.Exit

Enter your choice

1

Enter the value?9

Choose one from the below options...

1.Push

2.Pop

3.Show

4.Exit

Enter your choice


3

The elements are:

9

6

3

	MGM's COLLEGE OF ENGINEERING & TECHNOLOGY, NOIDA	LABORATORY MANUAL (DS)
	EXPERIMENT TITLE: WAP IMPLEMENTING CIRCULAR QUEUE	
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING		
EXPERIMENT NO.: MGMCOET/CSE/ST(CS/AIML)/DSL-PROGEAM-4	SEMESTER : III(ST)	PAGE:36

AIM: Implementing Circular Queue using Linked List in C

enqueue(data)

- Create a struct node type node.
- Insert the given data in the new node data section and NULL in address section.
- If Queue is empty then initialize front and rear from new node.
- Queue is not empty then initialize rear next and rear from new node.
- New node next initialize from front

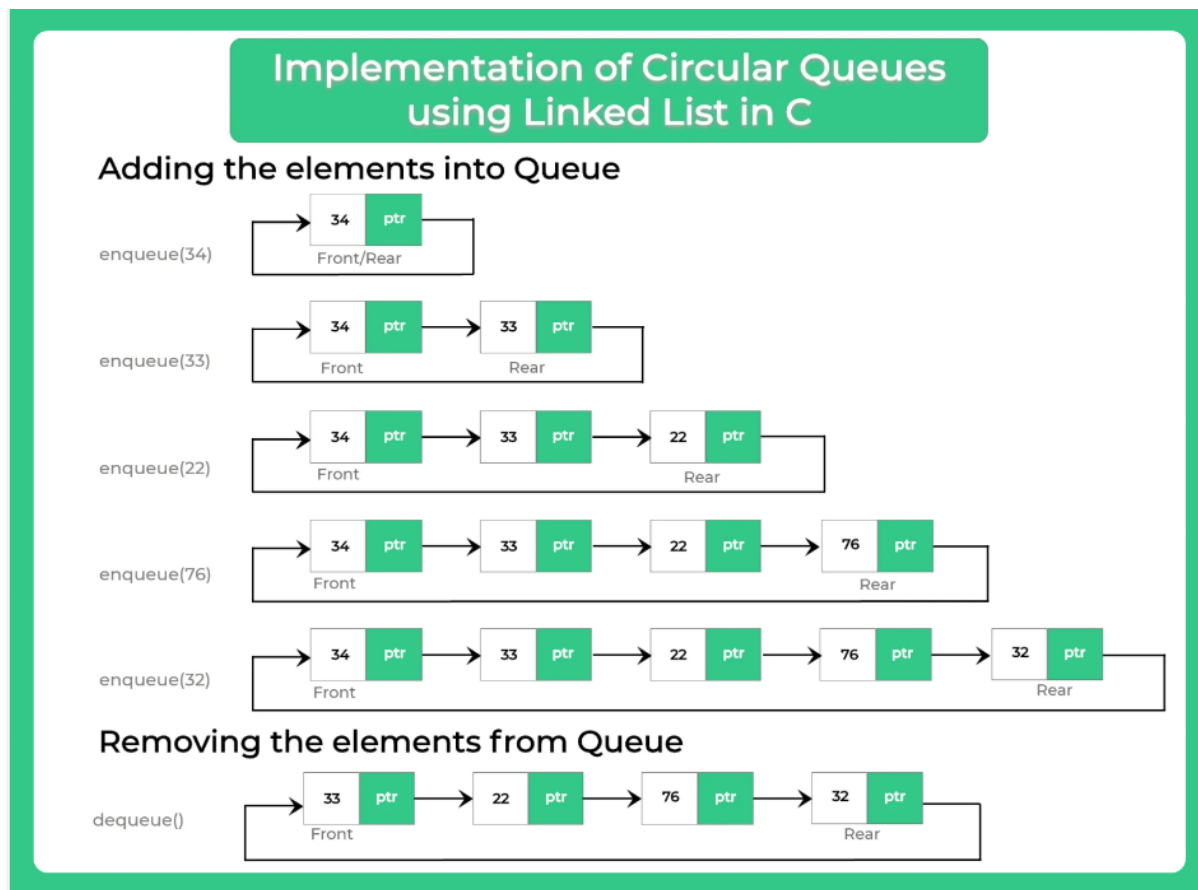
dequeue()

- Check if queue is empty or not.
- If queue is empty then dequeue is not possible.
- Else Initialize temp from front.
- If front is equal to the rear then initialize front and rear from null.
- Print data of temp and free temp memory.
- If there is more than one node in Queue then make front next to front then initialize rear next from front.
- Print temp and free temp.

print()

- Check if there is some data in the queue or not.
- If the queue is empty print "No data in the queue."
- Else define a node pointer and initialize it with front.

- Print data of node pointer until the next of node pointer becomes NULL.



PROGRAM:

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

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

```
struct node *f = NULL;
struct node *r = NULL;
```

```
void enqueue (int d)                //Insert elements in Queue
{
    struct node *n;
    n = (struct node *) malloc (sizeof (struct node));
    n->data = d;
```

```
n->next = NULL;
if ((r == NULL) && (f == NULL))
{
    f = r = n;
    r->next = f;
}
else
{
    r->next = n;
    r = n;
    n->next = f;
}
}

void dequeue ()                // Delete an element from Queue
{
    struct node *t;
    t = f;
    if ((f == NULL) && (r == NULL))
        printf ("\nQueue is Empty");
    else if (f == r)
    {
        f = r = NULL;
        free (t);
    }
    else
    {
        f = f->next;
        r->next = f;
        free (t);
    }
}

void display ()
{
    struct node *t;                // Print the elements of Queue
    t = f;
    if ((f == NULL) && (r == NULL))
        printf ("\nQueue is Empty");
    else
    {
        do
        {
            printf (" %d", t->data);
```


```
        t = t->next;
    }
    while (t != f);
}

int main ()
{
    enqueue (34);
    enqueue (22);
    enqueue (75);
    enqueue (99);
    enqueue (27);
    printf ("Circular Queue: ");
    display ();
    printf ("\n");

    dequeue ();
    printf ("Circular Queue After dequeue: ");
    display ();
    return 0;
}
```

OUTPUT:**Output**

```
/tmp/8AxvJ8puyG.o
Circular Queue:  34 22 75 99 27
Circular Queue After dequeue:  22 75 99 27|
```

	MGM's COLLEGE OF ENGINEERING & TECHNOLOGY, NOIDA	LABORATORY MANUAL (DS)
EXPERIMENT TITLE: WAP FOR BINARY SEARCH TREE		
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING		
EXPERIMENT NO.: MGMCOET/CSE/ST (CS/AIML)/DSL-PROGEAM-5	SEMESTER : III(ST)	PAGE:40

AIM: Implementing Trees: Binary search tree

THEORY:

In a binary tree, every node can have maximum of two children but there is no order of nodes based on their values. In binary tree, the elements are arranged as they arrive to the tree, from top to bottom and left to right.

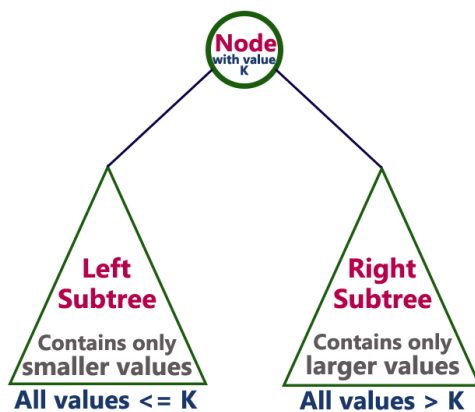
A binary tree has the following time complexities...

1. Search Operation - $O(n)$
2. Insertion Operation - $O(1)$
3. Deletion Operation - $O(n)$

To enhance the performance of binary tree, we use special type of binary tree known as Binary Search Tree. Binary search tree mainly focus on the search operation in binary tree. Binary search tree can be defined as follows...

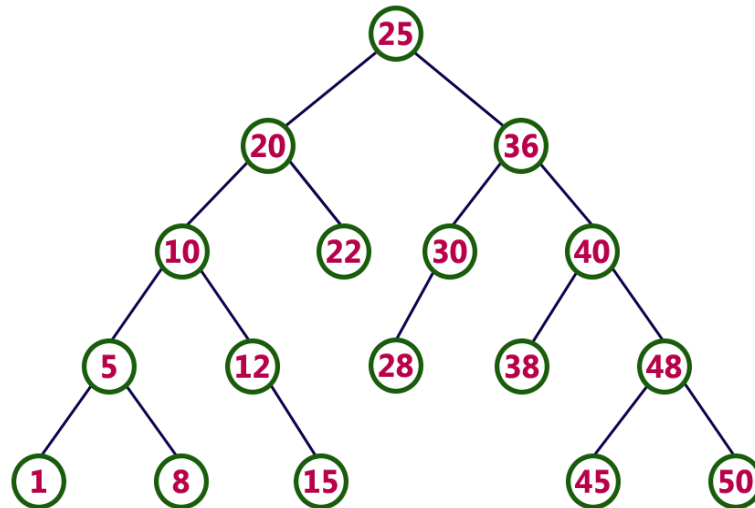
Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

In a binary search tree, all the nodes in left subtree of any node contains smaller values and all the nodes in right subtree of that contains larger values as shown in following figure...



Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



Every Binary Search Tree is a binary tree but all the Binary Trees need not to be binary search trees.

The following operations are performed on a binary search tree...

1. Search
2. Insertion
3. Deletion

Search Operation in BST

In a binary search tree, the search operation is performed with $O(\log n)$ time complexity. The search operation is performed as follows...

- Step 1: Read the search element from the user
- Step 2: Compare, the search element with the value of root node in the tree.

- Step 3: If both are matching, then display "Given node found!!!" and terminate the function
- Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.
- Step 5: If search element is smaller, then continue the search process in left subtree.
- Step 6: If search element is larger, then continue the search process in right subtree.
- Step 7: Repeat the same until we found exact element or we completed with a leaf node
- Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.
- Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in BST

In a binary search tree, the insertion operation is performed with $O(\log n)$ time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

- Step 1: Create a newNode with given value and set its left and right to NULL.
- Step 2: Check whether tree is Empty.
- Step 3: If the tree is Empty, then set root to newNode.
- Step 4: If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node (here it is root node).
- Step 5: If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.
- Step 6: Repeat the above step until we reach to a leaf node (e.i., reach to NULL).
- Step 7: After reaching a leaf node, then insert the newNode as left child if newNode is smaller or equal to that leaf else insert it as right child.

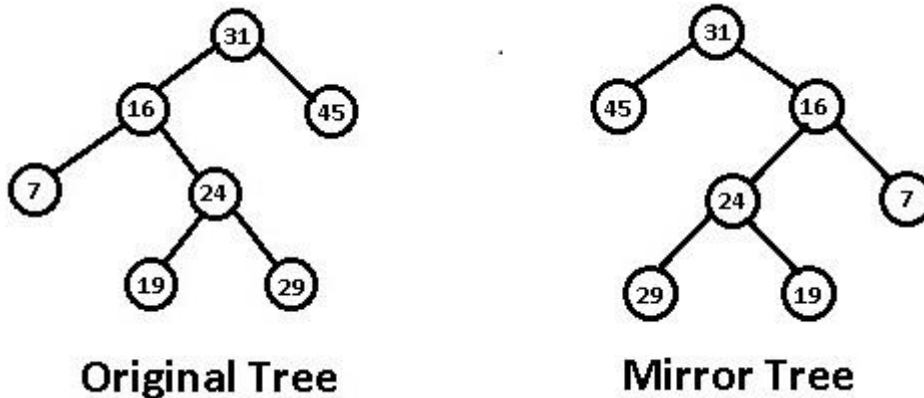
Deletion Operation in BST

In a binary search tree, the deletion operation is performed with $O(\log n)$ time complexity. Deleting a node from Binary search tree has following three cases...

- Case 1: Deleting a Leaf node (A node with no children)
- Case 2: Deleting a node with one child
- Case 3: Deleting a node with two children

Creation of Mirror Image of BST:

Following diagram shows a Binary Tree and its mirror image.



Suppose the root node is (X). We will construct a new root node (Y). Then $Y \rightarrow \text{left} = \text{right subtree of X}$ and $Y \rightarrow \text{right} = \text{left subtree of X}$. We will follow this procedure for all other nodes of the original tree recursively.

```
void mirror(struct node* node) {
    if (node==NULL) {
        return;
    }
    else {
        struct node* temp;

        // do the subtrees
        mirror(node->left);
        mirror(node->right);

        // swap the pointers in this node
        temp = node->left;
        node->left = node->right;
        node->right = temp;
    }
}
```

```
        node->right = temp;
    }
}
```

Binary Tree Traversals:

Tree Traversal algorithms can be classified broadly into two categories:

- Depth-First Search (DFS) Algorithms
- Breadth-First Search (BFS) Algorithms

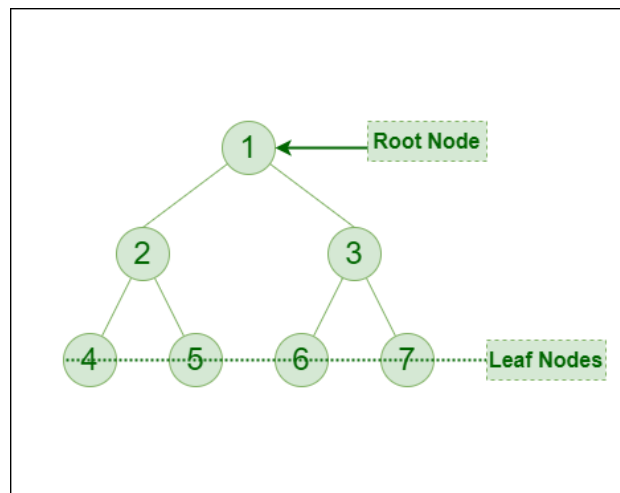
Tree Traversal using Depth-First Search (DFS) algorithm can be further classified into three categories:

- Preorder Traversal (current-left-right): Visit the current node before visiting any nodes inside the left or right subtrees. Here, the traversal is root – left child – right child. It means that the root node is traversed first then its left child and finally the right child.
- Inorder Traversal (left-current-right): Visit the current node after visiting all nodes inside the left subtree but before visiting any node within the right subtree. Here, the traversal is left child – root – right child. It means that the left child is traversed first then its root node and finally the right child.
- Postorder Traversal (left-right-current): Visit the current node after visiting all the nodes of the left and right subtrees. Here, the traversal is left child – right child – root. It means that the left child has traversed first then the right child and finally its root node.

Tree Traversal using Breadth-First Search (BFS) algorithm can be further classified into one category:

- Level Order Traversal: Visit nodes level-by-level and left-to-right fashion at the same level. Here, the traversal is level-wise. It means that the most left child has traversed first and then the other children of the same level from left to right have traversed.

Let us traverse the following tree with all four traversal methods:



Applications of Binary Tree:

1. In compilers, Expression Trees are used which is an application of binary trees.
2. Huffman coding trees are used in data compression algorithms.
3. Priority Queue is another application of binary tree that is used for searching maximum or minimum in $O(1)$ time complexity.
4. Represent hierarchical data.
5. Used in editing software like Microsoft Excel and spreadsheets.
6. Useful for indexing segmented at the database is useful in storing cache in the system,
7. Syntax trees are used for most famous compilers for programming like GCC, and AOCL to perform arithmetic operations.
8. For implementing priority queues.
9. Used to find elements in less time (binary search tree)
10. Used to enable fast memory allocation in computers.
11. Used to perform encoding and decoding operations.
12. Binary trees can be used to organize and retrieve information from large datasets, such as in inverted index and k-d trees.
13. Binary trees can be used to represent the decision-making process of computer-controlled characters in games, such as in decision trees.
14. Binary trees can be used to implement searching algorithms, such as in binary search trees which can be used to quickly find an element in a sorted list.
15. Binary trees can be used to implement sorting algorithms, such as in heap sort which uses a binary heap to sort elements efficiently.

PROGRAM:

// Tree traversal in C

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

```
struct node {
    int item;
    struct node* left;
    struct node* right;
};

// Inorder traversal
void inorderTraversal(struct node* root) {
    if (root == NULL) return;
    inorderTraversal(root->left);
    printf("%d ->", root->item);
    inorderTraversal(root->right);
}

// Preorder traversal
void preorderTraversal(struct node* root) {
    if (root == NULL) return;
    printf("%d ->", root->item);
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}

// Postorder traversal
void postorderTraversal(struct node* root) {
    if (root == NULL) return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    printf("%d ->", root->item);
}

// Create a new Node
struct node* createNode(value) {
    struct node* newNode = malloc(sizeof(struct node));
    newNode->item = value;
    newNode->left = NULL;
    newNode->right = NULL;

    return newNode;
}

// Insert on the left of the node
struct node* insertLeft(struct node* root, int value) {
    root->left = createNode(value);
    return root->left;
}
```

```
// Insert on the right of the node
struct node* insertRight(struct node* root, int value) {
    root->right = createNode(value);
    return root->right;
}

int main() {
    struct node* root = createNode(1);
    insertLeft(root, 2);
    insertRight(root, 3);
    insertLeft(root->left, 4);

    printf("Inorder traversal \n");
    inorderTraversal(root);


    printf("\nPreorder traversal \n");
    preorderTraversal(root);

    printf("\nPostorder traversal \n");
    postorderTraversal(root);
}
```

Output:

Output

```
/tmp/8AxvJ8puyG.o
Inorder traversal
4 ->2 ->1 ->3 ->
Preorder traversal
1 ->2 ->4 ->3 ->
Postorder traversal
4 ->2 ->3 ->1 ->
```

	MGM's COLLEGE OF ENGINEERING & TECHNOLOGY, NOIDA	LABORATORY MANUAL (DS)
	EXPERIMENT TITLE: WAP FOR IMPLEMENTSING GRAPHS	
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING		
EXPERIMENT NO.: MGMCOET/CSE/ST (CS/AIML)/DSL-PROGEAM-6	SEMESTER : III(ST)	PAGE:48

AIM: Implementing Graphs: Represent a graph using the Adjacency Matrix

THEORY:

Represent a graph using the Adjacency Matrix:

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by $G(V, E)$.

Representations of Graph

Here are the two most common ways to represent a graph :

1. Adjacency Matrix
2. Adjacency List

Adjacency Matrix

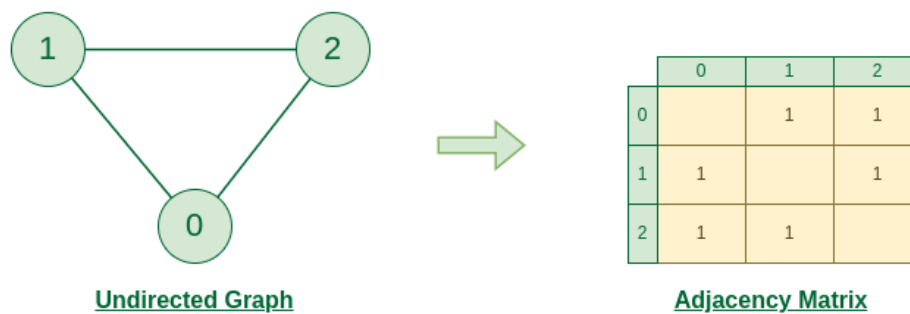
An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's). Let's assume there are n vertices in the graph So, create a 2D matrix $\text{adjMat}[n][n]$ having dimension $n \times n$.

If there is an edge from vertex i to j , mark $\text{adjMat}[i][j]$ as 1.

If there is no edge from vertex i to j , mark $\text{adjMat}[i][j]$ as 0

Representation of Undirected Graph to Adjacency Matrix:

The below figure shows an undirected graph. Initially, the entire Matrix is initialized to 0. If there is an edge from source to destination, we insert 1 to both cases ($\text{adjMat}[\text{destination}]$ and $\text{adjMat}[\text{source}]$) because we can go either way.

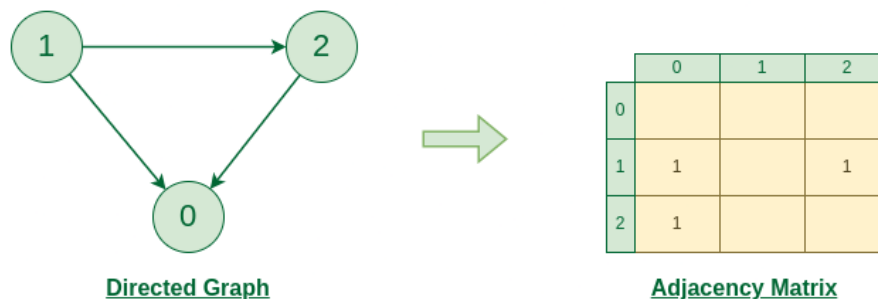


Graph Representation of Undirected graph to Adjacency Matrix

Fig: Undirected Graph to Adjacency Matrix

Representation of Directed Graph to Adjacency Matrix:

The below figure shows a directed graph. Initially, the entire Matrix is initialized to 0. If there is an edge from source to destination, we insert 1 for that particular **adjMat[destination]**.



Graph Representation of Directed graph to Adjacency Matrix

Fig: Directed Graph to Adjacency Matrix

PROGRAM:

```
#include <stdio.h>
int N, M;
void createAdjMatrix(int Adj[][N + 1],
int arr[][2])
{
for (int i = 0; i < N + 1; i++) {
for (int j = 0; j < N + 1; j++) {
Adj[i][j] = 0;
}
}
for (int i = 0; i < M; i++) {
```

```
        int x = arr[i][0];
        int y = arr[i][1];
        Adj[x][y] = 1;
        Adj[y][x] = 1;
    }
}
void printAdjMatrix(int Adj[][N + 1])
{
    for (int i = 1; i < N + 1; i++) {
        for (int j = 1; j < N + 1; j++) {
            printf("%d ", Adj[i][j]);
        }
        printf("\n");
    }
}
int main()
{
    N = 5;
    int arr[][2] = { { 1, 2 }, { 2, 3 }, { 4, 5 }, { 1, 5 } };
    M = sizeof(arr) / sizeof(arr[0]);
    int Adj[N + 1][N + 1];
    createAdjMatrix(Adj, arr);
    printf("The Adjacency Matrix is : \n");
    printAdjMatrix(Adj);
    return 0;
}
```

OUTPUT:**Output**

```
/tmp/EdoXyremzA.o
The Adjacency Matrix is :
0 1 0 0 1
1 0 1 0 0
0 1 0 0 0
0 0 0 0 1
1 0 0 1 0
```