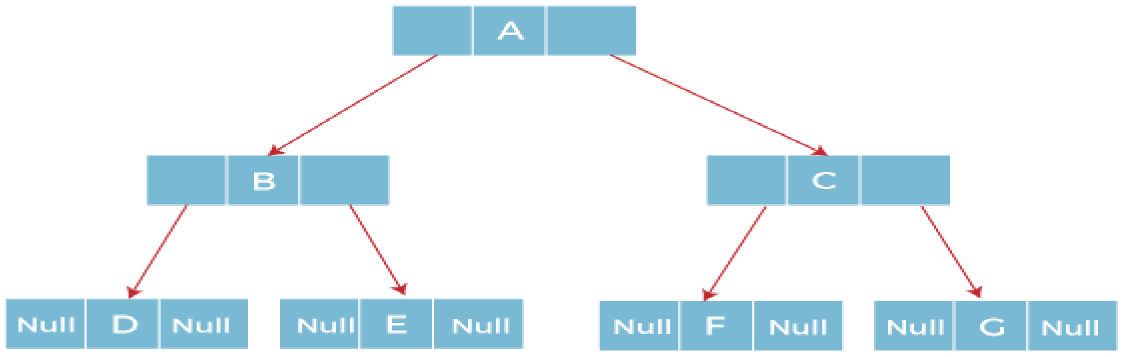
THREADED BINARY TREE (Explain Threaded Binary tree)

- In the linked representation of binary trees, more than one half of the link fields contain NULL values which results in wastage of storage space.
- If a binary tree consists of **n** nodes then **n+1** link fields contain NULL values.
- So in order to effectively manage the space, a method was devised by Perlis and Thornton in which the NULL links are replaced with special links known as threads.
- Such binary trees with threads are known as **threaded binary trees**.
- Each node in a threaded binary tree either contains a link to its child node or thread to other nodes in the tree.



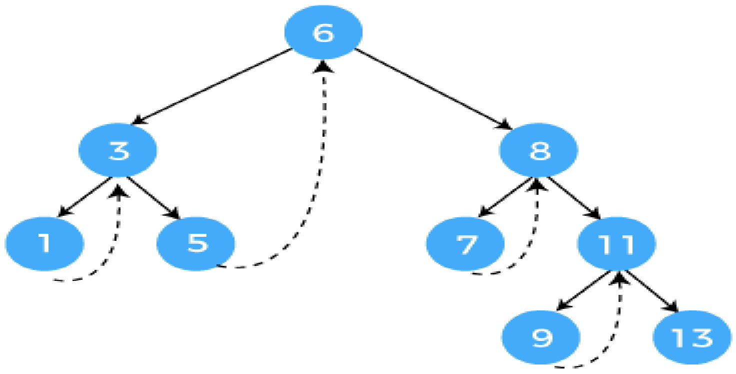
Threaded binary tree

Types of Threaded Binary Tree

There are two types of threaded Binary Tree:

- One-way threaded Binary Tree
- Two-way threaded Binary Tree

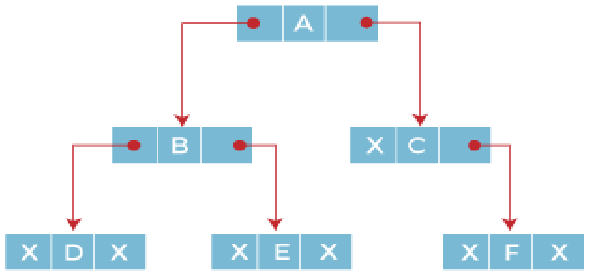
One-way threaded Binary trees:



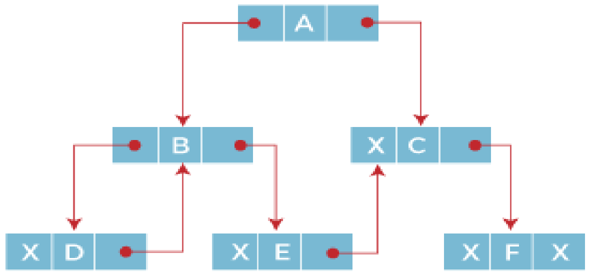
Single Threaded Binary Tree

- In one-way threaded binary trees, a thread will appear either in the right or left link field of a node.
- If it appears in the right link field of a node then it will point to the next node that will appear on performing in order traversal. Such trees are called **Right threaded binary trees**.
- If thread appears in the left field of a node then it will point to the nodes inorder predecessor. Such trees are called **Left threaded binary trees.**
- Left threaded binary trees are used less often as they don't yield the last advantages of right threaded binary trees.

- In one-way threaded binary trees, the right link field of last node and left link field of first node contains a NULL.
- In order to distinguish threads from normal links they are represented by dotted lines.



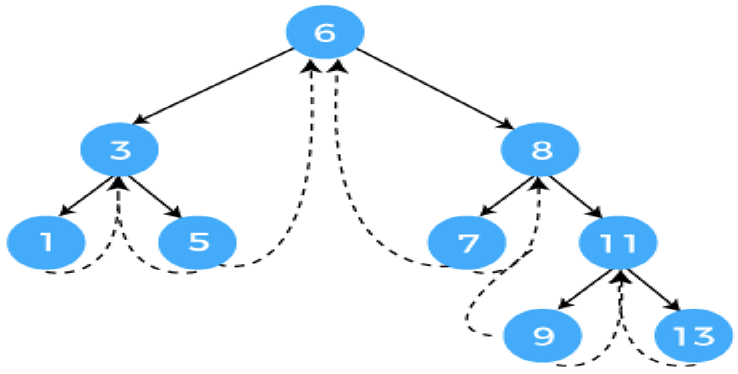
A binary tree (Inorder traversal - D, B, E, A, C, F)



A right - threaded binary tree

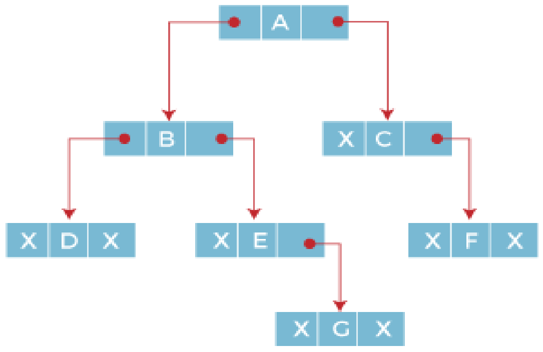
- The above figure shows the inorder traversal of this binary tree yields D, B, E, A, C, F. When this tree is represented as a right threaded binary tree, the right link field of leaf node D which contains a NULL value is replaced with a thread that points to node B which is the inorder successor of a node D. In the same way other nodes containing values in the right link field will contain NULL value.

Two-way threaded Binary Trees:

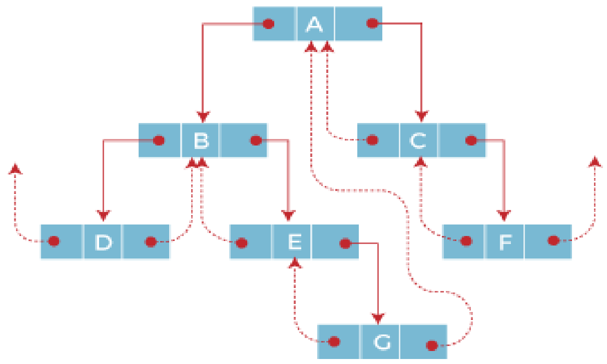


Double Threaded Binary Tree

- In two-way threaded Binary trees, the right link field of a node containing NULL values is replaced by a thread that points to nodes inorder successor and left field of a node containing NULL values is replaced by a thread that points to nodes inorder predecessor.

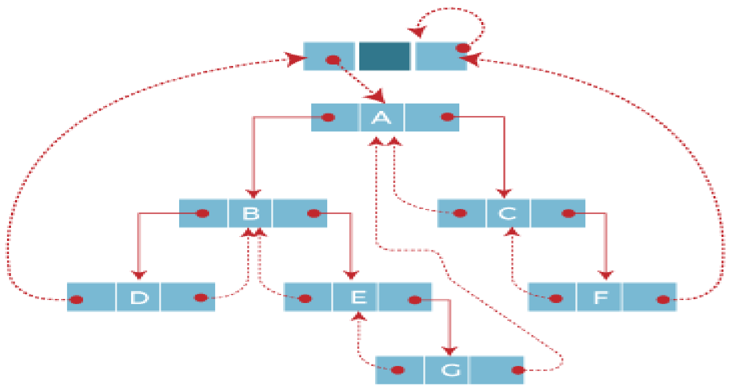


A binary tree (Inorder traversal - D, B, E, G, A, C, F)



A two - way threaded binary tree

- The above figure shows the inorder traversal of this binary tree yields D, B, E, G, A, C, F.
- If we consider the two-way threaded Binary tree, the node E whose left field contains NULL is replaced by a thread pointing to its inorder predecessor i.e. node B.
- Similarly, for node G whose right and left linked fields contain NULL values are replaced by threads such that right link field points to its inorder successor and left link field points to its inorder predecessor.
- In the same way, other nodes containing NULL values in their link fields are filled with threads.



Two-way threaded - tree with header node

- In the above figure of two-way threaded Binary tree, we noticed that no left thread is possible for the first node and no right thread is possible for the last node.
- This is because they don't have any inorder predecessor and successor respectively. This is indicated by threads pointing nowhere.
- So in order to maintain the uniformity of threads, we maintain a special node called the **header node**.
- The header node does not contain any data part and its left link field points to the root node and its right link field points to itself.
- If this header node is included in the two-way threaded Binary tree then this node becomes the inorder predecessor of the first node and inorder successor of the last node.
- Now threads of left link fields of the first node and right link fields of the last node will point to the header node.

Advantages of Threaded Binary Tree:

- In threaded binary tree, linear and fast traversal of nodes in the tree so there is no requirement of stack. If the stack is used then it consumes a lot of memory and time.
- It is more general as one can efficiently determine the successor and predecessor of any node by simply following the thread and links. It almost behaves like a circular linked list.

Disadvantages of Threaded Binary Tree:

- When implemented, the threaded binary tree needs to maintain the extra information for each node to indicate whether the link field of each node points to an ordinary node or the node's successor and predecessor.
- Insertion into and deletion from a threaded binary tree are more time consuming since both threads and ordinary links need to be maintained.

APPLIATIONS OF BINARY TREE(Write applications of binary tree)

Binary Tree is one of the most used Tree Data Structure and is used in real life Software systems. Following are the Applications of Binary Tree:

- Binary Tree is used to as the basic data structure in Microsoft Excel and spreadsheets in usual.
- Binary Tree is used to implement indexing of Segmented Database.
- Splay Tree (Binary Tree variant) is used in implemented efficient cache is hardware and software systems.
- Binary Space Partition Trees are used in Computer Graphics, Back face Culling, Collision detection, Ray Tracing and algorithms in rendering game graphics.
- Syntax Tree (Binary Tree with nodes as operations) are used to compute arithmetic expressions in compilers like GCC, AOCL and others.
- Binary Heap (Binary Tree variant of Heap) is used to implement Priority Queue efficiently which in turn is used in Heap Sort Algorithm.
- Binary Search Tree is used to search elements efficiently and used as a collision handling technique in Hash Map implementations.
- Balanced Binary Search Tree is used to represent memory to enable fast memory allocation.
- Huffman Tree (Binary Tree variant) is used internally in a Greedy Algorithm for Data Compression known as Huffman Encoding and Decoding.
- Merkle Tree/ Hash Tree (Binary Tree variant) is used in Blockchain implementations and p2p programs requiring signatures.
- Binary Tries (Tries with 2 child) is used to represent a routing data which vacillate efficient traversal.
- Morse code is used to encode data and uses a Binary Tree in its representation.
- Goldreich, Goldwasser and Micali (GGM) Tree (Binary Tree variant) is used compute pseudorandom functions using an arbitrary pseudorandom generator.
- Scapegoat tree (a self-balancing Binary Search Tree) is used in implementing Paul-Carole games to model a faulty search process.
- Treap (radomized Binary Search Tree)

Binary Tree is the most widely used Data Structure because:

- Binary Tree is the most simpliest and efficient data structure to be used in most Software Systems. It is the properties of Binary Tree that makes it so widely used.
- N-ary Tree which is the generalization of Binary Tree is complex to implement and is rarely a better fit.
- Binary Tree can be implemented as an array using ideas of Binary Heap. Hence, the ideas of OOP (Object Oriented Programming) is not necessary for a safe implementation.
- There is a wide range of variants of Binary Tree which makes it very likely to find a suitable variant for a specific problem. For example, if we want a Data Structure where recently accessed elements are closer to the beginning of the data structure so that access is fast, then we have a variant of Binary Tree known as Splay Tree.

SEARCH TECHNIQUES (what is searching?)

- Searching in data-structure refers to the process of finding a desired element in set of items. The desired element is called "target".

- The set of items to be searched in, can be any data-structure like − list, array, linked-list, tree or graph.
- Search refers to locating a desired element of specified properties in a collection of items.
- The process of locating the target data is known as Searching. If the search key is unique and if it determines a record uniquely, it is called a primary key.
- Sorted list of data makes searching easier and faster. The data structures linked list, array, tree and graph are used for storing data. The data may be stored on a secondary storage or primary storage area.
- If the search is applied on secondary storage data is called as "external searching".
- If the search is applied on data of primary storage is called as "internal searching".
- Internal searching is faster than external searching. Commonly used search methods are
   1. Linear search 2. Binary search

LINEAR SEARCH (Explain Linear search method with example)

- 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.
- 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).**

Algorithm

Linear_Search(A, n, key)

```
1. Let A is the array of n elements, and key if the value to be searched
2. Set i=0, flag=0
3. Compare  A[i] and key as
    If(A[i]=key) then
       Set flag=1 and goto step  6
4. Move to next data element      i=i+1
5. If(i<n) then goto step 3
6. If (flag=1)   return   i
   else   return  -1
7. stop
```

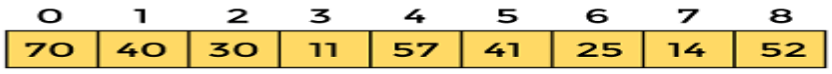
The steps used in the implementation of Linear Search are listed as follows -

- 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.**

```
int  Linear_Search(int A[], int n, int key)
{      int  i, flag=0;
           for(i=0;i<n;i++)
              if(key==A[i])
             {  flag=1;
                break;    }
           if(flag==1)
               return  I;
           else
               return  -1
       }
```

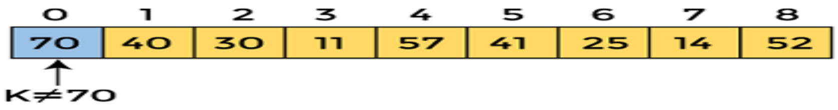
Working of Linear search

- 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 -

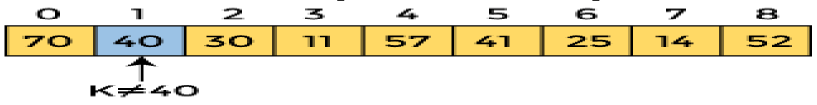


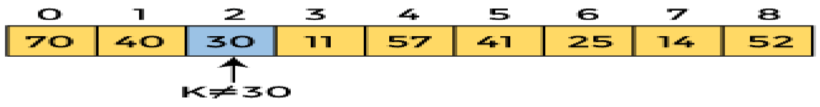
Let the element to be searched is **K = 41**

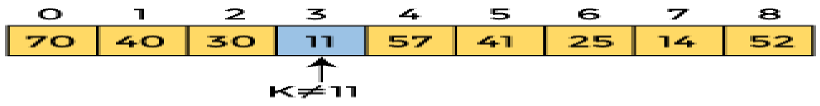
Now, start from the first element and compare **K** with each element of the array.

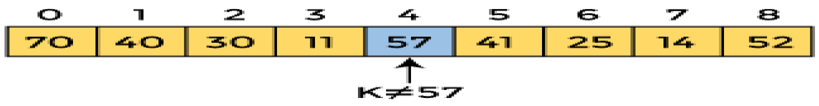


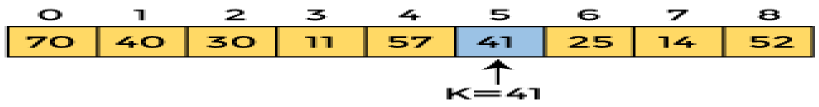
The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.











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

Linear Search complexity

Time Complexity

Case	**Time Complexity**

Best Case	O(1)
Average Case	O((n+1)/2) for successful search, for unsuccessful search O(n)
Worst Case	O(n)

- **Best Case Complexity -** In Linear search, best case occurs when the element we are finding is at the first position of the array. The best-case time complexity of linear search is **O(1).**
- **Average Case Complexity -** The average case time complexity of linear search is **O((n+1)/2).** For unsuccessful search **O(n)**
- **Worst Case Complexity -** In Linear search, the worst case occurs when the element we are looking is present at the end of the array. The worst-case in linear search could be when the target element is not present in the given array, and we have to traverse the entire array. The worst-case time complexity of linear search is **O(n).**
- The time complexity of linear search is **O(n)** because every element in the array is compared only once.

Space Complexity of linear search is O(1).

BINARY SEARCH(Explain Binary search algorithm with example)

- To search an element using binary search in a list, the list must be in Ascending order.
- Divide the list to be searched every time into two lists and search is done in only one of the list is called as binary search.
- In binary search, to search for a key, it is first compared with the middle element of the list and if it is matched, the search is successful.
- Else if the key is less than the middle element, then the search will continue in the first half of the list.
- Otherwise the key is searched in the second half of the list.
- The same process is repeated for one of the halves of the list till the list is reduced to size one.
- Binary search is implemented using recursive and non-recursive concepts.

Algorithm of Binary Search

Binary_Search(A, n, key)

```
1. Let n be the size of the list A, Key if the element to be searched
2. Let low=0, high=n-1
3. If low<=high then
        mid=(low+high)/2
     else  goto step 6
4. If(A[mid]=key) then
      Return mid,  goto step 7
   else
      if(key<A[mid]) then
        high=mid-1
     else
        low=mid+1
5. Goto step  3
```

6. return -1
7. Stop

Advantages:

1. Suitable for sorted data.
2. Efficient for large lists.
3. Suitable for storage structures that support direct access to data.
4. Time complexity is O(log $_2$(n))

Disadvantages

1. Not applicable for unsorted data
2. Not suitable for magnetic tapes and linked lists
3. Inefficient for small lists.

Binary Search complexity

Time Complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

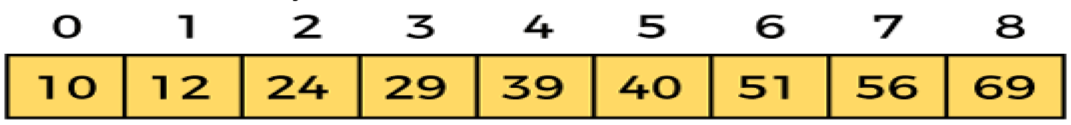
- **Best Case Complexity -** In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is **O(1).**
- **Average Case Complexity -** The average case time complexity of Binary search is **O(logn).**
- **Worst Case Complexity -** In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is **O(logn).**

Space Complexity: The space complexity of binary search is **O(1)**.

Example

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

Let the elements of array are -



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

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

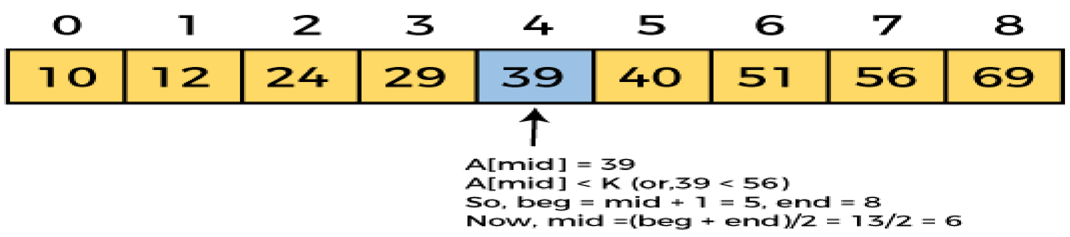
mid = (beg+ end)/2 **(here beg means low, end means high)**

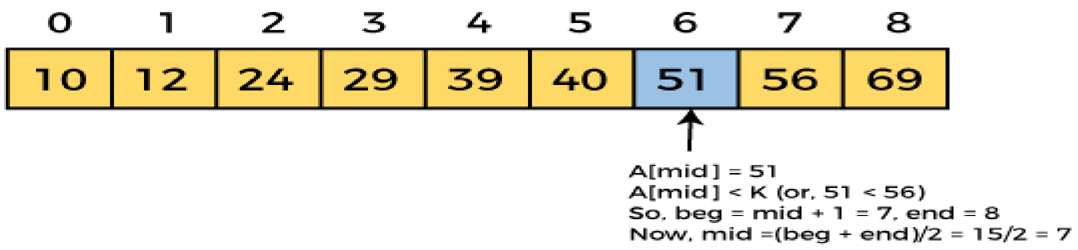
So, in the given array -

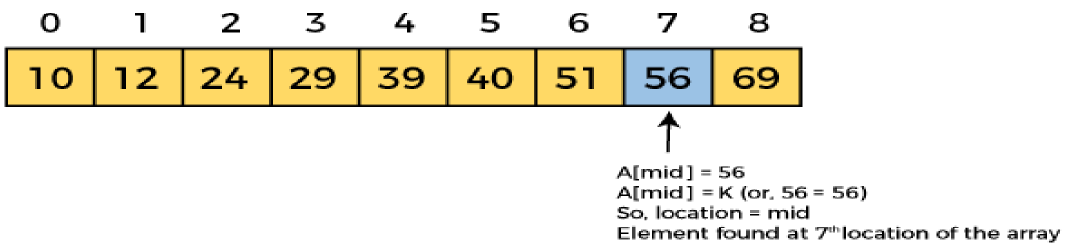
beg= 0

end = 8

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







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

SORTING (What is sorting? Write about types of sorting?)

- Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order
- For example, consider an array A = {A1, A2, A3, A4, ?? An }, the array is called to be in ascending order if element of A are arranged like A1 > A2 > A3 > A4 > A5 > … > An .

Consider an array;

int A[10] = { 5, 4, 10, 2, 30, 45, 34, 14, 18, 9)

The Array sorted in ascending order will be given as;

A[] = { 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 }

There are many techniques by using which, sorting can be performed.

Sorting Algorithms

SN	Sorting Algorithms	Description

1	Bubble Sort	It is the simplest sort method which performs sorting by repeatedly moving the largest element to the highest index of the array. It comprises of comparing each element to its adjacent element and replace them accordingly.
2	Heap Sort	In the heap sort, Min heap or max heap is maintained from the array elements deending upon the choice and the elements are sorted by deleting the root element of the heap.
3	Insertion Sort	As the name suggests, insertion sort inserts each element of the array to its proper place. It is a very simple sort method which is used to arrange the deck of cards while playing bridge.
4	Merge Sort	Merge sort follows divide and conquer approach in which, the list is first divided into the sets of equal elements and then each half of the list is sorted by using merge sort. The sorted list is combined again to form an elementary sorted array.
5	Quick Sort	Quick sort is the most optimized sort algorithms which performs sorting in O(n log n) comparisons. Like Merge sort, quick sort also work by using divide and conquer approach.
6	Selection Sort	Selection sort finds the smallest element in the array and place it on the first place on the list, then it finds the second smallest element in the array and place it on the second place. This process continues until all the elements are moved to their correct ordering. It carries running time O(n2) which is worst than insertion sort.

BUBBLE SORT (Explain Bubble sort)

- Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order.
- It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water.
- Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.
- Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world.
- It is not suitable for large data sets. The average and worst-case complexity of Bubble sort is **$O(n^2)$,** where **n** is a number of items.
- Bubble sort is majorly used where -
  - complexity does not matter
  - simple and shortcode is preferred

Algorithm

```
1.let A be the array to be sorted of size n
2. for i=1 to n-1
  For j=0 to n-i
```

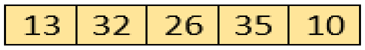
```
 If A[j]>A[j+1] then
  Swap  A[j], A[j+1];
3.stop
```

Working of Bubble sort Algorithm

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

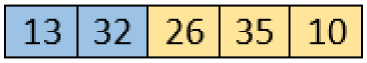
To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is **O(n^2).**

Let the elements of array are -

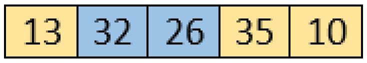


First Pass

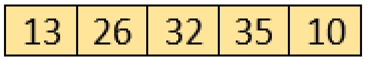
Sorting will start from the initial two elements. Let compare them to check which is greater.



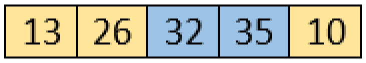
Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.



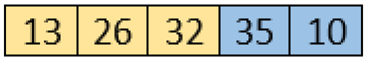
Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -



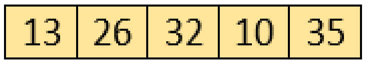
Now, compare 32 and 35.



Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.
Now, the comparison will be in between 35 and 10.



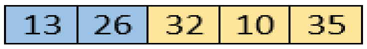
Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -

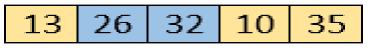


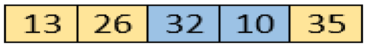
Now, move to the second iteration.

Second Pass

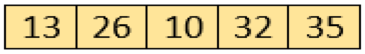
The same process will be followed for second iteration.

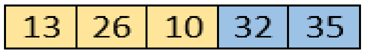






Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -

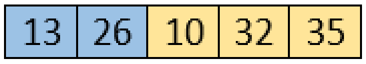


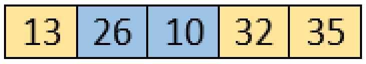


Now, move to the third iteration.

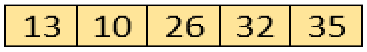
Third Pass

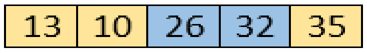
The same process will be followed for third iteration.

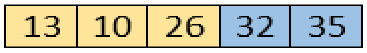




Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -



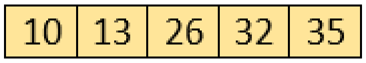




Now, move to the fourth iteration.

Fourth pass

Similarly, after the fourth iteration, the array will be -



Hence, there is no swapping required, so the array is completely sorted.

Bubble sort complexity

Time Complexity

Case	**Time Complexity**
Best Case	$O(n)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **O(n).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is **$O(n^2)$.**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is **$O(n^2)$.**

Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of bubble sort is O(1). It is because, in bubble sort, an extra variable is required for swapping.
- The space complexity of optimized bubble sort is O(2). It is because two extra variables are required in optimized bubble sort.

Implementation of Bubble sort

```
void Insertion(int A[], int n)
{
   int i, j, t;
   for (i = 1; i < n-1; i++)
   {
       for (j = 0;  j < n-i;  j++)
         if (A[j] > A[j+1])
          { t=A[j];
             A[j]=A[j+1];
          A[j+1]=t;
          }
     }
}
```

INSERTION SORT (Explain Insertion sort)

- The working procedure of insertion sort is also simple.
- 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.

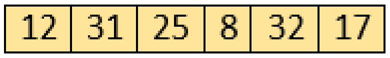
Algorithm

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.
Step2 - Pick the next element, and store it separately in a **key.**
Step3 - 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.

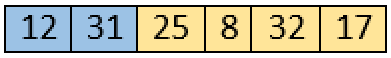
```
1.let A is the array of n elements
2.for i=1 to n
 Begin
  Ele=A[i];
  j=i;
While(j>0 and A[j-1]>ele)
Begin
   A[j]=A[j-1]
   J=j-1
End
A[j]=ele
End
3.stop
```

Working of Insertion sort Algorithm

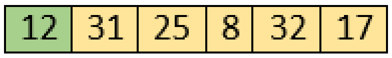
To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example. Let the elements of array are -



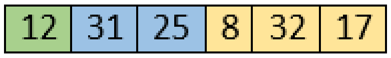
Initially, the first two elements are compared in insertion sort.

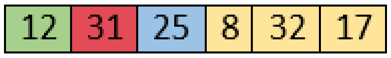


Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.

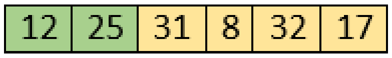


Now, move to the next two elements and compare them.

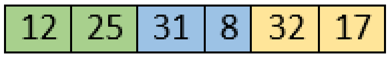


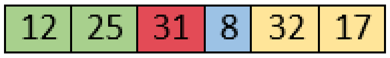


Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.
For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

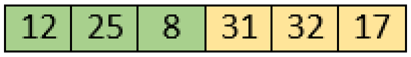


Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

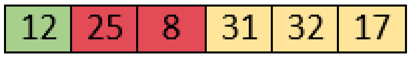




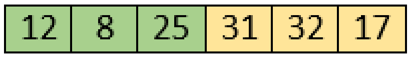
Both 31 and 8 are not sorted. So, swap them.



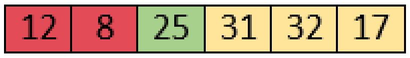
After swapping, elements 25 and 8 are unsorted.



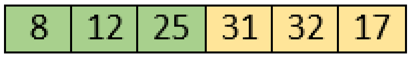
So, swap them.



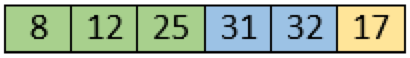
Now, elements 12 and 8 are unsorted.



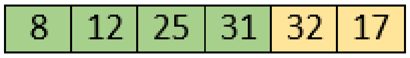
So, swap them too.



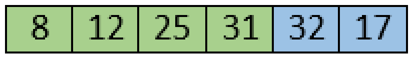
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



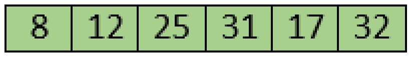
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

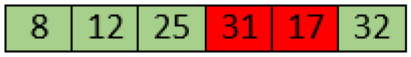


Move to the next elements that are 32 and 17.

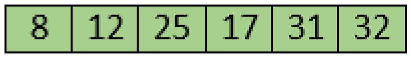


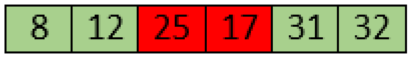
17 is smaller than 32. So, swap them.



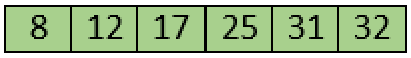


Swapping makes 31 and 17 unsorted. So, swap them too.





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



Now, the array is completely sorted.

Insertion sort complexity

Time Complexity

Case	**Time Complexity**
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is **$O(n^2)$**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **$O(n^2)$**.

Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of insertion sort is O(1). It is because, in insertion sort, an extra variable is required for swapping.

Implementation of insertion sort

```
void  Insertion_sort(int A[], int n)
{
  int  i,  j,ele;
  for(i=1;i<n;i++)
  {  ele=A[i];
      j=i;
     while(j>0 && A[j-1]>ele)
     {
        A[j]=A[j-1];
        j=j-1;
      }
   A[j]=ele;
}
}
```

SELECTION SORT (Explain Selection sort)

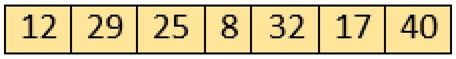
- In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.
- It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, 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

Algorithm

```
SELECTION SORT(arr, n)
    Step 1: Repeat Steps 2 and 3 for i = 0 to n-1
    Step 2: CALL SMALLEST(arr, i, n, pos)
    Step 3: SWAP arr[i] with arr[pos]
    Step 4: EXIT
SMALLEST (arr, i, n, pos)
    Step 1: [INITIALIZE] SET SMALL = arr[i]
    Step 2: [INITIALIZE] SET pos = i
    Step 3: Repeat for j = i+1 to n
    if (SMALL > arr[j])
        SET SMALL = arr[j]
    SET pos = j
    Step 4: RETURN pos
```

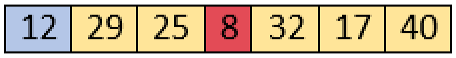
Working of Selection sort Algorithm

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 -

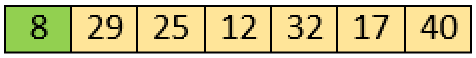


Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

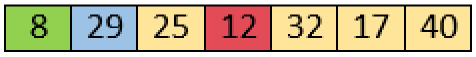
At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.



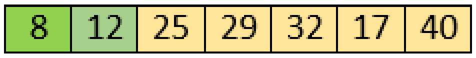
So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.



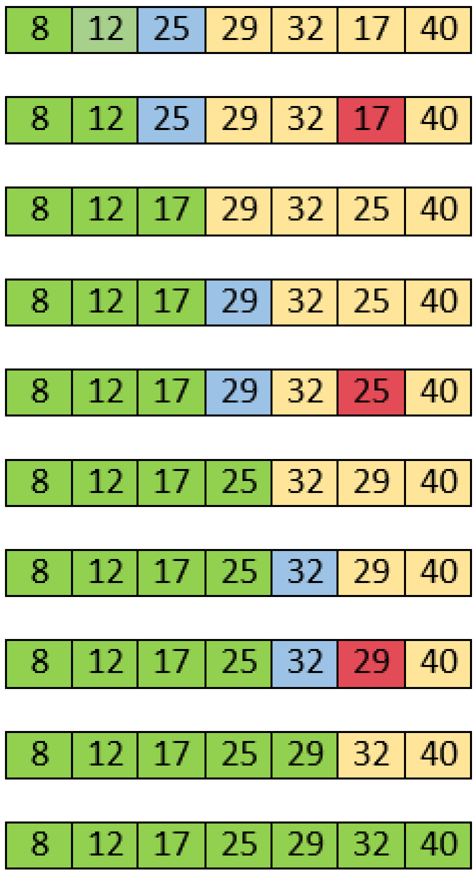
For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.



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.



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.

Selection sort complexity

Time Complexity

Case	**Time Complexity**
Best Case	O(n^2)
Average Case	O(n^2)
Worst Case	O(n^2)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sort is **O(n^2)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is **O(n^2)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of selection sort is **O(n^2)**.

Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of selection sort is O(1). It is because, in selection sort, an extra variable is required for swapping.

Implementation of selection sort

```
void selection(int A[], int n)
{
   int i, j, small;
   for (i = 0; i < n-1; i++)
   {
      small = i;
      for (j = i+1; j < n; j++)
      if (A[j] < A[small])
         small = j;
if(small!=i)
{
   int temp = A[small];
   A[small] = A[i];
   A[i] = temp;
   }
}  }
```

QUICK SORT (Explain Quick sort with algorithm and example)

- The working procedure of Quicksort is also simple.
- 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.

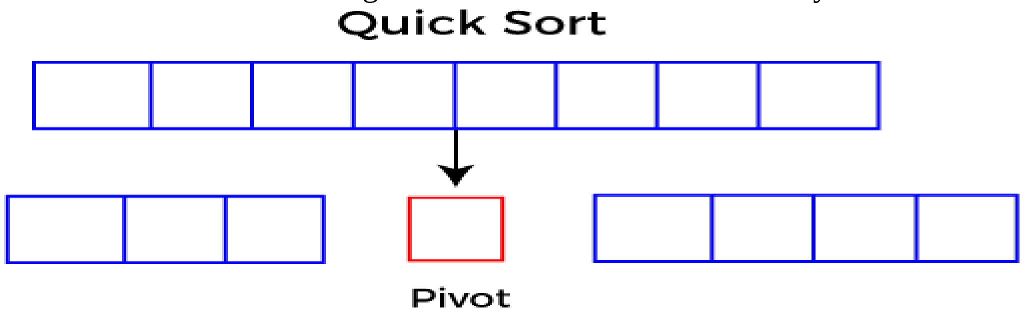
Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

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:

```
QUICKSORT (array A, start, end)
{
  if (start < end)
 {
 p = partition(A, start, end)
 QUICKSORT (A, start, p - 1)
 QUICKSORT (A, p + 1, end)
 }
}
```

Partition Algorithm:

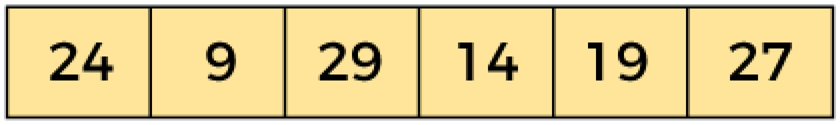
The partition algorithm rearranges the sub-arrays in a place.

```
PARTITION (array A, start, end)
{
  pivot = A[end]
  i= start-1
  for j= start to end -1
 {
  do if (A[j] < pivot) {
     then i= i + 1
  swap A[i] with A[j]
   }}
  swap A[i+1] with A[end]
  return i+1
```

}

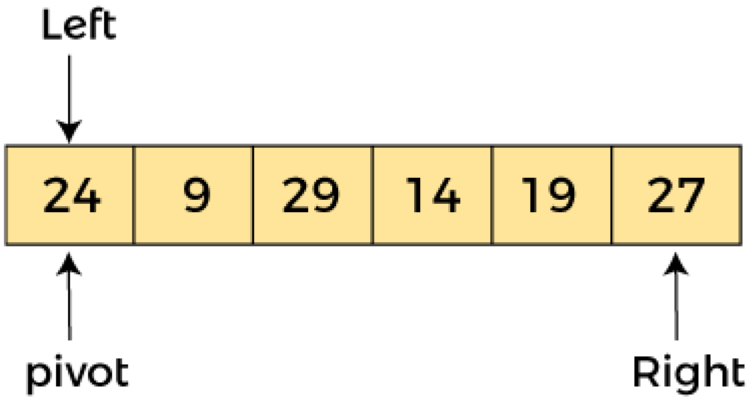
Working of Quick Sort Algorithm

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 -

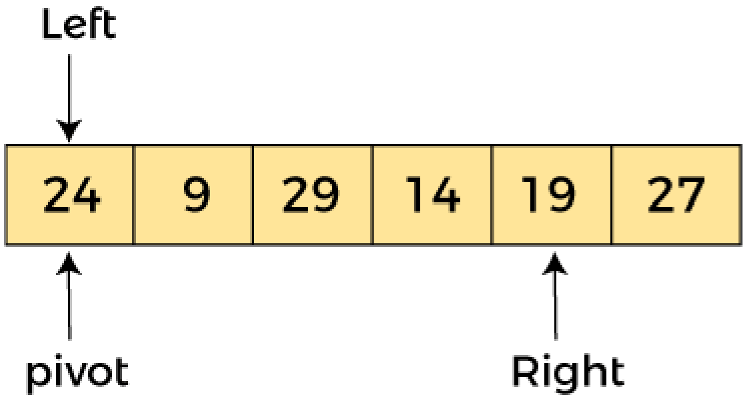


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

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

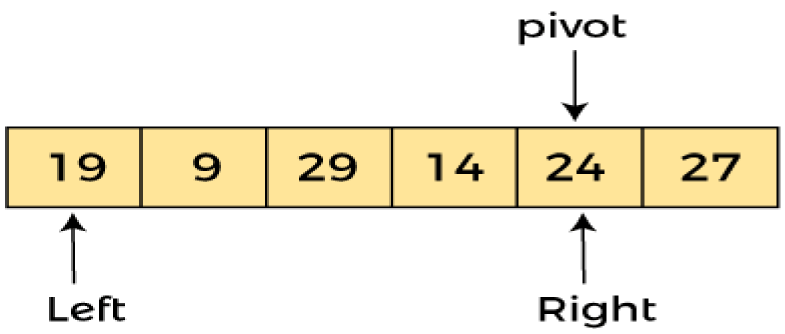


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



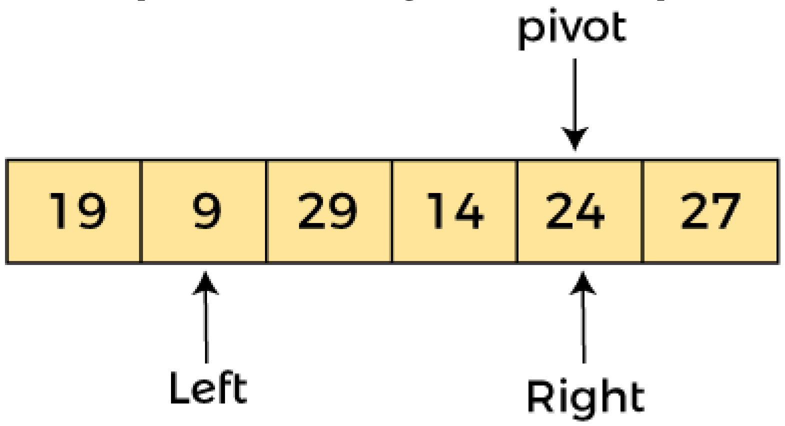
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

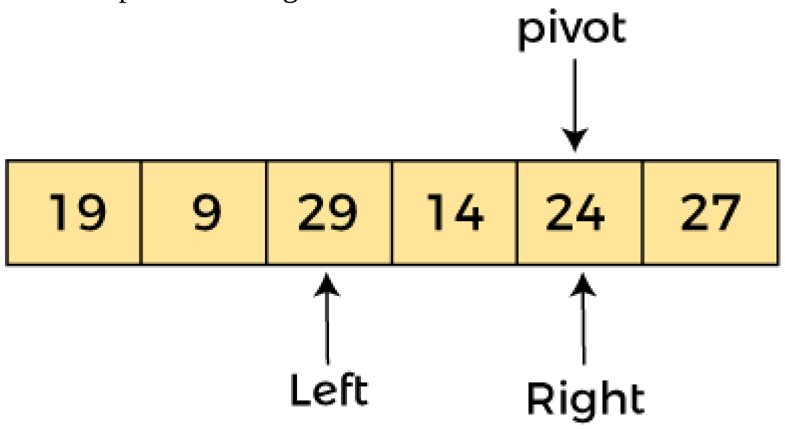


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

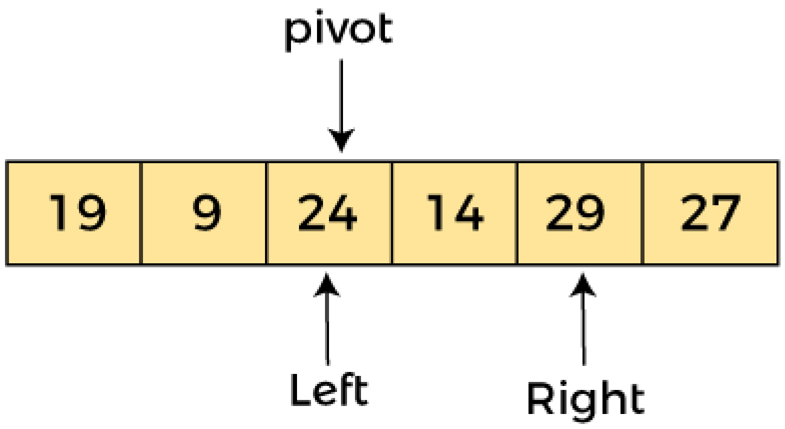
As a[pivot] > a[left], so algorithm moves one position to right as -



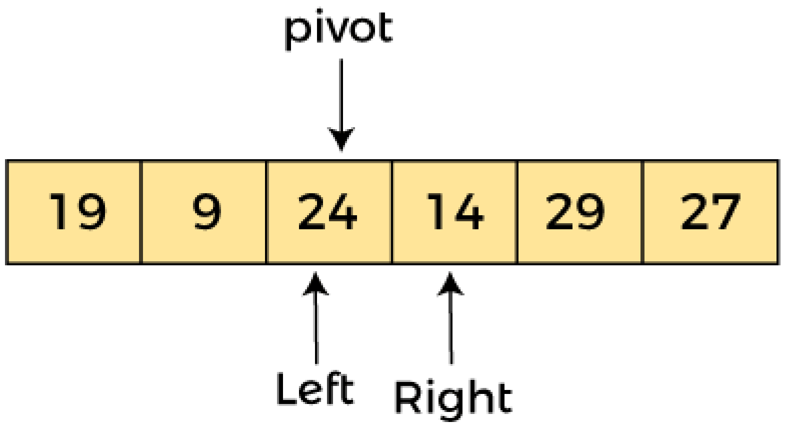
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



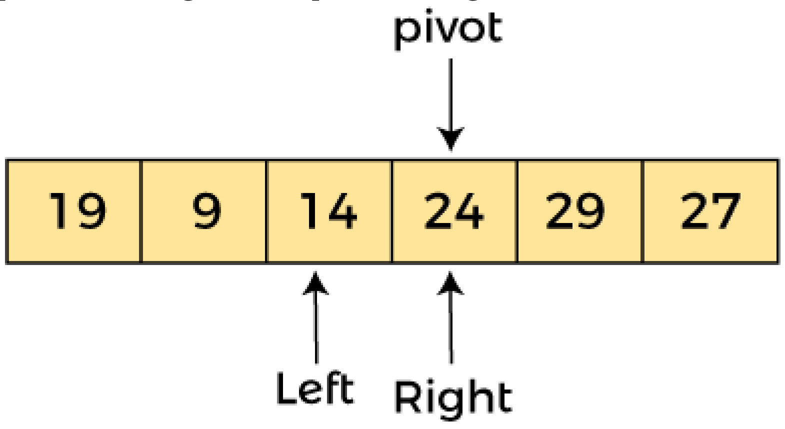
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



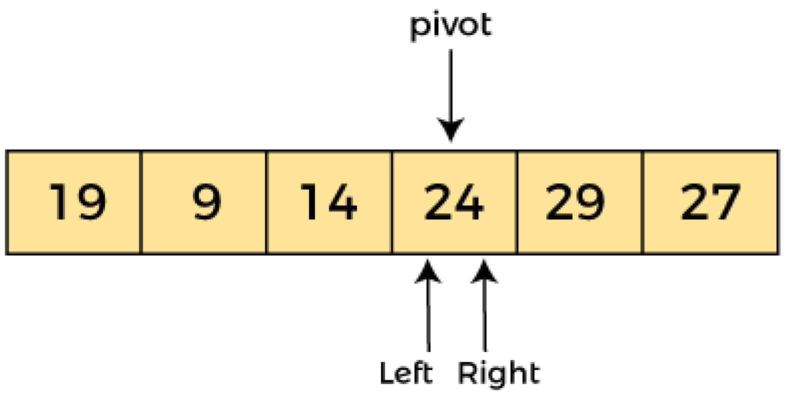
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



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



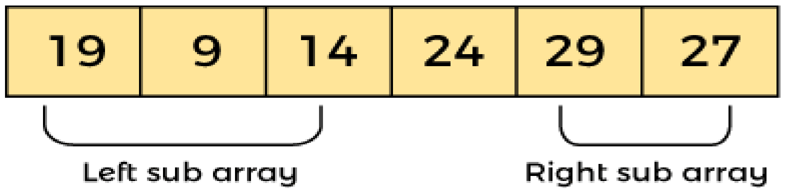
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



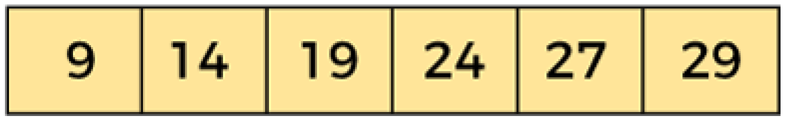
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

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 -



Quicksort complexity

Time Complexity

Case	**Time Complexity**
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	$O(n^2)$

- **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n*logn)**.

- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n*logn)**.
- **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **O(n^2)**.
- Though the worst-case complexity of quicksort is more than other sorting algorithms such as **Merge sort** and **Heap sort**, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.

Space Complexity

Space Complexity	O(n*logn)
Stable	NO

- The space complexity of quicksort is O(n*logn).

MERGE SORT (Explain merge sort technique with algorithm and example)

- Merge sort is the sorting technique that follows the divide and conquer approach.
- Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements.
- It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.
- The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process.
- The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

Algorithm

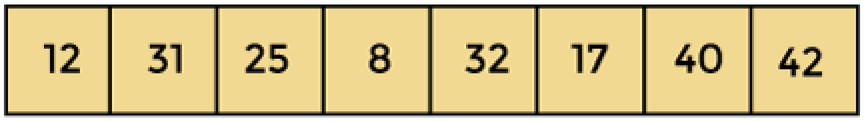
In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.

```
MERGE_SORT(arr, beg, end)
1. if beg < end
2. set mid = (beg + end)/2
3. MERGE_SORT(arr, beg, mid)
4. MERGE_SORT(arr, mid + 1, end)
5. MERGE (arr, beg, mid, end)
6. end of if
END MERGE_SORT
```

- The important part of the merge sort is the **MERGE** function.
- This function performs the merging of two sorted sub-arrays that are **A[beg…mid]** and **A[mid+1…end]**, to build one sorted array **A[beg…end]**.
- So, the inputs of the **MERGE** function are **A[], beg, mid,** and **end**.
- The implementation of the **MERGE** function is given as follows -

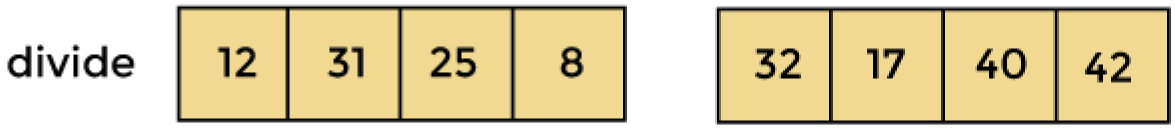
Working of Merge sort Algorithm

To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example. Let the elements of array are -

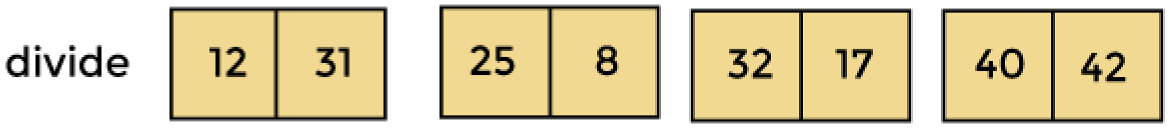


According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

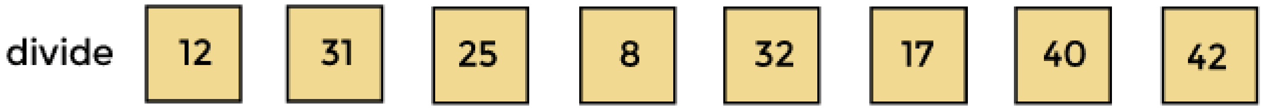
As there are eight elements in the given array, so it is divided into two arrays of size 4.



Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.



Now, again divide these arrays to get the atomic value that cannot be further divided.

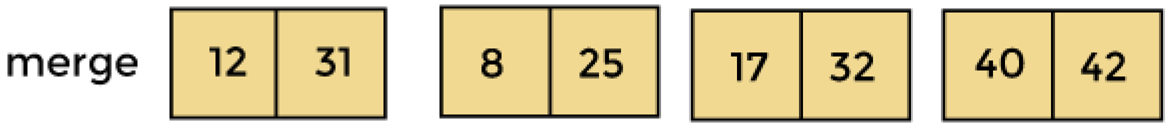


Now, combine them in the same manner they were broken.

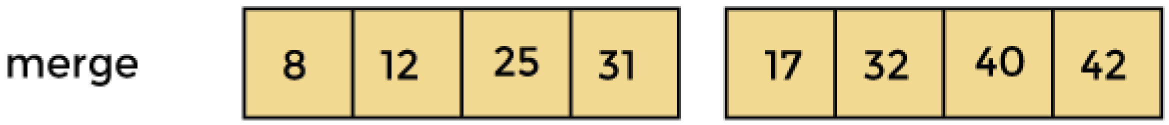
In combining, first compare the element of each array and then combine them into another array in sorted order.

So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25.

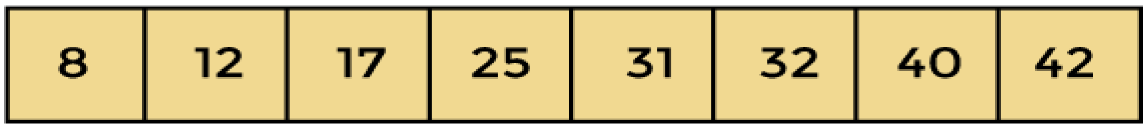
Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



Now, the array is completely sorted.

<u>Merge sort complexity</u>

Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n*logn)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n*logn)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **O(n*logn)**.

Space Complexity

Space Complexity	O(n)
Stable	YES

- The space complexity of merge sort is O(n). It is because, in merge sort, an extra variable is required for swapping.

SYMBOL TABLE (Write about Symbol table)

- Symbol table is an important data structure used in a compiler.
- Symbol table is used to store the information about the occurrence of various entities such as objects, classes, variable name, interface, function name etc. it is used by both the analysis and synthesis phases.
- The symbol table used for following purposes:
    - It is used to store the name of all entities in a structured form at one place.
    - It is used to verify if a variable has been declared.
    - It is used to determine the scope of a name.
    - It is used to implement type checking by verifying assignments and expressions in the source code are semantically correct.
- A symbol table can either be linear or a hash table. Using the following format, it maintains the entry for each name.

  <symbol name, type, attribute>
- For example, suppose a variable store the information about the following variable declaration: **static int** salary; then, it stores an entry in the following format:

  <salary, **int**, **static**>
- The clause attribute contains the entries related to the name.

Implementation

The symbol table can be implemented in the unordered list if the compiler is used to handle the small amount of data.

A symbol table can be implemented in one of the following techniques:

- Linear (sorted or unsorted) list
- Hash table
- Binary search tree

Symbol table are mostly implemented as hash table.

Operations

The symbol table provides the following operations:

Insert ()

- Insert () operation is more frequently used in the analysis phase when the tokens are identified and names are stored in the table.
- The insert() operation is used to insert the information in the symbol table like the unique name occurring in the source code.
- In the source code, the attribute for a symbol is the information associated with that symbol. The information contains the state, value, type and scope about the symbol.
- The insert () function takes the symbol and its value in the form of argument.

For example:

int x; Should be processed by the compiler as: insert (x, **int**)

lookup()

- In the symbol table, lookup() operation is used to search a name. It is used to determine:
    - The existence of symbol in the table.
    - The declaration of the symbol before it is used.
    - Check whether the name is used in the scope.
    - Initialization of the symbol.
    - Checking whether the name is declared multiple times.

The basic format of lookup() function is as follows:

lookup (symbol) :This format is varies according to the programming language.

Data structure for symbol table

- A compiler contains two type of symbol table: global symbol table and scope symbol table.
- Global symbol table can be accessed by all the procedures and scope symbol table.

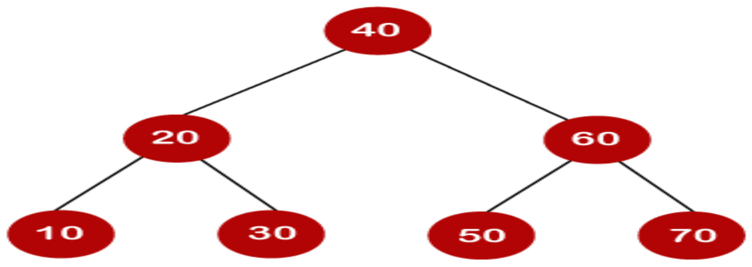
- Data structure hierarchy of symbol table is stored in the semantic analyzer. If you want to search the name in the symbol table then you can search it using the following algorithm:
  - First a symbol is searched in the current symbol table.
  - If the name is found then search is completed else the name will be searched in the symbol table of parent until,
  - The name is found or global symbol is searched.

OPTIMAL BINARY SEARCH TREE

- As we know that in binary search tree, the nodes in the left subtree have lesser value than the root node and the nodes in the right subtree have greater value than the root node.
- We know the key values of each node in the tree, and we also know the frequencies of each node in terms of searching means how much time is required to search a node.
- The frequency and key-value determine the overall cost of searching a node.
- The cost of searching is a very important factor in various applications. The overall cost of searching a node should be less.
- The time required to search a node in BST is more than the balanced binary search tree as a balanced binary search tree contains a lesser number of levels than the BST.
- There is one way that can reduce the cost of a binary search tree is known as an **optimal binary search tree**.

Let's understand through an example.

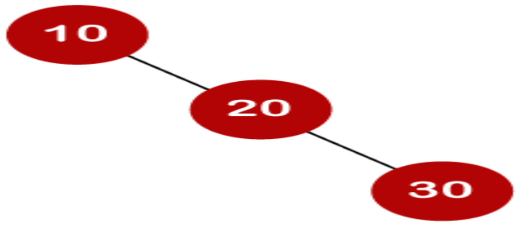
If the keys are 10, 20, 30, 40, 50, 60, 70



- In the above tree, all the nodes on the left subtree are smaller than the value of the root node, and all the nodes on the right subtree are larger than the value of the root node.
- The maximum time required to search a node is equal to the minimum height of the tree, equal to logn.
- Now we will see how many binary search trees can be made from the given number of keys.
- For example: 10, 20, 30 are the keys, and the following are the binary search trees that can be made out from these keys.
- The Formula for calculating the number of trees:

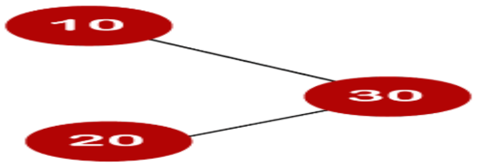
$$\frac{{}^{2n}C_n}{n+1}$$

- When we use the above formula, then it is found that total 5 number of trees can be created.
- The cost required for searching an element depends on the comparisons to be made to search an element.
- Now, we will calculate the average cost of time of the above binary search trees.



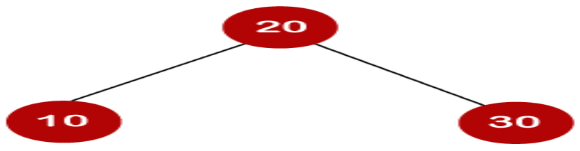
- In the above tree, total number of 3 comparisons can be made. The average number of comparisons can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



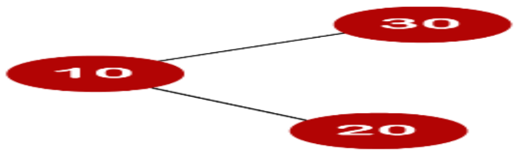
- In the above tree, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



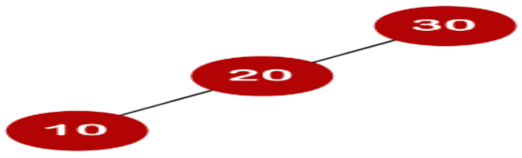
- In the above tree, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+2}{3} = 5/3$$



- In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



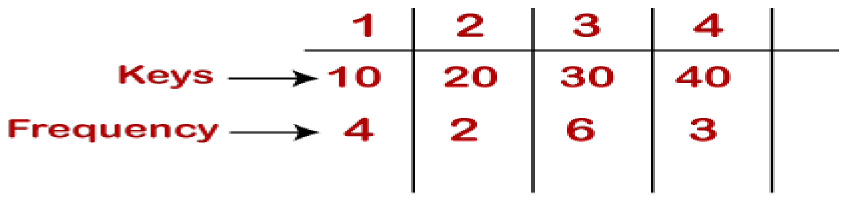
- In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$

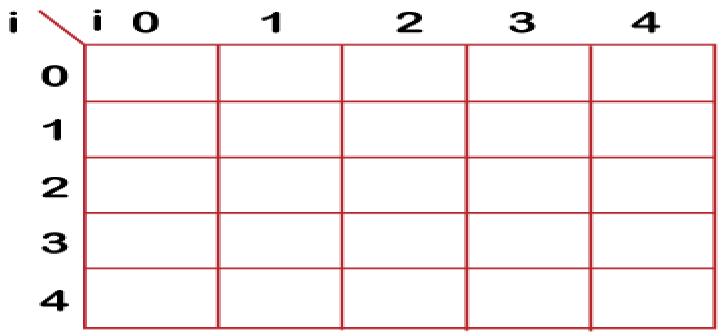
- In the third case, the number of comparisons is less because the height of the tree is less, so it's a balanced binary search tree.
- Till now, we read about the height-balanced binary search tree. To find the optimal binary search tree, we will determine the frequency of searching a key.
- Let's assume that frequencies associated with the keys 10, 20, 30 are 3, 2, 5.
- The above trees have different frequencies. The tree with the lowest frequency would be considered the optimal binary search tree. The tree with the frequency 17 is the lowest, so it would be considered as the optimal binary search tree.

<u>Dynamic Approach</u>

Consider the below table, which contains the keys and frequencies.



	1	2	3	4
Keys →	10	20	30	40
Frequency →	4	2	6	3



i \ i	0	1	2	3	4
0					
1					
2					
3					
4					

First, we will calculate the values where j-i is equal to zero.

When i=0, j=0, then j-i = 0

When i = 1, j=1, then j-i = 0

When i = 2, j=2, then j-i = 0

When i = 3, j=3, then j-i = 0

When i = 4, j=4, then j-i = 0

Therefore, c[0, 0] = 0, c[1 , 1] = 0, c[2,2] = 0, c[3,3] = 0, c[4,4] = 0

Now we will calculate the values where j-i equal to 1.

When j=1, i=0 then j-i = 1

When j=2, i=1 then j-i = 1

When j=3, i=2 then j-i = 1

When j=4, i=3 then j-i = 1

Now to calculate the cost, we will consider only the jth value.

The cost of c[0,1] is 4 (The key is 10, and the cost corresponding to key 10 is 4).

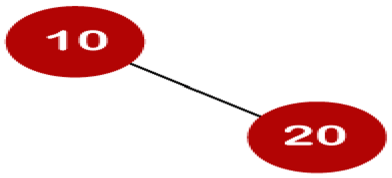
The cost of c[1,2] is 2 (The key is 20, and the cost corresponding to key 20 is 2).

The cost of c[2,3] is 6 (The key is 30, and the cost corresponding to key 30 is 6)
The cost of c[3,4] is 3 (The key is 40, and the cost corresponding to key 40 is 3)

	0	1	2	3	4
0	0	4			
1		0	2		
2			0	6	
3				0	3
4					0

Now we will calculate the values where j-i = 2
When j=2, i=0 then j-i = 2
When j=3, i=1 then j-i = 2
When j=4, i=2 then j-i = 2
In this case, we will consider two keys.

- When i=0 and j=2, then keys 10 and 20. There are two possible trees that can be made out from these two keys shown below:



In the first binary tree, cost would be: 4*1 + 2*2 = 8
In the second binary tree, cost would be: 4*2 + 2*1 = 10
The minimum cost is 8; therefore, c[0,2] = 8

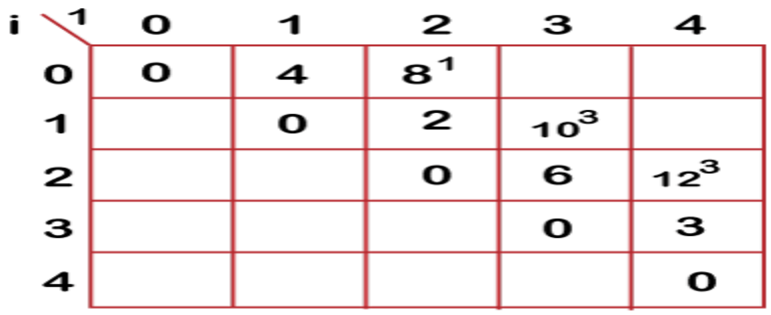
	0	1	2	3	4
0	0	4	8		
1		0	2		
2			0	6	
3				0	3
4					0

- When i=1 and j=3, then keys 20 and 30. There are two possible trees that can be made out from these two keys shown below:

In the first binary tree, cost would be: 1*2 + 2*6 = 14
In the second binary tree, cost would be: 1*6 + 2*2 = 10
The minimum cost is 10; therefore, c[1,3] = 10

- When i=2 and j=4, we will consider the keys at 3 and 4, i.e., 30 and 40. There are two possible trees that can be made out from these two keys shown as below:

In the first binary tree, cost would be: 1*6 + 2*3 = 12
In the second binary tree, cost would be: 1*3 + 2*6 = 15
The minimum cost is 12, therefore, c[2,4] = 12

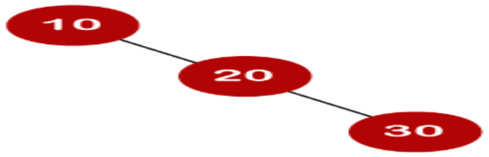


i \ j	0	1	2	3	4
0	0	4	8^1		
1		0	2	10^3	
2			0	6	12^3
3				0	3
4					0

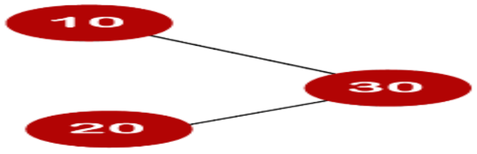
Now we will calculate the values when j-i = 3
When j=3, i=0 then j-i = 3
When j=4, i=1 then j-i = 3

- When i=0, j=3 then we will consider three keys, i.e., 10, 20, and 30.

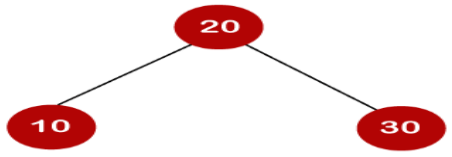
The following are the trees that can be made if 10 is considered as a root node.



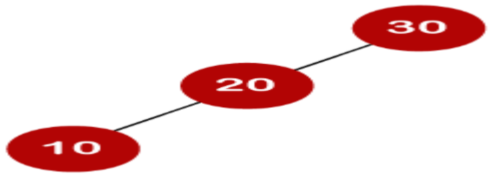
In the above tree, 10 is the root node, 20 is the right child of node 10, and 30 is the right child of node 20.
Cost would be: 1*4 + 2*2 + 3*6 = 26



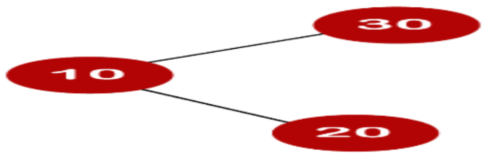
In the above tree, 10 is the root node, 30 is the right child of node 10, and 20 is the left child of node 20.
Cost would be: 1*4 + 2*6 + 3*2 = 22
The following tree can be created if 20 is considered as the root node.



In the above tree, 20 is the root node, 30 is the right child of node 20, and 10 is the left child of node 20.
Cost would be: 1*2 + 4*2 + 6*2 = 22
The following are the trees that can be created if 30 is considered as the root node.



In the above tree, 30 is the root node, 20 is the left child of node 30, and 10 is the left child of node 20.
Cost would be: 1*6 + 2*2 + 3*4 = 22



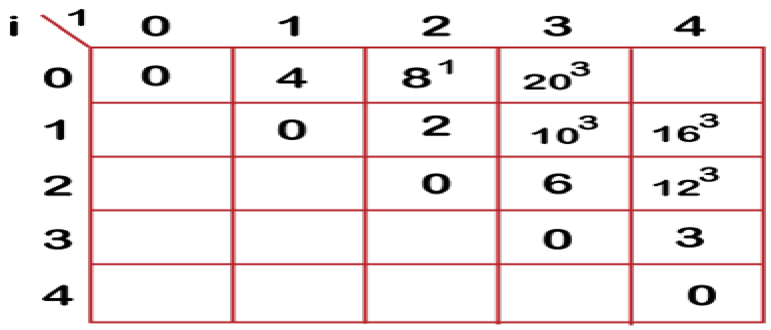
In the above tree, 30 is the root node, 10 is the left child of node 30 and 20 is the right child of node 10.

Cost would be: 1*6 + 2*4 + 3*2 = 20

Therefore, the minimum cost is 20 which is the 3rd root. So, c[0,3] is equal to 20.

- When i=1 and j=4 then we will consider the keys 20, 30, 40

c[1,4] = min{ c[1,1] + c[2,4], c[1,2] + c[3,4], c[1,3] + c[4,4] } + 11
= min{0+12, 2+3, 10+0}+ 11
= min{12, 5, 10} + 11
The minimum value is 5; therefore, c[1,4] = 5+11 = 16



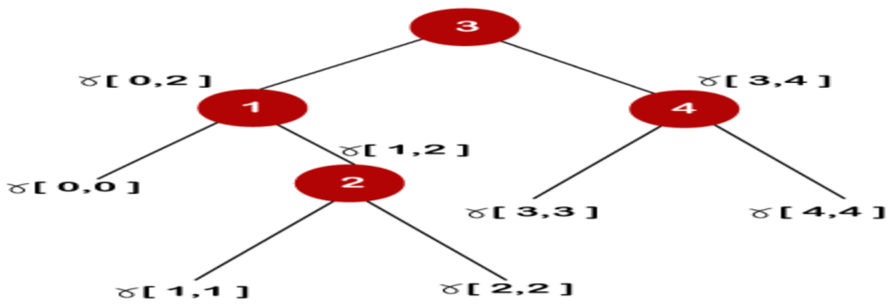
i \ 1	0	1	2	3	4
0	0	4	8^1	20^3	
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

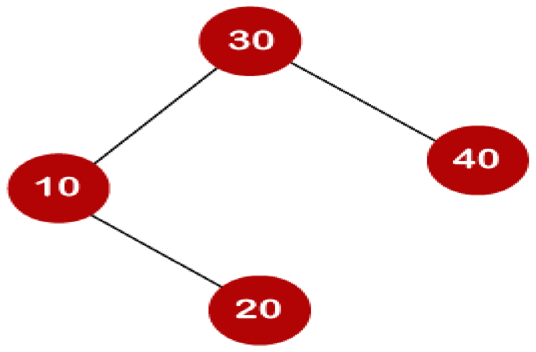
- **Now we will calculate the values when j-i = 4**

When j=4 and i=0 then j-i = 4
In this case, we will consider four keys, i.e., 10, 20, 30 and 40. The frequencies of 10, 20, 30 and 40 are 4, 2, 6 and 3 respectively.
w[0, 4] = 4 + 2 + 6 + 3 = 15
If we consider 10 as the root node then
C[0, 4] = min {c[0,0] + c[1,4]}+ w[0,4]
= min {0 + 16} + 15= 31
If we consider 20 as the root node then
C[0,4] = min{c[0,1] + c[2,4]} + w[0,4]
= min{4 + 12} + 15
= 16 + 15 = 31
If we consider 30 as the root node then,
C[0,4] = min{c[0,2] + c[3,4]} +w[0,4]
= min {8 + 3} + 15
= 26
If we consider 40 as the root node then,
C[0,4] = min{c[0,3] + c[4,4]} + w[0,4]
= min{20 + 0} + 15
= 35
In the above cases, we have observed that 26 is the minimum cost; therefore, c[0,4] is equal to 26.

i \ 1	0	1	2	3	4
0	0	4	8^1	20^3	26^3
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

The optimal binary tree can be created as:



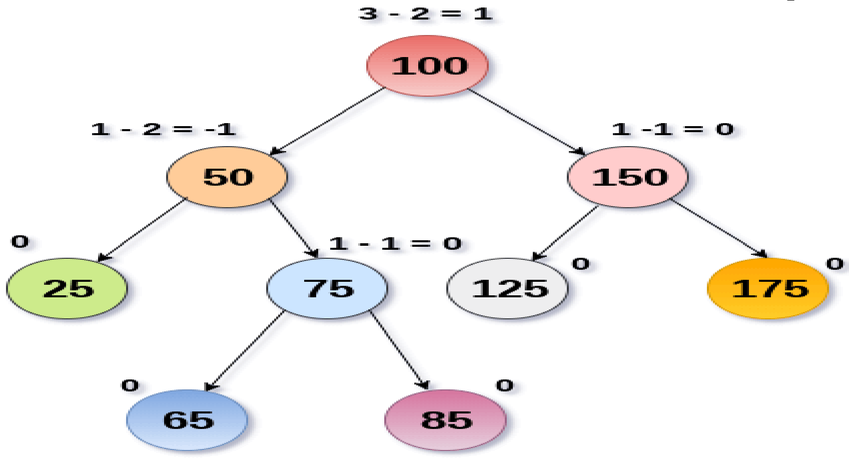


General formula for calculating the minimum cost is:
C[i,j] = min{c[i, k-1] + c[k,j]} + w(i,j)

AVL TREE(Explain AVL Tree)(Explain Height balanced tree)

- AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.
- AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.
- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.
- Balance Factor (k) = height (left(k)) - height (right(k))
- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

- An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

Complexity

Algorithm	Average case	Worst case
Space	o(n)	o(n)
Search	o(log n)	o(log n)
Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

Operations on AVL tree

- Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree.
- Searching and traversing do not lead to the violation in property of AVL tree.
- However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

AVL tree controls the height of the binary search tree by not letting it to be skewed.

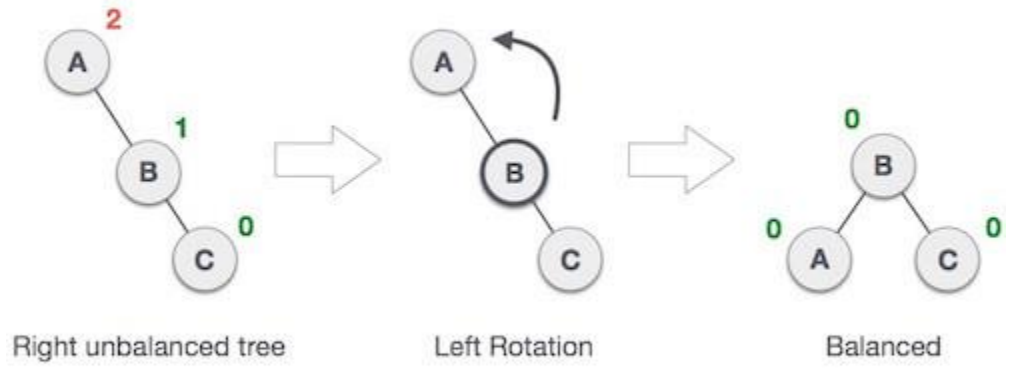
- The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extended to **O(n)** if the BST becomes skewed (i.e. worst case).
- By limiting this height to log n, AVL tree imposes an upper bound on each operation to be **O(log n)** where n is the number of nodes.

AVL Rotations

- We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:
  1. L L rotation: Inserted node is in the left subtree of left subtree of A
  2. R R rotation : Inserted node is in the right subtree of right subtree of A
  3. L R rotation : Inserted node is in the right subtree of left subtree of A
  4. R L rotation : Inserted node is in the left subtree of right subtree of A
- Where node A is the node whose balance Factor is other than -1, 0, 1.
- The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

1. RR Rotation

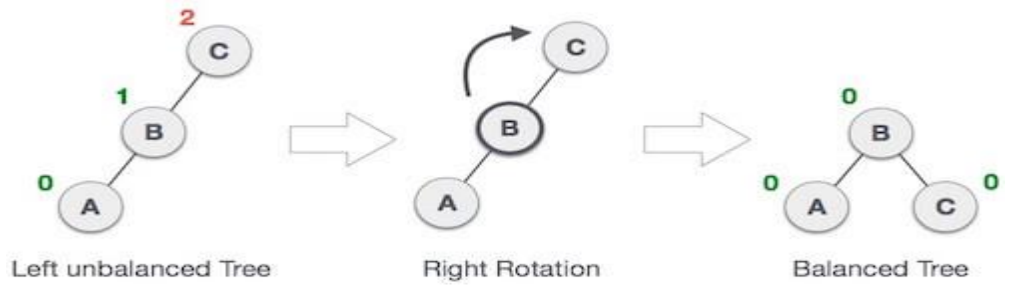
- When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



- In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

2. LL Rotation

- When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.

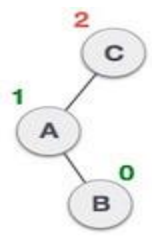
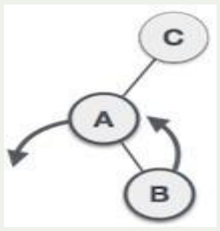
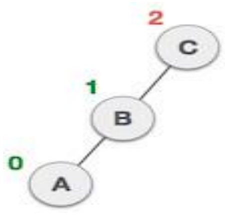
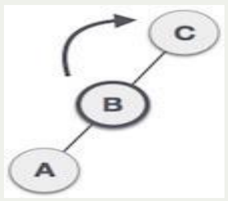
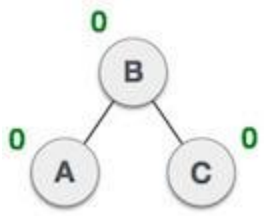


- In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

3. LR Rotation

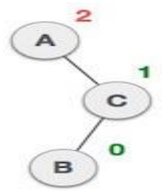
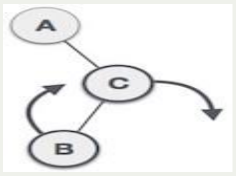
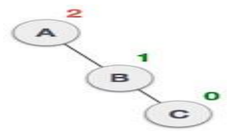
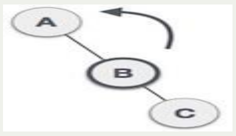
- Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

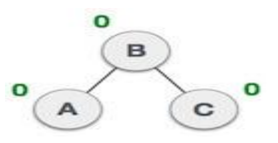
Let us understand each and every step very clearly:

State	Action
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**.
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C**
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

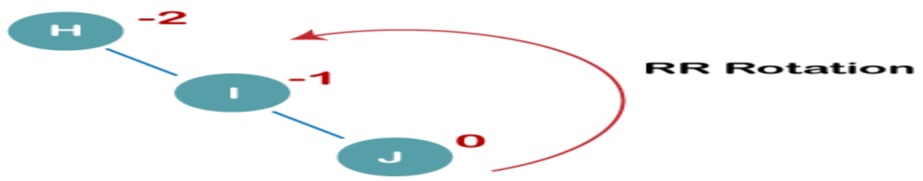
4. RL Rotation

- As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

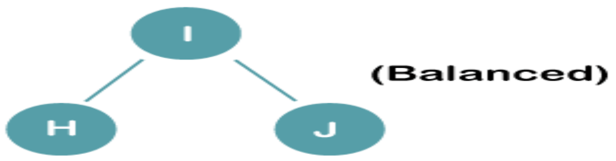
State	Action
	A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**.
	After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.
	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B.

	Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

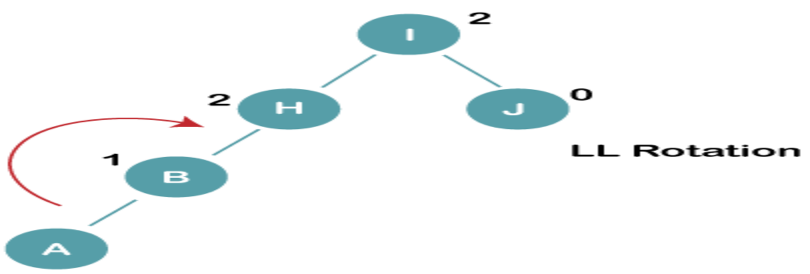
Q: Construct an AVL tree having the following elements
H, I, J, B, A, E, C, F, D, G, K, L
1. Insert H, I, J



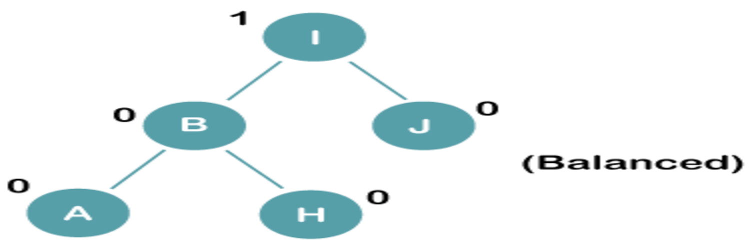
On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.
The resultant balance tree is:



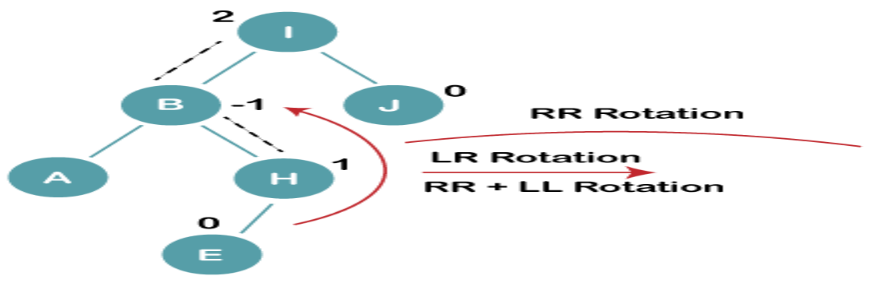
2. Insert B, A



On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H.
The resultant balance tree is:



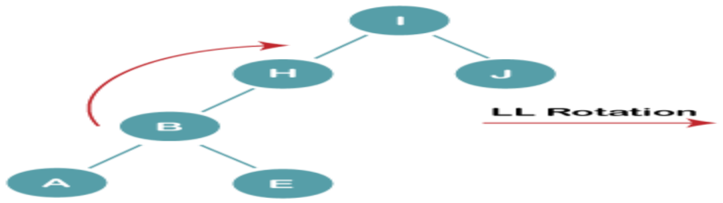
3. Insert E



On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation

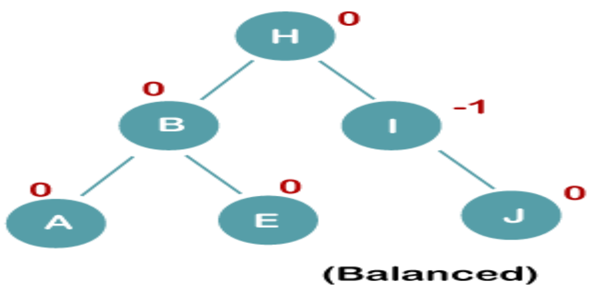
3 a) We first perform RR rotation on node B

The resultant tree after RR rotation is:

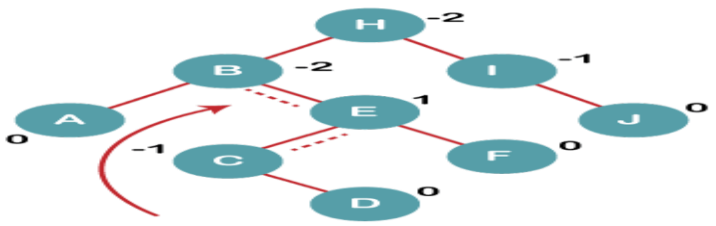


3b) We first perform LL rotation on the node I

The resultant balanced tree after LL rotation is:



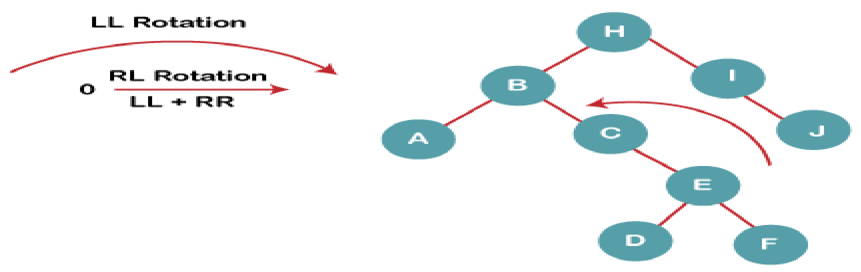
4. Insert C, F, D



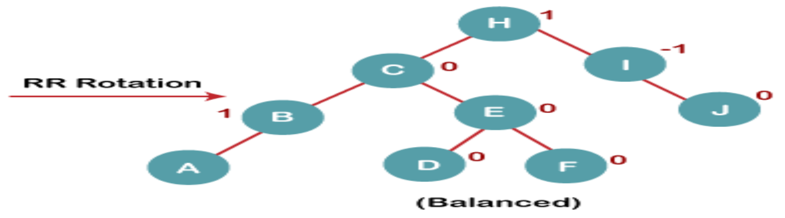
On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2, since if we travel from D to B we find that it is inserted in the right subtree of left subtree of B, we will perform RL Rotation on node I. RL = LL + RR rotation.

4a) We first perform LL rotation on node E

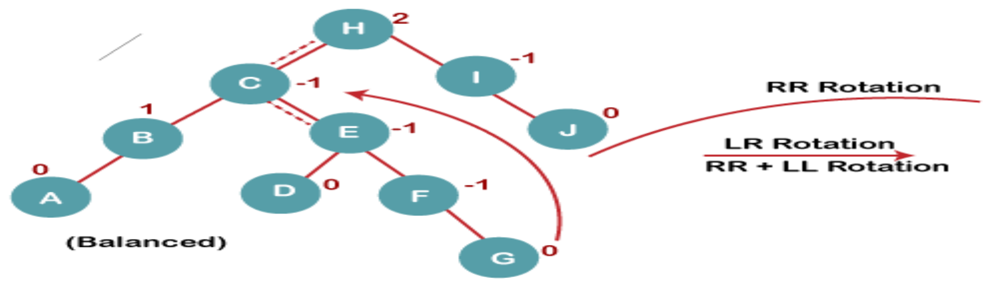
The resultant tree after LL rotation is:



4b) We then perform RR rotation on node B
The resultant balanced tree after RR rotation is:

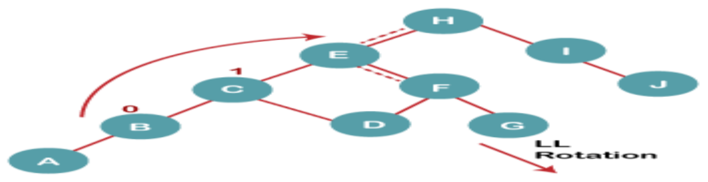


5. Insert G

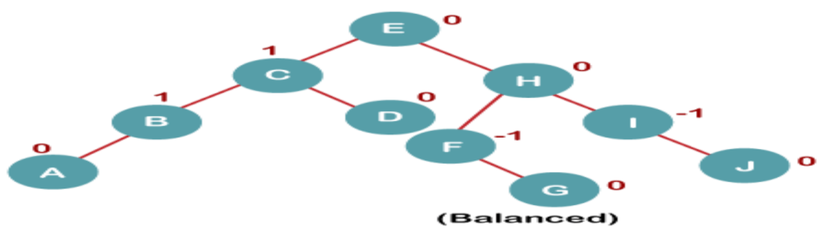


On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation.

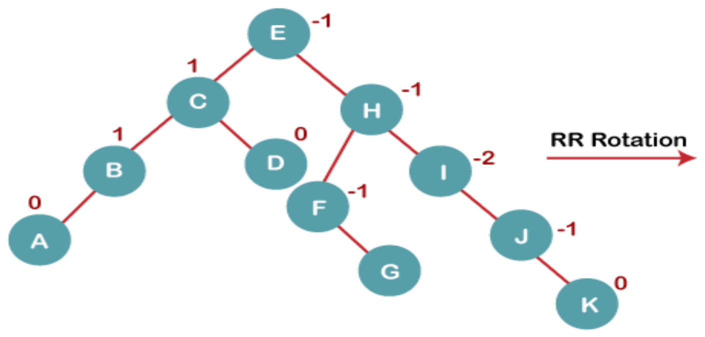
5 a) We first perform RR rotation on node C
The resultant tree after RR rotation is:



5 b) We then perform LL rotation on node H
The resultant balanced tree after LL rotation is:

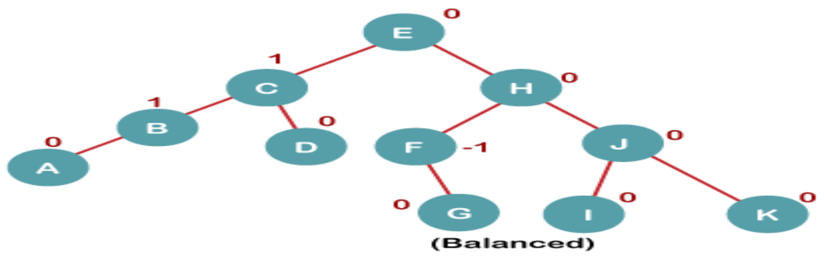


6. Insert K



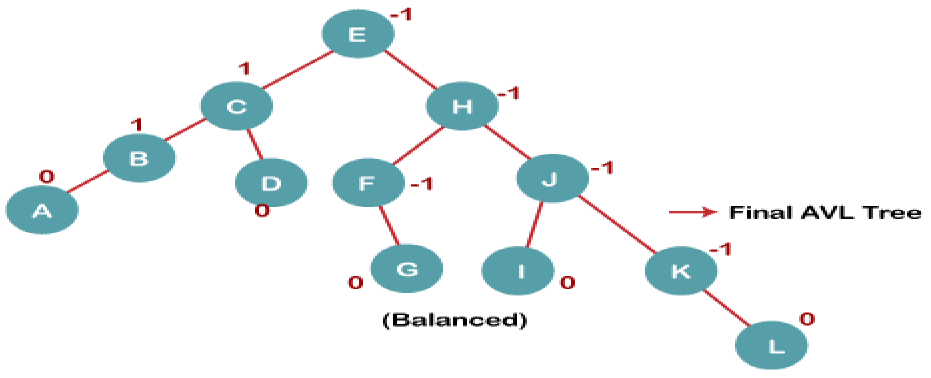
On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.

The resultant balanced tree after RR rotation is:



7. Insert L

On inserting the L tree is still balanced as the Balance Factor of each node is now either, -1, 0, +1. Hence the tree is a Balanced AVL tree


E -1
C 1
H -1
B 1
D 0
F -1
J -1
A 0
G 0
I 0
K -1
L 0
Final AVL Tree
(Balanced)