**[Binary Search Algorithm | Example | Time Complexity](https://www.gatevidyalay.com/binary-search-binary-search-algorithm/)**

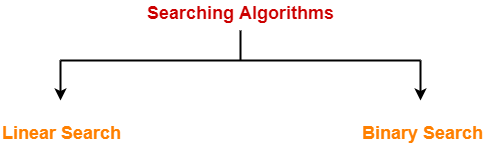
**Searching-**

* Searching is a process of finding a particular element among several given elements.
* The search is successful if the required element is found.
* Otherwise, the search is unsuccessful.

**Searching Algorithms-**

Searching Algorithms are a family of algorithms used for the purpose of searching.

The searching of an element in the given array may be carried out in the following two ways-



1. Linear Search
2. Binary Search

**Binary Search-**

* Binary Search is one of the fastest searching algorithms.
* It is used for finding the location of an element in a linear array.
* It works on the principle of divide and conquer technique.

Binary Search Algorithm can be applied only on **Sorted arrays**.

So, the elements must be arranged in-

* Either ascending order if the elements are numbers.
* Or dictionary order if the elements are strings.

To apply binary search on an unsorted array,

* First, sort the array using some sorting technique.
* Then, use binary search algorithm.

**Binary Search Algorithm-**

Consider-

* There is a linear array ‘a’ of size ‘n’.
* Binary search algorithm is being used to search an element ‘item’ in this linear array.
* If search ends in success, it sets loc to the index of the element otherwise it sets loc to -1.
* Variables beg and end keeps track of the index of the first and last element of the array or sub array in which the element is being searched at that instant.
* Variable mid keeps track of the index of the middle element of that array or sub array in which the element is being searched at that instant.

Then, Binary Search Algorithm is as follows-

Begin

Set low = 0

Set high = n-1

Set mid = (low +high) / 2

**while** ( (low <= high) and (a[mid] ≠ item) ) **do**

**if** (item < a[mid]) **then**

Set high = mid - 1

**else**

Set low = mid + 1

endif

Set mid = (low + high) / 2

endwhile

**if** (low >high) **then**

Set loc = -1

**else**

Set low = mid

endif

End

C Program

#include<stdio.h>

main()

{

int c,first,last,middle,n,search,a[5],i,j,key,temp, flag=0;

printf("Please Enter the Number[] of Elements to be Entered:\t");

scanf("%d",&n);

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

{

printf("\nPlease EnterElement%d:",i+1);

scanf("%d",&a[i]);

}

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

printf("\nTheElement:a[%d]:%d",i,a[i]);

printf("\nPress ANY key to Continue...");

getch();

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

{

for(j=0;j<n-i;j++)

{

if(a[j]>a[j+1])

{

temp=a[j];

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

a[j+1]=temp;

}

}

}

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

printf("\nTheElement:a[%d]:%d",i,a[i]);

printf("Elements are in ORDER:");

getch();

printf("Please Enter the Key to be Searched:");

scanf("%d",&key);

printf("Key:%d",key);

int low=0,high=n-1,mid;

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

{

mid=(low+high)/2;

if(key==a[mid])

{

flag=1;

break;

getch();

}

else

if(key>a[mid])

{

low=mid+1;

getch();

continue;

}

else

{

high=mid-1;

getch();

continue;

}

}

if(flag==1)

printf("key found at INDEX %d", a[mid],mid-1);

else

printf("key not found");

}

|  |
| --- |
| **Explanation**    Binary Search Algorithm searches an element by comparing it with the middle most element of the array.  Then, following three cases are possible-    **Case-01**    If the element being searched is found to be the middle most element, its index is returned.    **Case-02**    If the element being searched is found to be greater than the middle most element,  then its search is further continued in the right sub array of the middle most element.    **Case-03**    If the element being searched is found to be smaller than the middle most element,  then its search is further continued in the left sub array of the middle most element.    This iteration keeps on repeating on the sub arrays until the desired element is found  or size of the sub array reduces to zero. |

**Time Complexity Analysis-**

Binary Search time complexity analysis is done below-

* In each iteration or in each recursive call, the search gets reduced to half of the array.
* So for n elements in the array, there are log2n iterations or recursive calls.

Thus, we have-

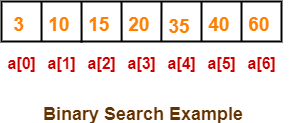
|  |
| --- |
| **Time Complexity of Binary Search Algorithm is O(log2n).**  Here, n is the number of elements in the sorted linear array. |

This time complexity of binary search remains unchanged irrespective of the element position even if it is not present in the array.

**Binary Search Example-**

Consider-

* We are given the following sorted linear array.
* Element 15 has to be searched in it using Binary Search Algorithm.



Binary Search Algorithm works in the following steps-

**Step-01:**

* To begin with, we take beg=0 and end=6.
* We compute location of the middle element as-

mid

= (beg + end) / 2

= (0 + 6) / 2

= 3

* Here, a[mid] = a[3] = 20 ≠ 15 and beg < end.
* So, we start next iteration.

**Step-02:**

* Since a[mid] = 20 > 15, so we take end = mid – 1 = 3 – 1 = 2 whereas beg remains unchanged.
* We compute location of the middle element as-

mid

= (beg + end) / 2

= (0 + 2) / 2

= 1

* Here, a[mid] = a[1] = 10 ≠ 15 and beg < end.
* So, we start next iteration.

**Step-03:**

* Since a[mid] = 10 < 15, so we take beg = mid + 1 = 1 + 1 = 2 whereas end remains unchanged.
* We compute location of the middle element as-

mid

= (beg + end) / 2

= (2 + 2) / 2

= 2

* Here, a[mid] = a[2] = 15 which matches to the element being searched.
* So, our search terminates in success and index 2 is returned.

**Binary Search Algorithm Advantages-**

The advantages of binary search algorithm are-

* It eliminates half of the list from further searching by using the result of each comparison.
* It indicates whether the element being searched is before or after the current position in the list.
* This information is used to narrow the search.
* For large lists of data, it works significantly better than linear search.

**Binary Search Algorithm Disadvantages-**

The disadvantages of binary search algorithm are-

* It employs recursive approach which requires more stack space.
* Programming binary search algorithm is error prone and difficult.
* The interaction of binary search with memory hierarchy i.e. caching is poor.

(because of its random access nature)

**Important Note-**

For in-memory searching, if the interval to be searched is small,

* Linear search may exhibit better performance than binary search.
* This is because it exhibits better locality of reference.

[**Selection Sort Algorithm | Example | Time Complexity**](https://www.gatevidyalay.com/selection-sort-selection-sort-algorithm/)

**Selection Sort-**

* Selection sort is one of the easiest approaches to sorting.
* It is inspired from the way in which we sort things out in day to day life.
* It is an in-place sorting algorithm because it uses no auxiliary data structures while sorting.

**How Selection Sort Works?**

Consider the following elements are to be sorted in ascending order using selection sort-

6, 2, 11, 7, 5

Selection sort works as-

* It finds the first smallest element (2).
* It swaps it with the first element of the unordered list.
* It finds the second smallest element (5).
* It swaps it with the second element of the unordered list.
* Similarly, it continues to sort the given elements.

As a result, sorted elements in ascending order are-

2, 5, 6, 7, 11

**Selection Sort Algorithm-**

Let A be an array with n elements. Then, selection sort algorithm used for sorting is as follows-

2, 5, 6, 7, 11 n=5

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

{

index = i;

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

{

**if**(A[j] < A[index])

index = j;

}

temp = A[i];

A[i] = A[index];

A[index] = temp;

}

 Here,

* i = variable to traverse the array A
* index = variable to store the index of minimum element
* j = variable to traverse the unsorted sub-array
* temp = temporary variable used for swapping

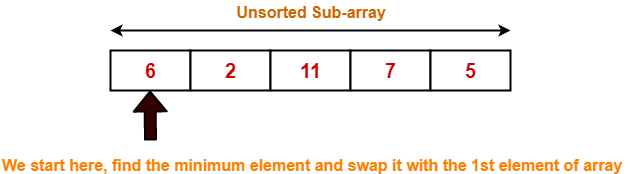
**Selection Sort Example-**

Consider the following elements are to be sorted in ascending order-

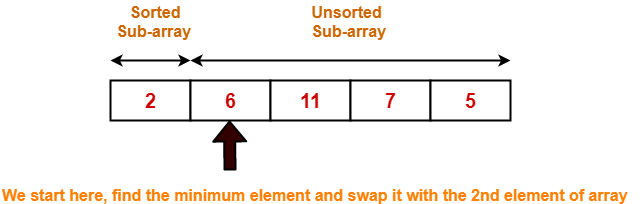
6, 2, 11, 7, 5

The above selection sort algorithm works as illustrated below-

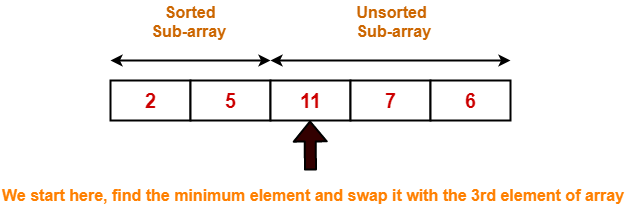
**Step-01: For i = 0**



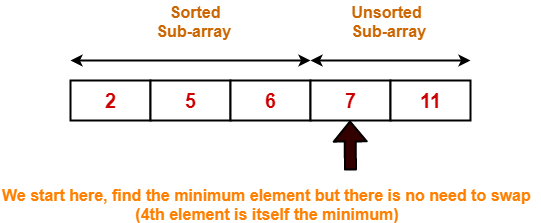
**Step-02: For i = 1**



**Step-03: For i = 2**



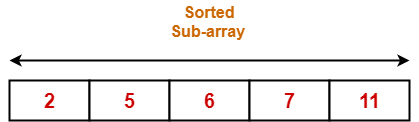
**Step-04: For i = 3**



**Step-05: For i = 4**

Loop gets terminated as ‘i’ becomes 4.

The state of array after the loops are finished is as shown-



With each loop cycle,

* The minimum element in unsorted sub-array is selected.
* It is then placed at the correct location in the sorted sub-array until array A is completely sorted.

**Time Complexity Analysis-**

* Selection sort algorithm consists of two nested loops.
* Owing to the two nested loops, it has O(n2) time complexity.

|  |  |
| --- | --- |
|  | **Time Complexity** |
| Best Case | n2 |
| Average Case | n2 |
| Worst Case | n2 |

**Space Complexity Analysis-**

* Selection sort is an in-place algorithm.
* It performs all computation in the original array and no other array is used.
* Hence, the space complexity works out to be O(1).

**Important Notes-**

* Selection sort is not a very efficient algorithm when data sets are large.
* This is indicated by the average and worst case complexities.
* Selection sort uses minimum number of swap operations O(n) among all the sorting algorithms.
* **Bubble Sort Algorithm | Example | Time Complexity**

**Bubble Sort-**

* Bubble sort is the easiest sorting algorithm to implement.
* It is inspired by observing the behavior of air bubbles over foam.
* It is an in-place sorting algorithm.
* It uses no auxiliary data structures (extra space) while sorting.

**How Bubble Sort Works?**

* Bubble sort uses multiple passes (scans) through an array.
* In each pass, bubble sort compares the adjacent elements of the array.
* It then swaps the two elements if they are in the wrong order.
* In each pass, bubble sort places the next largest element to its proper position.
* In short, it bubbles down the largest element to its correct position.

**Bubble Sort Algorithm-**

The bubble sort algorithm is given below-

for(int pass=1 ; pass<=n-1 ; ++pass) // Making passes through array

{

**for**(int i=0 ; i<=n-2 ; ++i)

{

**if**(A[i] > A[i+1]) // If adjacent elements are in wrong order

swap(i,i+1,A); // Swap them

}

}

//swap function : Exchange elements from array A at position x,y

**void** swap(int x, int y, int[] A)

{

int temp = A[x];

A[x] = A[y];

A[y] = temp;

**return** ;

}

// pass : Variable to count the number of passes that are done till now

// n : Size of the array

// i : Variable to traverse the array A

// swap() : Function to swap two numbers from the array

// x,y : Indices of the array that needs to be swapped

**Bubble Sort Example-**

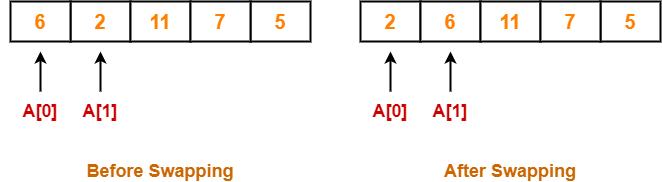
Consider the following array A-



Now, we shall implement the above bubble sort algorithm on this array.

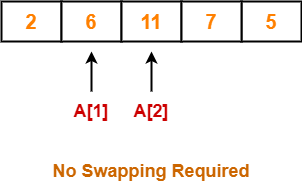
**Step-01:**

* We have pass=1 and i=0.
* We perform the comparison A[0] > A[1] and swaps if the 0th element is greater than the 1th element.
* Since 6 > 2, so we swap the two elements.



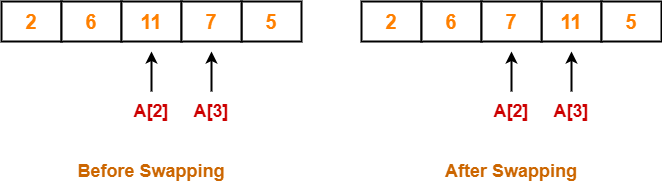
**Step-02:**

* We have pass=1 and i=1.
* We perform the comparison A[1] > A[2] and swaps if the 1th element is greater than the 2th element.
* Since 6 < 11, so no swapping is required.



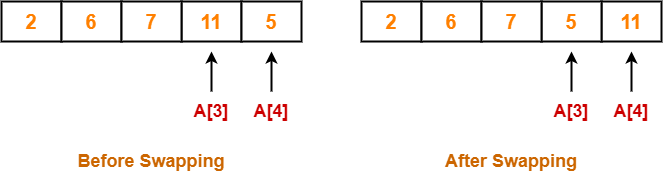
**Step-03:**

* We have pass=1 and i=2.
* We perform the comparison A[2] > A[3] and swaps if the 2nd element is greater than the 3rd element.
* Since 11 > 7, so we swap the two elements.



**Step-04:**

* We have pass=1 and i=3.
* We perform the comparison A[3] > A[4] and swaps if the 3rd element is greater than the 4th element.
* Since 11 > 5, so we swap the two elements.



Finally after the first pass, we see that the largest element 11 reaches its correct position.

**Step-05:**

* Similarly after pass=2, element 7 reaches its correct position.
* The modified array after pass=2 is shown below-

https://www.gatevidyalay.com/wp-content/uploads/2020/09/Bubble-Sort-Example-Step-By-Step-Step-05-1.png

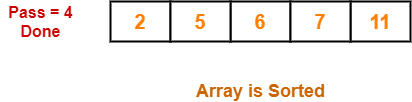
**Step-06:**

* Similarly after pass=3, element 6 reaches its correct position.
* The modified array after pass=3 is shown below-

https://www.gatevidyalay.com/wp-content/uploads/2020/09/Bubble-Sort-Example-Step-By-Step-Step-06.png

**Step-07:**

* No further improvement is done in pass=4.
* This is because at this point, elements 2 and 5 are already present at their correct positions.
* The loop terminates after pass=4.
* Finally, the array after pass=4 is shown below-



**Optimization Of Bubble Sort Algorithm-**

* If the array gets sorted after a few passes like one or two, then ideally the algorithm should terminate.
* But still the above algorithm executes the remaining passes which costs extra comparisons.

**Optimized Bubble Sort Algorithm-**

The optimized bubble sort algorithm is shown below-

for (int pass=1 ; pass<=n-1 ; ++pass)

{

flag=0 // flag denotes are there any swaps done in pass

**for** (int i=0 ; i<=n-2 ; ++i)

{

**if**(A[i] > A[i+1])

{

swap(i,i+1,A);

flag=1 // After swap, set flag to 1

}

}

**if**(flag == 0) break; // No swaps indicates we can terminate loop

}

**void** swap(int x, int y, int[] A)

{

int temp = A[x];

A[x] = A[y];

A[y] = temp;

**return**;

}

**Explanation-**

* To avoid extra comparisons, we maintain a flag variable.
* The flag variable helps to break the outer loop of passes after obtaining the sorted array.
* The initial value of the flag variable is set to 0.
* The zero value of flag variable denotes that we have not encountered any swaps.
* Once we need to swap adjacent values for correcting their wrong order, the value of flag variable is set to 1.
* If we encounter a pass where flag == 0, then it is safe to break the outer loop and declare the array is sorted.

**Time Complexity Analysis-**

* Bubble sort uses two loops- inner loop and outer loop.
* The inner loop deterministically performs O(n) comparisons.

**Worst Case-**

* In worst case, the outer loop runs O(n) times.
* Hence, the worst case time complexity of bubble sort is O(n x n) = O(n2).

**Best Case-**

* In best case, the array is already sorted but still to check, bubble sort performs O(n) comparisons.
* Hence, the best case time complexity of bubble sort is O(n).

**Average Case-**

* In average case, bubble sort may require (n/2) passes and O(n) comparisons for each pass.
* Hence, the average case time complexity of bubble sort is O(n/2 x n) = Θ(n2).

The following table summarizes the time complexities of bubble sort in each case-

|  |  |
| --- | --- |
|  | **Time Complexity** |
| **Best Case** | **O(n)** |
| **Average Case** | **Θ(n2)** |
| **Worst Case** | **O(n2)** |

From here, it is clear that bubble sort is not at all efficient in terms of time complexity of its algorithm.

**Space Complexity Analysis-**

* Bubble sort uses only a constant amount of extra space for variables like flag, i, n.
* Hence, the space complexity of bubble sort is O(1).
* It is an in-place sorting algorithm i.e. it modifies elements of the original array to sort the given array.

**Properties-**

Some of the important properties of bubble sort algorithm are-

* Bubble sort is a stable sorting algorithm.
* Bubble sort is an in-place sorting algorithm.
* The worst case time complexity of bubble sort algorithm is O(n2).
* The space complexity of bubble sort algorithm is O(1).
* Number of swaps in bubble sort = Number of inversion pairs present in the given array.
* Bubble sort is beneficial when array elements are less and the array is nearly sorted.

**PRACTICE PROBLEMS BASED ON MERGE SORT ALGORITHM-**

**Problem-01:**

The number of swapping needed to sort the numbers 8, 22, 7, 9, 31, 5, 13 in ascending order using bubble sort is--------

1. 11
2. 12 2. 8,5 4. 22,31 5. 7,5 6. 9,5 7. 31,5
   1. 31,13 9. 8,31, 10. 8.13
3. 13
4. 10

Inversion Pair

8, 22, 7, 9, 31, 5, 13

**Solution-**

In bubble sort, Number of swaps required = Number of inversion pairs.

Here, there are 10 inversion pairs present which are-

1. (8,7)
2. (22,7)
3. (22,9)
4. (8,5)
5. (22,5)
6. (7,5)
7. (9,5)
8. (31,5)
9. (22,13)
10. (31,13)

***Thus, Option (D) is correct.***

**Problem-02:**

When will bubble sort take worst-case time complexity?

1. The array is sorted in ascending order.
2. The array is sorted in descending order.
3. Only the first half of the array is sorted.
4. Only the second half of the array is sorted.

**Solution-**

* In bubble sort, Number of swaps required = Number of inversion pairs.
* When an array is sorted in descending order, the number of inversion pairs = n(n-1)/2 which is maximum for any permutation of array.

***Thus, Option (B) is correct.***

 MAY 18, 2022 WEDNESDAY

**Insertion Sort Algorithm | Example | Time Complexity**

**Insertion Sort-**

* Insertion sort is an in-place sorting algorithm.
* It uses no auxiliary data structures while sorting.
* It is inspired from the way in which we sort playing cards.

**How Insertion Sort Works?**

Consider the following elements are to be sorted in ascending order-

6, 2, 11, 7, 5

Insertion sort works as-

Firstly,

* It selects the second element (2).
* It checks whether it is smaller than any of the elements before it.
* Since 2 < 6, so it shifts 6 towards right and places 2 before it.
* The resulting list is 2, 6, 11, 7, 5.

Secondly,

* It selects the third element (11).
* It checks whether it is smaller than any of the elements before it.
* Since 11 > (2, 6), so no shifting takes place.
* The resulting list remains the same.

Thirdly,

* It selects the fourth element (7).
* It checks whether it is smaller than any of the elements before it.
* Since 7 < 11, so it shifts 11 towards right and places 7 before it.
* The resulting list is 2, 6, 7, 11, 5.

Fourthly,

* It selects the fifth element (5).
* It checks whether it is smaller than any of the elements before it.
* Since 5 < (6, 7, 11), so it shifts (6, 7, 11) towards right and places 5 before them.
* The resulting list is 2, 5, 6, 7, 11.

As a result, sorted elements in ascending order are-

2, 5, 6, 7, 11

12, 8, 10, 5, 3, 7

1. 8<12 🡪 8,12,10,5,3,7
2. 10>8, 10<12, 🡪8,10,12,5,3,7
3. 5<(8,10,12)🡪 5, 8, 10, 12, 3, 7
4. 3<(5,8,10,12)🡪 3,5, 8, 10, 12, 7
5. 7>(3,5) 7<(8,10,12)🡪 3,5, 7, 8, 10, 12

**Insertion Sort Algorithm-**

Let A be an array with n elements. The insertion sort algorithm used for sorting is as follows-

for (i = 1 ; i < n ; i++) 12, 8, 10, 5, 3, 7 A[0]=12, A[1]=8, A[2]=10, A[3]=5, A[4]=3, A[5]=7,

{ n=6

key = A [ i ]; key=A[2]=10

j = i - 1; j=2-1=1

**while**(j > 0 && A [ j ] > key) T && A[1]=8>10

{

A [ j+1 ] = A [ j ];

j--;

}

A [ j+1 ] = key; A[1]=8

}

Here,

* i = variable to traverse the array A
* key = variable to store the new number to be inserted into the sorted sub-array
* j = variable to traverse the sorted sub-array

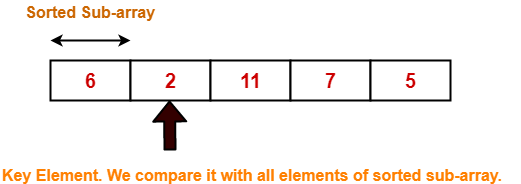
**Insertion Sort Example-**

Consider the following elements are to be sorted in ascending order-

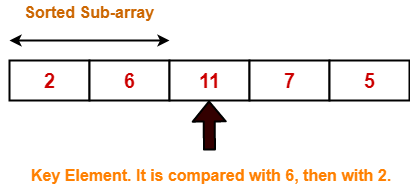
6, 2, 11, 7, 5

The above insertion sort algorithm works as illustrated below-

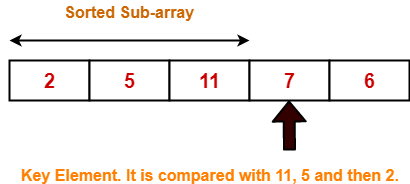
**Step-01: For i = 1**



**Step-02: For i = 2**



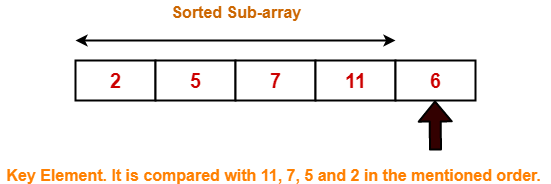
**Step-03: For i = 3**



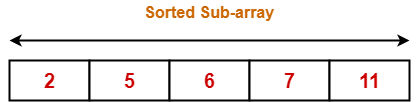
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 5 | 11 | 7 | 6 |  | For j = 2; 11 > 7 so A[3] = 11 |
| 2 | 5 | 11 | 11 | 6 | For j = 1; 5 < 7 so loop stops and A[2] = 7 |
| 2 | 5 | 7 | 11 | 6 | After inner loop ends |

Working of inner loop when i = 3

**Step-04: For i = 4**



Loop gets terminated as ‘i’ becomes 5. The state of array after the loops are finished-



With each loop cycle,

* One element is placed at the correct location in the sorted sub-array until array A is completely sorted.

**Time Complexity Analysis-**

* Selection sort algorithm consists of two nested loops.
* Owing to the two nested loops, it has O(n2) time complexity.

|  |  |
| --- | --- |
|  | **Time Complexity** |
| Best Case | n |
| Average Case | n2 |
| Worst Case | n2 |

**Space Complexity Analysis-**

* Selection sort is an in-place algorithm.
* It performs all computation in the original array and no other array is used.
* Hence, the space complexity works out to be O(1).

**Important Notes-**

* Insertion sort is not a very efficient algorithm when data sets are large.
* This is indicated by the average and worst case complexities.
* Insertion sort is adaptive and number of comparisons are less if array is partially sorted.

[**Merge Sort Algorithm | Example | Time Complexity**](https://www.gatevidyalay.com/merge-sort-algorithm-example-time-complexity/)

**Merge Sort-**

* Merge sort is a famous sorting algorithm.
* It uses a divide and conquer paradigm for sorting.
* It divides the problem into sub problems and solves them individually.
* It then combines the results of sub problems to get the solution of the original problem.

  2,7,6,4,5 3,11,8,10,9

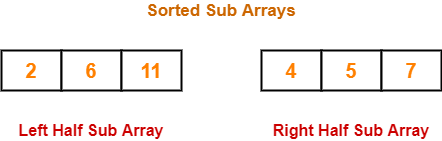
**How Merge Sort Works? 2,4,5,6,7 5,8,9,10,11**

**2,3,4,5,6,7,8,10,11**

Before learning how merge sort works, let us learn about the merge procedure of merge sort algorithm.

The merge procedure of merge sort algorithm is used to merge two sorted arrays into a third array in sorted order.

Consider we want to merge the following two sorted sub arrays into a third array in sorted order-



The merge procedure of merge sort algorithm is given below-

// L : Left Sub Array , R : Right Sub Array , A : Array

merge(L, R, A)

{

nL = length(L) // Size of Left Sub Array

nR = length(R) // Size of Right Sub Array

i = j = k = 0

**while**(i<nL && j<nR)

{ 2<3 && 3<3 L[0]=2, L[1]=6, L[2]=11 R[0]=4, R[1]=5, R[2]=7

/\* When both i and j are valid i.e. when both the sub arrays have elements to insert in A \*/

**if**(L[i] <= R[j]) L[2]<=R[2], 11<=7 🡪 F

{

A[k] = L[i] A[3]=L[1]=6 A[0]=2, A[1]=4, A[2]=5, A[3]=6, A[4]=7, A[0]=2,

k = k+1 k=3+1=4

i = i+1 i=1+1=2

}

**else**

{

A[k] = R[j] A[4]=R[2]=7

k = k+1 k=4+1=5

j = j+1 j=2+1=3

}

}

// Adding Remaining elements from left sub array to array A

**while**(i<nL)

{ 2<3 A[0]=2, A[1]=4, A[2]=5, A[3]=6, A[4]=7, A[5]=11

A[k] = L[i] A[5]=L[2]=11

i = i+1 i=2+1=3

k = k+1 k=5+1=6

}

// Adding Remaining elements from right sub array to array A

**while**(j<nR)

{3<3

A[k] = R[j]

j = j+1

k = k+1

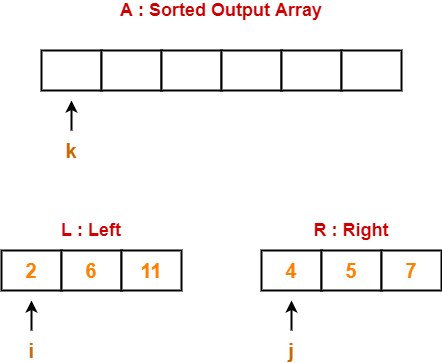
}

}

The above merge procedure of merge sort algorithm is explained in the following steps-

**Step-01:**

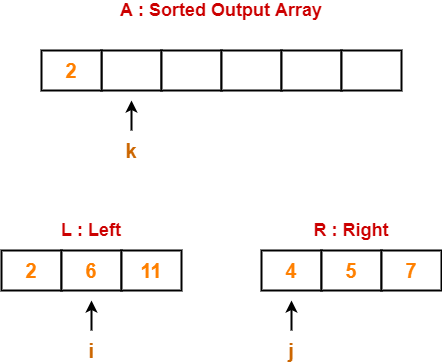
* Create two variables i and j for left and right sub arrays.
* Create variable k for sorted output array.



**Step-02:**

* We have i = 0, j = 0, k = 0.
* Since L[0] < R[0], so we perform A[0] = L[0] i.e. we copy the first element from left sub array to our sorted output array.
* Then, we increment i and k by 1.

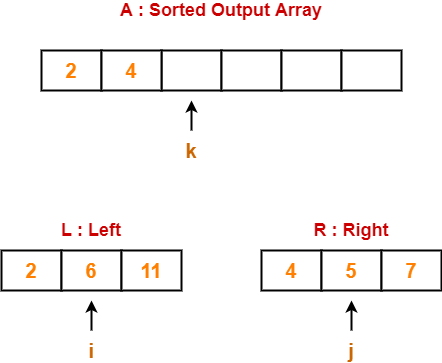
Then, we have-



**Step-03:**

* We have i = 1, j = 0, k = 1.
* Since L[1] > R[0], so we perform A[1] = R[0] i.e. we copy the first element from right sub array to our sorted output array.
* Then, we increment j and k by 1.

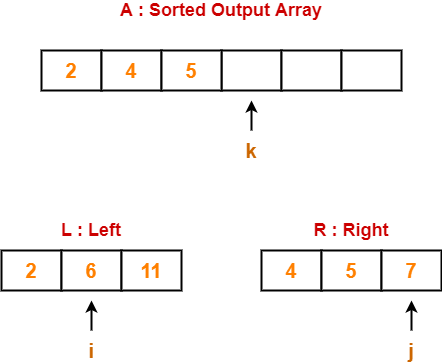
Then, we have-



**Step-04:**

* We have i = 1, j = 1, k = 2.
* Since L[1] > R[1], so we perform A[2] = R[1].
* Then, we increment j and k by 1.

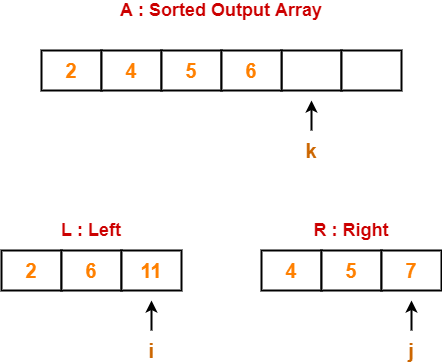
Then, we have-



**Step-05:**

* We have i = 1, j = 2, k = 3.
* Since L[1] < R[2], so we perform A[3] = L[1].
* Then, we increment i and k by 1.

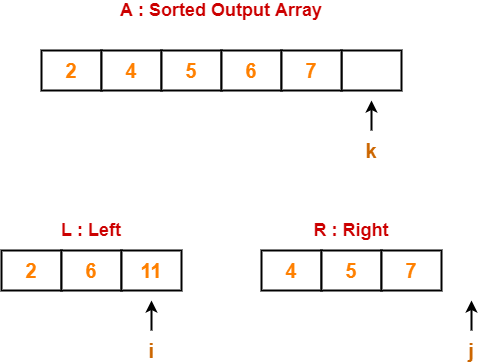
Then, we have-



**Step-06:**

* We have i = 2, j = 2, k = 4.
* Since L[2] > R[2], so we perform A[4] = R[2].
* Then, we increment j and k by 1.

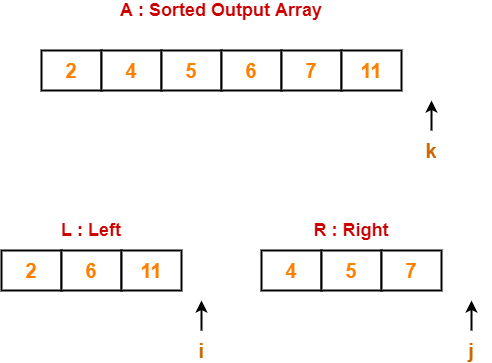
Then, we have-



**Step-07:**

* Clearly, all the elements from right sub array have been added to the sorted output array.
* So, we exit the first while loop with the condition while(i<nL && j<nR) since now j>nR.
* Then, we add remaining elements from the left sub array to the sorted output array using next while loop.

Finally, our sorted output array is-



Basically,

* After finishing elements from any of the sub arrays, we can add the remaining elements from the other sub array to our sorted output array as it is.
* This is because left and right sub arrays are already sorted.

|  |
| --- |
| **Time Complexity**  The above mentioned merge procedure takes Θ(n) time.  This is because we are just filling an array of size n from left & right sub arrays by incrementing i and j at most Θ(n) times. |

**Merge Sort Algorithm-**

Merge Sort Algorithm works in the following steps-

* It divides the given unsorted array into two halves- left and right sub arrays.
* The sub arrays are divided recursively.
* This division continues until the size of each sub array becomes 1.
* After each sub array contains only a single element, each sub array is sorted trivially.
* Then, the above discussed merge procedure is called.
* The merge procedure combines these trivially sorted arrays to produce a final sorted array.

The division procedure of merge sort algorithm which uses recursion is given below-

// A : Array that needs to be sorted

MergeSort(A)

{

n = length(A)

**if** n<2 **return**

mid = n/2

left = new\_array\_of\_size(mid) // Creating temporary array for left

right = new\_array\_of\_size(n-mid) // and right sub arrays

**for**(int i=0 ; i<=mid-1 ; ++i)

{

left[i] = A[i] // Copying elements from A to left

}

**for**(int i=mid ; i<=n-1 ; ++i)

{

right[i-mid] = A[i] // Copying elements from A to right

}

MergeSort(left) // Recursively solving for left sub array

MergeSort(right) // Recursively solving for right sub array

merge(left, right, A) // Merging two sorted left/right sub array to final array

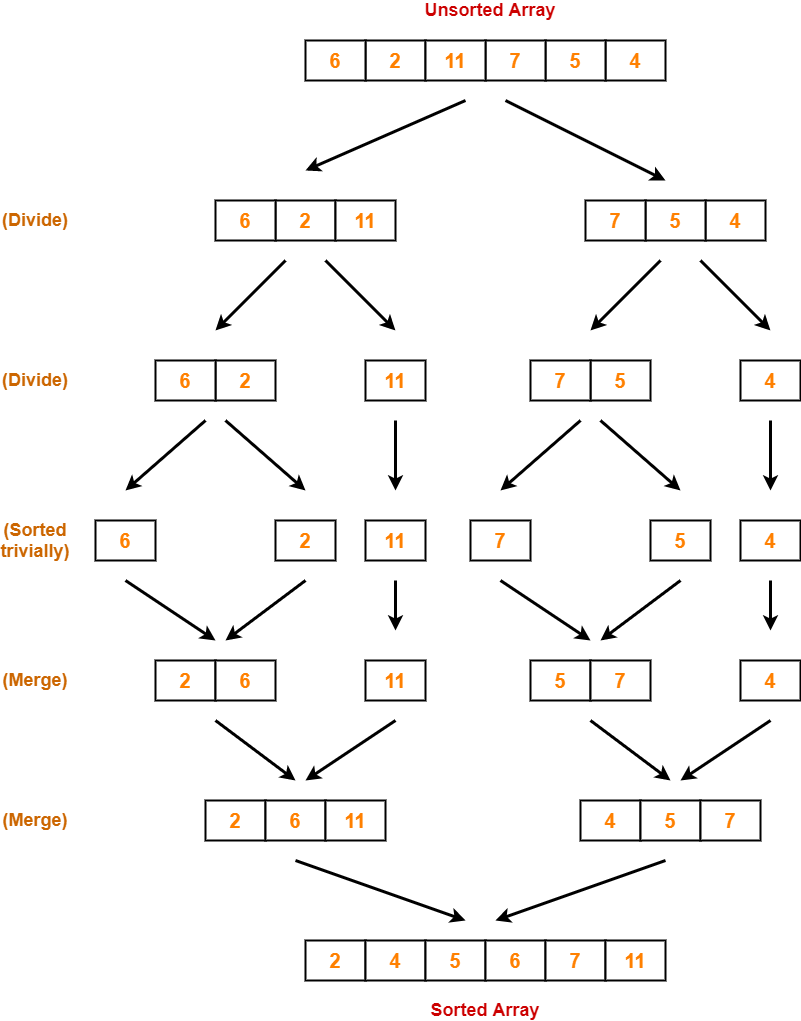
}

**Merge Sort Example-**

Consider the following elements have to be sorted in ascending order-

6, 2, 11, 7, 5, 4

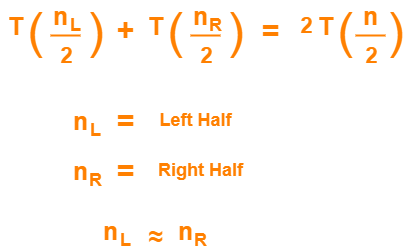
The merge sort algorithm works as-



**Time Complexity Analysis-**

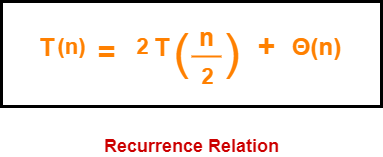
In merge sort, we divide the array into two (nearly) equal halves and solve them recursively using merge sort only.

So, we have-



Finally, we merge these two sub arrays using merge procedure which takes Θ(n) time as explained above.

If T(n) is the time required by merge sort for sorting an array of size n, then the recurrence relation for time complexity of merge sort is-



On solving this recurrence relation, we get T(n) = Θ(nlogn).

***Thus, time complexity of merge sort algorithm is T(n) = Θ(nlogn).***

 Nlon n <n2

**Space Complexity Analysis-**

* Merge sort uses additional memory for left and right sub arrays.
* Hence, total Θ(n) extra memory is needed.

**Properties-**

Some of the important properties of merge sort algorithm are-

* Merge sort uses a divide and conquer paradigm for sorting.
* Merge sort is a recursive sorting algorithm.
* Merge sort is a stable sorting algorithm.
* Merge sort is not an in-place sorting algorithm.
* The time complexity of merge sort algorithm is Θ(nlogn).
* The space complexity of merge sort algorithm is Θ(n).

|  |
| --- |
| **NOTE**  Merge sort is the best sorting algorithm in terms of time complexity Θ(nlogn)  if we are not concerned with auxiliary space used. |

**PRACTICE PROBLEMS BASED ON MERGE SORT ALGORITHM-**

**Problem-**

Assume that a merge sort algorithm in the worst case takes 30 seconds for an input of size 64. Which of the following most closely approximates the maximum input size of a problem that can be solved in 6 minutes?

1. 256
2. 512
3. 1024
4. 2048

**Solution-**

We know, time complexity of merge sort algorithm is Θ(nlogn).

**Step-01:**

It is given that a merge sort algorithm in the worst case takes 30 seconds for an input of size 64.

So, we have-

k x nlogn = 30                   (for n = 64)

k x 64 log64 = 30 k x 64 log 64= k x 64 log 26

k x 64 x 6 = 30 k x 64 x 6= 30

k = 30 /(64 x 6)

k = 5/64

From here, k = 5 / 64.

**Step-02:**

Let n be the maximum input size of a problem that can be solved in 6 minutes (or 360 seconds).

Then, we have-

k x nlogn = 360

(5/64) x nlogn = 360         { Using Result of Step-01 }

nlogn = 72 x 64

nlogn = 4608

log nn = 4608 512\*9=4608

nn = 24608

nn = 2512 \* 9

nn = (29)512

nn = (512)512

n= 512

On solving this equation, we get n = 512.

Thus, correct option is (B).

**Quick Sort Algorithm | Example | Time Complexity**

**Quick Sort-**

* Quick Sort is a famous sorting algorithm.
* It sorts the given data items in ascending order.
* It uses the idea of divide and conquer approach.
* It follows a recursive algorithm.

**Quick Sort Algorithm-**

Consider-

* a = Linear Array in memory
* beg = Lower bound of the sub array in question
* end = Upper bound of the sub array in question

Then, Quick Sort Algorithm is as follows-

Partition\_Array (a , beg , **end** , loc)

Begin

Set left = beg , right = **end** , loc = beg

Set done = **false**

**While** (not done) **do**

**While** ( (a[loc] <= a[right] ) and (loc ≠ right) ) **do**

Set right = right - 1

**end** **while**

**if** (loc = right) **then**

Set done = **true**

**else** **if** (a[loc] > a[right]) **then**

Interchange a[loc] and a[right]

Set loc = right

**end** **if**

**if** (not done) **then**

**While** ( (a[loc] >= a[left] ) and (loc ≠ left) ) **do**

Set left = left + 1

**end** **while**

**if** (loc = left) **then**

Set done = **true**

**else** **if** (a[loc] < a[left]) **then**

Interchange a[loc] and a[left]

Set loc = left

**end** **if**

**end** **if**

**end** **while**

End

**How Does Quick Sort Works?**

* Quick Sort follows a recursive algorithm.
* It divides the given array into two sections using a partitioning element called as pivot.

The division performed is such that-

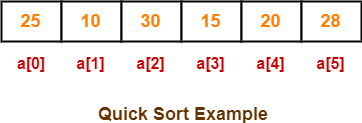
* All the elements to the left side of pivot are smaller than pivot.
* All the elements to the right side of pivot are greater than pivot.

After dividing the array into two sections, the pivot is set at its correct position.

Then, sub arrays are sorted separately by applying quick sort algorithm recursively.

**Quick Sort Example-**

Consider the following array has to be sorted in ascending order using quick sort algorithm-



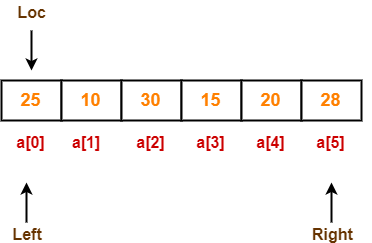
Quick Sort Algorithm works in the following steps-

**Step-01:**

Initially-

* **Left** and **Loc** (pivot) points to the first element of the array.
* **Right** points to the last element of the array.

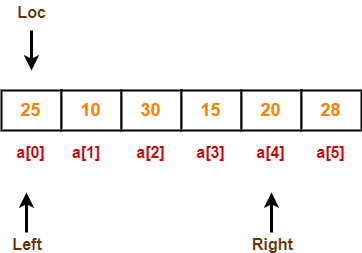
So to begin with, we set **loc** = 0, **left** = 0 and **right** = 5 as-



**Step-02:**

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] < a[right], so algorithm moves **right** one position towards left as-

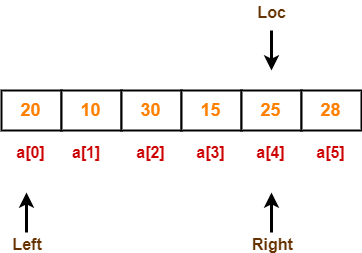


Now, **loc** = 0, **left** = 0 and **right** = 4.

**Step-03:**

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] > a[right], so algorithm swaps a[loc] and a[right] and **loc** points at **right** as-

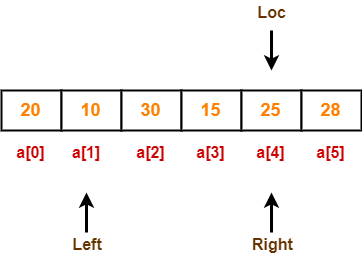


Now, **loc** = 4, **left** = 0 and **right** = 4.

**Step-04:**

Since **loc** points at **right**, so algorithm starts from **left** and move towards right.

As a[loc] > a[left], so algorithm moves **left** one position towards right as-

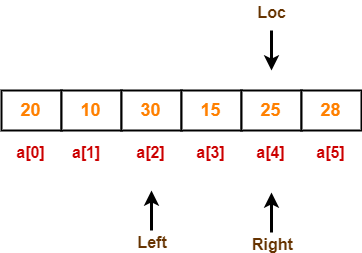


Now, **loc** = 4, **left** = 1 and **right** = 4.

**Step-05:**

Since **loc** points at right, so algorithm starts from **left** and move towards right.

As a[loc] > a[left], so algorithm moves **left** one position towards right as-

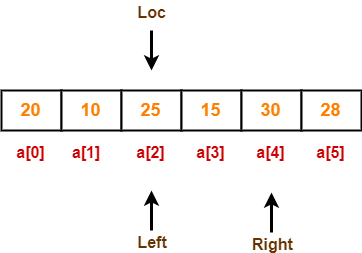


Now, **loc** = 4, **left** = 2 and **right** = 4.

**Step-06:**

Since **loc** points at **right**, so algorithm starts from **left** and move towards right.

As a[loc] < a[left], so we algorithm swaps a[loc] and a[left] and **loc** points at **left** as-

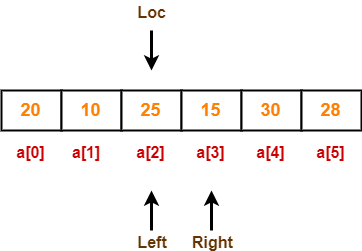


Now, **loc** = 2, **left** = 2 and **right** = 4.

**Step-07:**

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] < a[right], so algorithm moves **right** one position towards left as-

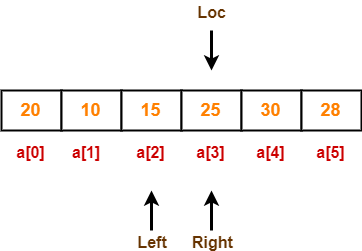


Now, **loc** = 2, **left** = 2 and **right** = 3.

**Step-08:**

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] > a[right], so algorithm swaps a[loc] and a[right] and **loc** points at **right** as-

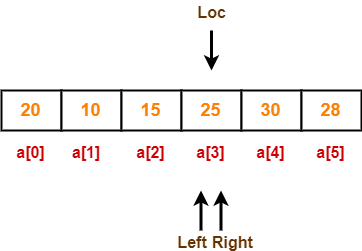


Now, **loc** = 3, **left** = 2 and **right** = 3.

**Step-09:**

Since **loc** points at **right**, so algorithm starts from **left** and move towards right.

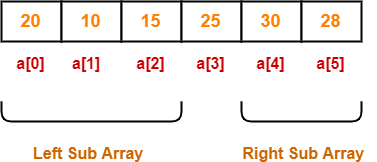
As a[loc] > a[left], so algorithm moves**left** one position towards right as-



Now, **loc** = 3, **left** = 3 and **right** = 3.

Now,

* **loc**, **left** and **right** points at the same element.
* This indicates the termination of procedure.
* The pivot element 25 is placed in its final position.
* All elements to the right side of element 25 are greater than it.
* All elements to the left side of element 25 are smaller than it.



Now, quick sort algorithm is applied on the left and right sub arrays separately in the similar manner.

**Quick Sort Analysis-**

* To find the location of an element that splits the array into two parts, O(n) operations are required.
* This is because every element in the array is compared to the partitioning element.
* After the division, each section is examined separately.
* If the array is split approximately in half (which is not usually), then there will be log2n splits.
* Therefore, total comparisons required are f(n) = n x log2n = O(nlog2n).

|  |
| --- |
| Order of Quick Sort = O(nlog2n) |

**Worst Case-**

* Quick Sort is sensitive to the order of input data.
* It gives the worst performance when elements are already in the ascending order.
* It then divides the array into sections of 1 and (n-1) elements in each call.
* Then, there are (n-1) divisions in all.
* Therefore, here total comparisons required are f(n) = n x (n-1) = O(n2).

|  |
| --- |
| Order of Quick Sort in worst case = O(n2) |

**Advantages of Quick Sort-**

The advantages of quick sort algorithm are-

* Quick Sort is an in-place sort, so it requires no temporary memory.
* Quick Sort is typically faster than other algorithms.

(because its inner loop can be efficiently implemented on most architectures)

* Quick Sort tends to make excellent usage of the memory hierarchy like virtual memory or caches.
* Quick Sort can be easily parallelized due to its divide and conquer nature.

**Disadvantages of Quick Sort-**

The disadvantages of quick sort algorithm are-

* The worst case complexity of quick sort is O(n2).
* This complexity is worse than O(nlogn) worst case complexity of algorithms like merge sort, heap sort etc.
* It is not a stable sort i.e. the order of equal elements may not be preserved.

MAY 20, 2022 FRIDAY

[**Topological Sort | Topological Sort Examples**](https://www.gatevidyalay.com/topological-sort-topological-sorting/)

**Topological Sort-**

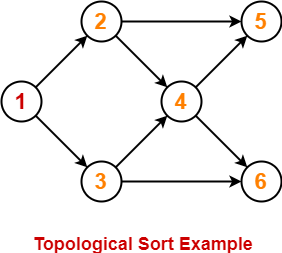
|  |
| --- |
| Topological Sort is a linear ordering of the vertices in such a way that  if there is an edge in the DAG going from vertex ‘u’ to vertex ‘v’,  then ‘u’ comes before ‘v’ in the ordering. |

It is important to note that-

* Topological Sorting is possible if and only if the graph is a [**Directed Acyclic Graph**](https://www.gatevidyalay.com/directed-acyclic-graphs/).
* There may exist multiple different topological orderings for a given directed acyclic graph.

**Topological Sort Example-**

Consider the following directed acyclic graph-



* 1. 1 2 4 5

For this graph, following 4 different topological orderings are possible-

* **1 2 3 4 5 6**
* **1 2 3 4 6 5**
* **1 3 2 4 5 6**
* **1 3 2 4 6 5**

**Applications of Topological Sort-**

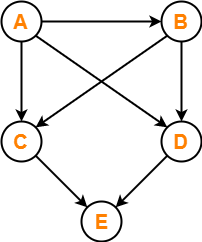
Few important applications of topological sort are-

* Scheduling jobs from the given dependencies among jobs
* Instruction Scheduling
* Determining the order of compilation tasks to perform in makefiles
* Data Serialization

**PRACTICE PROBLEMS BASED ON TOPOLOGICAL SORT-**

**Problem-01:**

Find the number of different topological orderings possible for the given graph-

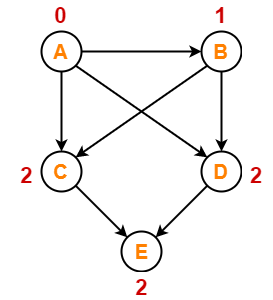


**Solution-**

The topological orderings of the above graph are found in the following steps-

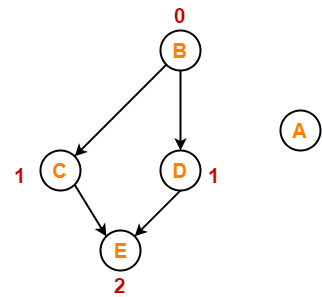
**Step-01:**

Write in-degree of each vertex-



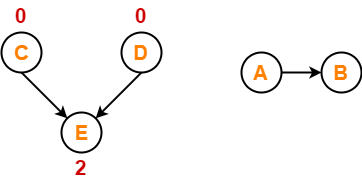
**Step-02:**

* Vertex-A has the least in-degree.
* So, remove vertex-A and its associated edges.
* Now, update the in-degree of other vertices.



**Step-03:**

* Vertex-B has the least in-degree.
* So, remove vertex-B and its associated edges.
* Now, update the in-degree of other vertices.



**Step-04:**

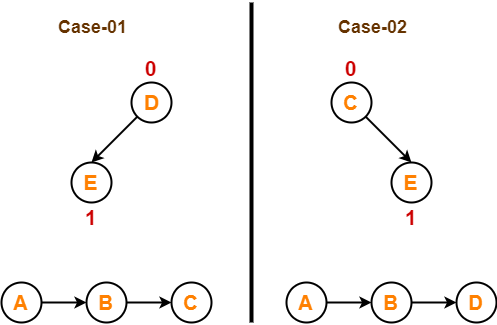
There are two vertices with the least in-degree. So, following 2 cases are possible-

In case-01,

* Remove vertex-C and its associated edges.
* Then, update the in-degree of other vertices.

In case-02,

* Remove vertex-D and its associated edges.
* Then, update the in-degree of other vertices.



**Step-05:**

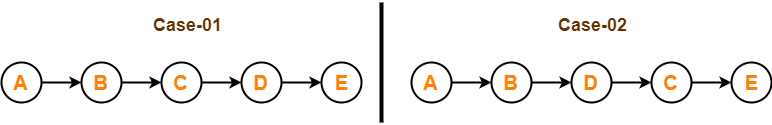
Now, the above two cases are continued separately in the similar manner.

In case-01,

* Remove vertex-D since it has the least in-degree.
* Then, remove the remaining vertex-E.

In case-02,

* Remove vertex-C since it has the least in-degree.
* Then, remove the remaining vertex-E.



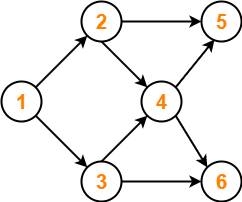
**Conclusion-**

For the given graph, following **2** different topological orderings are possible-

* **A B C D E**
* **A B D C E**

**Problem-02:**

Find the number of different topological orderings possible for the given graph-

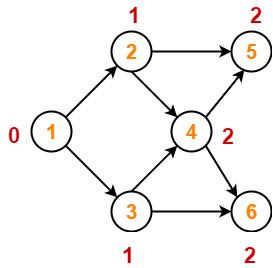


**Solution-**

The topological orderings of the above graph are found in the following steps-

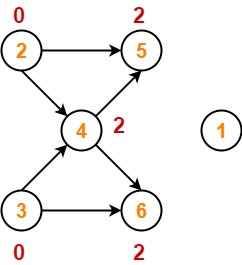
**Step-01:**

Write in-degree of each vertex-



**Step-02:**

* Vertex-1 has the least in-degree.
* So, remove vertex-1 and its associated edges.
* Now, update the in-degree of other vertices.



**Step-03:**

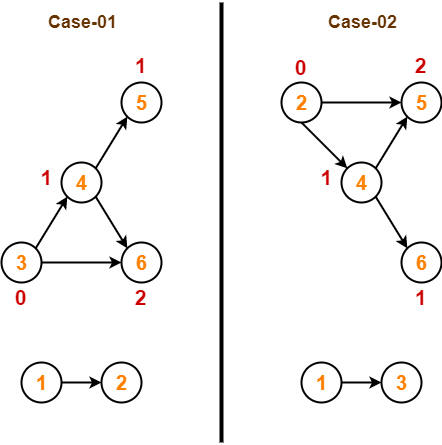
There are two vertices with the least in-degree. So, following 2 cases are possible-

In case-01,

* Remove vertex-2 and its associated edges.
* Then, update the in-degree of other vertices.

In case-02,

* Remove vertex-3 and its associated edges.
* Then, update the in-degree of other vertices.



**Step-04:**

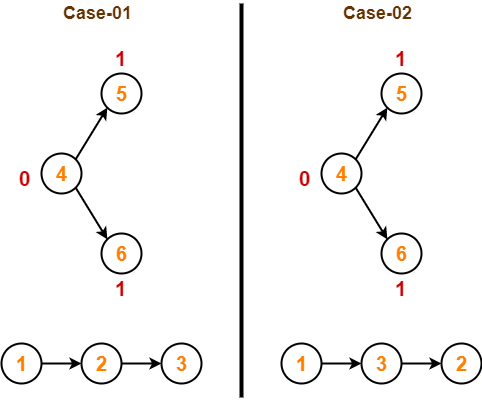
Now, the above two cases are continued separately in the similar manner.

In case-01,

* Remove vertex-3 since it has the least in-degree.
* Then, update the in-degree of other vertices.

In case-02,

* Remove vertex-2 since it has the least in-degree.
* Then, update the in-degree of other vertices.



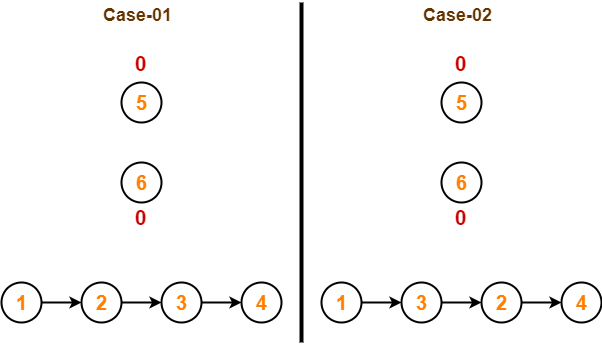
**Step-05:**

In case-01,

* Remove vertex-4 since it has the least in-degree.
* Then, update the in-degree of other vertices.

In case-02,

* Remove vertex-4 since it has the least in-degree.
* Then, update the in-degree of other vertices.

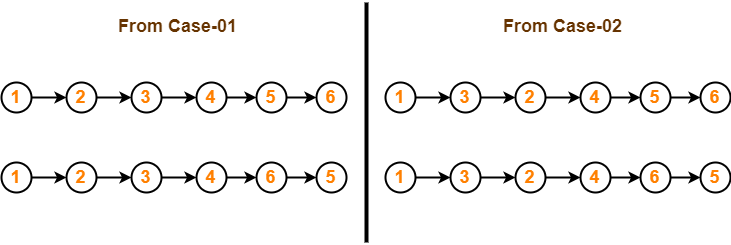


**Step-06:**

In case-01,

* There are 2 vertices with the least in-degree.
* So, 2 cases are possible.
* Any of the two vertices may be taken first.

Same is with case-02.



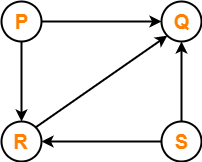
**Conclusion-**

For the given graph, following **4** different topological orderings are possible-

* **1 2 3 4 5 6**
* **1 2 3 4 6 5**
* **1 3 2 4 5 6**
* **1 3 2 4 6 5**

**Problem-03:**

Consider the directed graph given below. Which of the following statements is true?



1. The graph does not have any topological ordering.
2. Both PQRS and SRPQ are topological orderings.
3. Both PSRQ and SPRQ are topological orderings.
4. PSRQ is the only topological ordering.

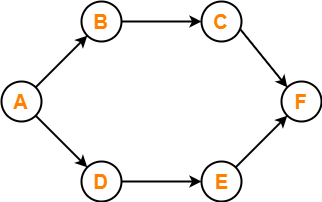
**Solution-**

* The given graph is a directed acyclic graph.
* So, topological orderings exist.
* P and S must appear before R and Q in topological orderings as per the definition of topological sort.

Thus, Correct option is (C).

**Problem-04:**

Consider the following directed graph-



The number of different topological orderings of the vertices of the graph is \_\_\_\_\_\_\_\_ ?

**Solution-**

Number of different topological orderings possible = 6.

Thus, Correct answer is **6**.

**Heap Sort Algorithm**

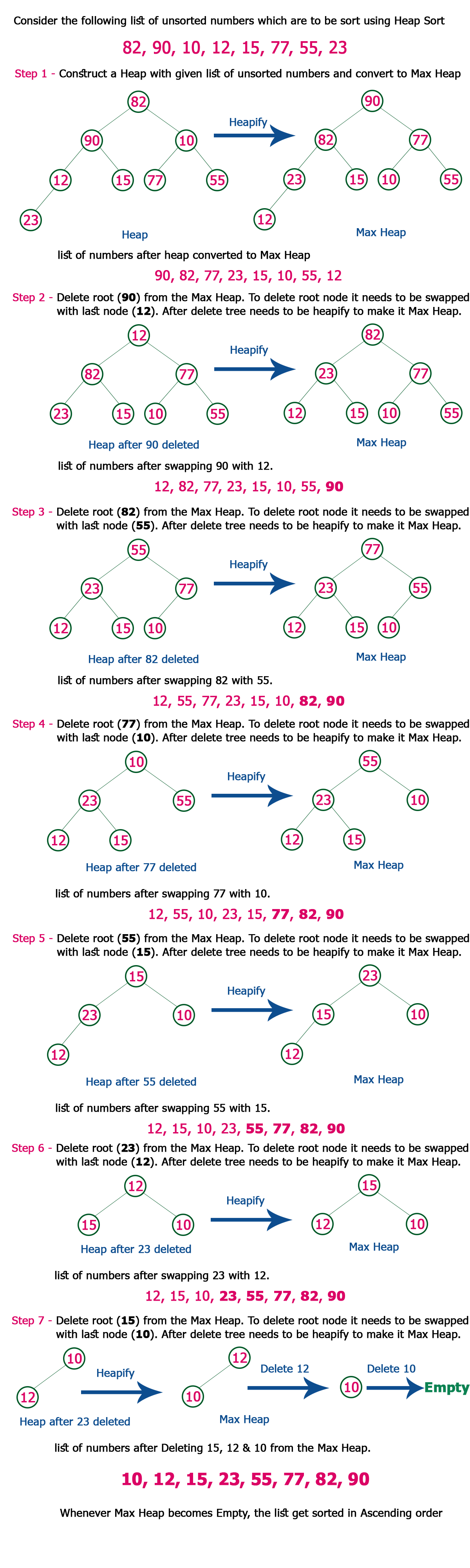
Heap sort is one of the sorting algorithms used to arrange a list of elements in order. Heapsort algorithm uses one of the tree concepts called **Heap Tree**. In this sorting algorithm, we use **Max Heap** to arrange list of elements in Descending order and **Min Heap** to arrange list elements in Ascending order.

**Step by Step Process**

The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

* **Step 1 -**Construct a **Binary Tree** with given list of Elements.
* **Step 2 -**Transform the Binary Tree into **Min Heap.**
* **Step 3 -**Delete the root element from Min Heap using **Heapify** method.
* **Step 4 -**Put the deleted element into the Sorted list.
* **Step 5 -**Repeat the same until Min Heap becomes empty.
* **Step 6 -**Display the sorted list.

**Example**



**Complexity of the Heap Sort Algorithm**

To sort an unsorted list with **'n'** number of elements, following are the complexities...

**Worst Case : O(n log n)**  
**Best Case : O(n log n)**  
**Average Case : O(n log n)**

## ****Heapsort Algorithm****

Heapsort(arr)

buildMaxHeap(arr)

for (int i = n - 1; i >= 0; i--) {

swap(&arr[0], &arr[i]);

heapsize--;

maxHeapify(arr,0);

}

## ****Heapsort Algorithm Dry Run****

input:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 23 | 10 | 16 | 11 | 20 |

The first step – call build max heap

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 23 | 20 | 16 | 11 | 10 |

For i=4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 20 | 11 | 16 | 10 | 23 |

After i=3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 16 | 11 | 10 | 20 | 23 |

After i=2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 11 | 10 | 16 | 20 | 23 |

After i=1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 10 | 11 | 16 | 20 | 23 |

After i=0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 10 | 11 | 16 | 20 | 23 |

## ****Heapsort in C****

#include <stdio.h>

void swap(int \*a, int \*b) {

int tmp = \*a;

\*a = \*b;

\*b = tmp;

}

void heapify(int arr[], int n, int i) {

int max = i;

int leftChild = 2 \* i + 1;

int rightChild = 2 \* i + 2;

if (leftChild < n && arr[leftChild] > arr[max])

max = leftChild;

if (rightChild < n && arr[rightChild] > arr[max])

max = rightChild;

if (max != i) {

swap(&arr[i], &arr[max]);

heapify(arr, n, max);

}

}

void heapSort(int arr[], int n) {

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

for (int i = n - 1; i >= 0; i--) {

swap(&arr[0], &arr[i]);

heapify(arr, i, 0);

}

}

void display(int arr[], int n) {

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

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

printf("\n");

}

int main() {

int arr[] = {11, 34, 9, 5, 16, 10};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array:\n");

display(arr, n);

heapSort(arr, n);

printf("Sorted array:\n");

display(arr, n);

}

Output of the program:

Original array:

11 34 9 5 16 10

Sorted array:

5 9 10 11 16 34

Red Black Tree

A Red Black Tree is a category of the self-balancing binary search tree. It was created in 1972 by Rudolf Bayer who termed them **"symmetric binary B-trees**."

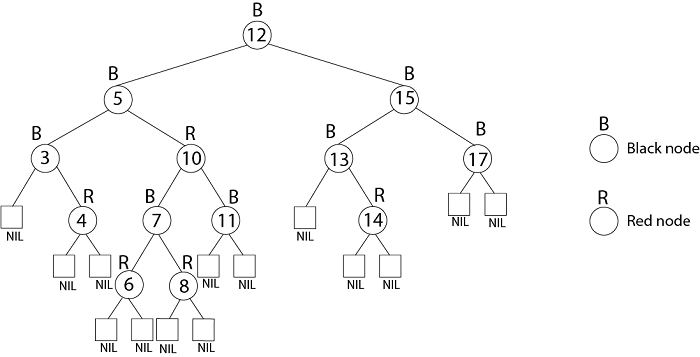
A red-black tree is a Binary tree where a particular node has color as an extra attribute, either red or black. By check the node colors on any simple path from the root to a leaf, red-black trees secure that no such path is higher than twice as long as any other so that the tree is generally balanced.

Properties of Red-Black Trees

A red-black tree must satisfy these properties:

1. The root is always black.
2. A nil is recognized to be black. This factor that every non-NIL node has two children.
3. **Black Children Rule:** The children of any red node are black.
4. **Black Height Rule:** For particular node v, there exists an integer bh (v) such that specific downward path from v to a nil has correctly bh (v) black real (i.e. non-nil) nodes. Call this portion the black height of v. We determine the black height of an RB tree to be the black height of its root.

A tree T is an almost red-black tree (ARB tree) if the root is red, but other conditions above hold.

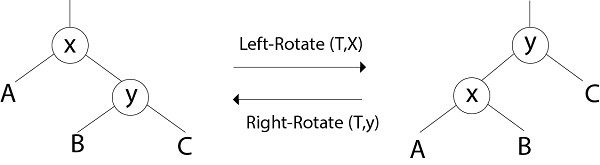


Operations on RB Trees:

The search-tree operations TREE-INSERT and TREE-DELETE, when runs on a red-black tree with n keys, take O (log n) time. Because they customize the tree, the conclusion may violate the red-black properties. To restore these properties, we must change the color of some of the nodes in the tree and also change the pointer structure.

1. Rotation:

Restructuring operations on red-black trees can generally be expressed more clearly in details of the rotation operation.



Clearly, the order (Ax By C) is preserved by the rotation operation. Therefore, if we start with a BST and only restructure using rotation, then we will still have a BST i.e. rotation do not break the BST-Property.

**LEFT ROTATE (T, x)**

1. y ← right [x]

2. right [x] ← left [y]

3. p [left[y]] ← x

4. p[y] ← p[x]

5. If p[x] = nil [T]

then root [T] ← y

else if x = left [p[x]]

then left [p[x]] ← y

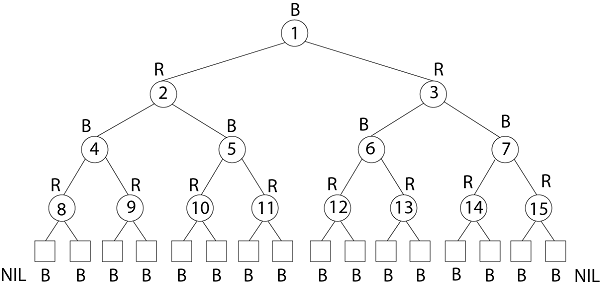
else right [p[x]] ← y

6. left [y] ← x.

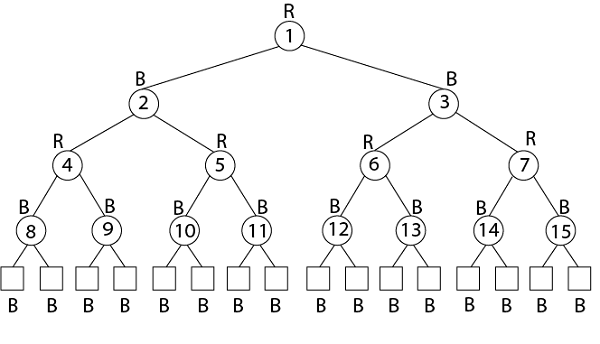
7. p [x] ← y.

**Example:** Draw the complete binary tree of height 3 on the keys {1, 2, 3... 15}. Add the NIL leaves and color the nodes in three different ways such that the black heights of the resulting trees are: 2, 3 and 4.

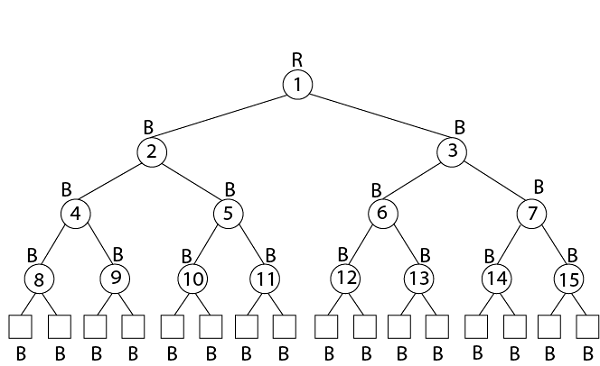
**Solution:**



**Tree with black-height-2**



Tree with black-height-3



Tree with black-height-4

2. Insertion:

* Insert the new node the way it is done in Binary Search Trees.
* Color the node red
* If an inconsistency arises for the red-black tree, fix the tree according to the type of discrepancy.

A discrepancy can decision from a parent and a child both having a red color. This type of discrepancy is determined by the location of the node concerning grandparent, and the color of the sibling of the parent.

**RB-INSERT (T, z)**

1. y ← nil [T]

2. x ← root [T]

3. while x ≠ NIL [T]

4. do y ← x

5. if key [z] < key [x]

6. then x ← left [x]

7. else x ← right [x]

8. p [z] ← y

9. if y = nil [T]

10. then root [T] ← z

11. else if key [z] < key [y]

12. then left [y] ← z

13. else right [y] ← z

14. left [z] ← nil [T]

15. right [z] ← nil [T]

16. color [z] ← RED

17. RB-INSERT-FIXUP (T, z)

After the insert new node, Coloring this new node into black may violate the black-height conditions and coloring this new node into red may violate coloring conditions i.e. root is black and red node has no red children. We know the black-height violations are hard. So we color the node red. After this, if there is any color violation, then we have to correct them by an RB-INSERT-FIXUP procedure.

**RB-INSERT-FIXUP (T, z)**

1. while color [p[z]] = RED

2. do if p [z] = left [p[p[z]]]

3. then y ← right [p[p[z]]]

4. If color [y] = RED

5. then color [p[z]] ← BLACK //Case 1

6. color [y] ← BLACK //Case 1

7. color [p[z]] ← RED //Case 1

8. z ← p[p[z]] //Case 1

9. else if z= right [p[z]]

10. then z ← p [z] //Case 2

11. LEFT-ROTATE (T, z) //Case 2

12. color [p[z]] ← BLACK //Case 3

13. color [p [p[z]]] ← RED //Case 3

14. RIGHT-ROTATE (T,p [p[z]]) //Case 3

15. else (same as then clause)

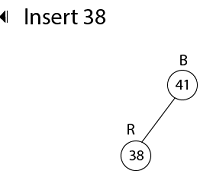
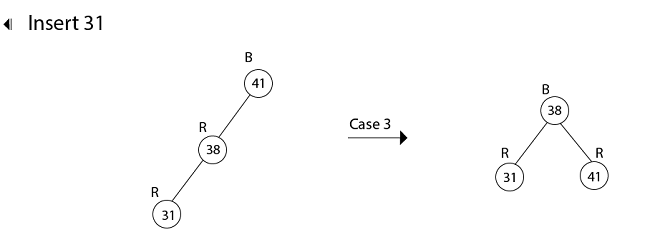
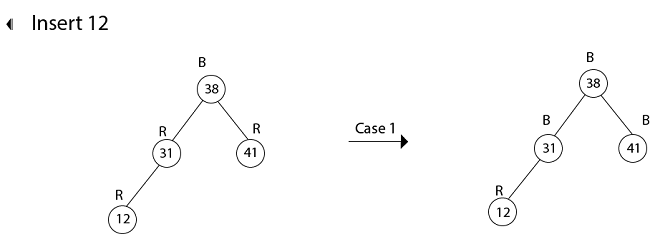
With "right" and "left" exchanged

16. color [root[T]] ← BLACK

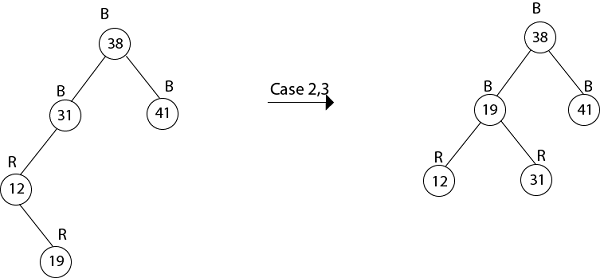
**Example:** Show the red-black trees that result after successively inserting the keys 41,38,31,12,19,8 into an initially empty red-black tree.

**Solution:**

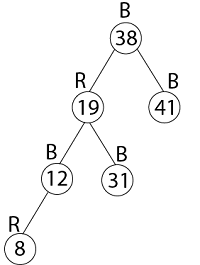
Insert 41

DAA Red Black Tree  
  
  


**Insert 19**


Thus the final tree is



3. Deletion:

First, search for an element to be deleted

* If the element to be deleted is in a node with only left child, swap this node with one containing the largest element in the left subtree. (This node has no right child).
* If the element to be deleted is in a node with only right child, swap this node with the one containing the smallest element in the right subtree (This node has no left child).
* If the element to be deleted is in a node with both a left child and a right child, then swap in any of the above two ways. While swapping, swap only the keys but not the colors.
* The item to be deleted is now having only a left child or only a right child. Replace this node with its sole child. This may violate red constraints or black constraint. Violation of red constraints can be easily fixed.
* If the deleted node is black, the black constraint is violated. The elimination of a black node y causes any path that contained y to have one fewer black node.
* Two cases arise:
  + The replacing node is red, in which case we merely color it black to make up for the loss of one black node.
  + The replacing node is black.

The strategy RB-DELETE is a minor change of the TREE-DELETE procedure. After splicing out a node, it calls an auxiliary procedure RB-DELETE-FIXUP that changes colors and performs rotation to restore the red-black properties.

**RB-DELETE (T, z)**

1. if left [z] = nil [T] or right [z] = nil [T]

2. then y ← z

3. else y ← TREE-SUCCESSOR (z)

4. if left [y] ≠ nil [T]

5. then x ← left [y]

6. else x ← right [y]

7. p [x] ← p [y]

8. if p[y] = nil [T]

9. then root [T] ← x

10. else if y = left [p[y]]

11. then left [p[y]] ← x

12. else right [p[y]] ← x

13. if y≠ z

14. then key [z] ← key [y]

15. copy y's satellite data into z

16. if color [y] = BLACK

17. then RB-delete-FIXUP (T, x)

18. return y

**RB-DELETE-FIXUP (T, x)**

1. while x ≠ root [T] and color [x] = BLACK

2. do if x = left [p[x]]

3. then w ← right [p[x]]

4. if color [w] = RED

5. then color [w] ← BLACK //Case 1

6. color [p[x]] ← RED //Case 1

7. LEFT-ROTATE (T, p [x]) //Case 1

8. w ← right [p[x]] //Case 1

9. If color [left [w]] = BLACK and color [right[w]] = BLACK

10. then color [w] ← RED //Case 2

11. x ← p[x] //Case 2

12. else if color [right [w]] = BLACK

13. then color [left[w]] ← BLACK //Case 3

14. color [w] ← RED //Case 3

15. RIGHT-ROTATE (T, w) //Case 3

16. w ← right [p[x]] //Case 3

17. color [w] ← color [p[x]] //Case 4

18. color p[x] ← BLACK //Case 4

19. color [right [w]] ← BLACK //Case 4

20. LEFT-ROTATE (T, p [x]) //Case 4

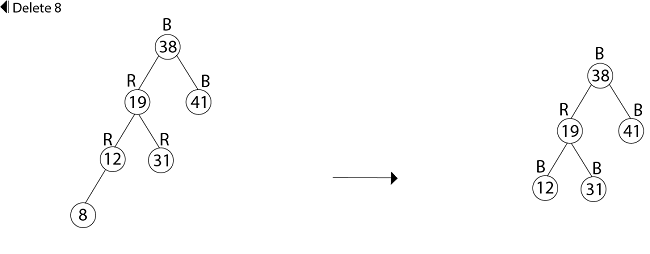
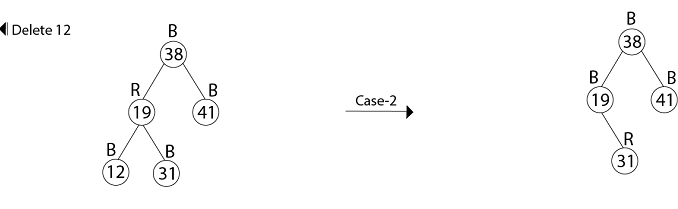
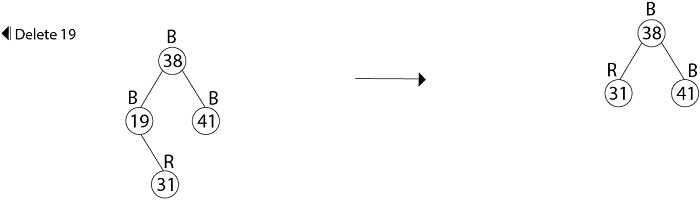
21. x ← root [T] //Case 4

22. else (same as then clause with "right" and "left" exchanged)

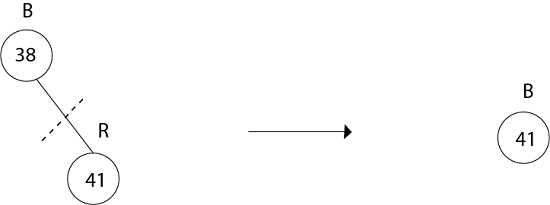
23. color [x] ← BLACK

**Example:** In a previous example, we found that the red-black tree that results from successively inserting the keys 41,38,31,12,19,8 into an initially empty tree. Now show the red-black trees that result from the successful deletion of the keys in the order 8, 12, 19,31,38,41.

**Solution:**


**Delete 38**



**Delete 41**

No Tree.

RED-BLACK TREEs A red-black tree is a self-balancing binary search tree that was invented in 1972 by Rudolf Bayer who called it the ‘symmetric binary B-tree’. Although a red-black tree is complex, it has good worst-case running time for its operations and is efficient to use as searching, insertion, and deletion can all be done in O(log n) time, where n is the number of nodes in the tree. Practically, a red-black tree is a binary search tree which inserts and removes intelligently, to keep the tree reasonably balanced. A special point to note about the red-black tree is that in this tree, no data is stored in the leaf nodes.

10.5.1 Properties of Red-Black Trees A red-black tree is a binary search tree in which every node has a colour which is either red or black. Apart from the other restrictions of a binary search tree, the red-black tree has the following additional requirements: 1. The colour of a node is either red or black. 2. The colour of the root node is always black. 3. All leaf nodes are black. 4. Every red node has both the children coloured in black. 5. Every simple path from a given node to any of its leaf nodes has an equal number of black nodes. Look at Fig. 10.55 which shows a red-black tree. 16 9 27 7 11 21 45 11 NULL NULL NULL NULL 36 63 NULL NULL NULL NULL NUL