Sorting Merging and Searching Algorithms

Contents

[**Quick Sort in Ascending Order** 2](#_Toc78704575)

[**Quick Sort in Descending Order** 3](#_Toc78704576)

[**Order and Space Complexity Table** 4](#_Toc78704577)

[**Merge Sort** 5](#_Toc78704578)

[**Merge two already sorted arrays sot that the resulting array, ie third array will be sorted one** 6](#_Toc78704579)

[**Merge two arrays one is small and other big enough to accommodate the small one, the resulting array will be sorted** 7](#_Toc78704580)

[**Insertion Sort** 9](#_Toc78704581)

[**Selection Sort** 10](#_Toc78704582)

[**Binary Search** 11](#_Toc78704583)

[**Binary Search Tree** 12](#_Toc78704584)

[**Traversal** 12](#_Toc78704585)

[**Depth-first** 12](#_Toc78704586)

[**Breadth-First** 13](#_Toc78704587)

# **Quick Sort in Ascending Order**

**Time Complexity: Best Case and Average Case : O(*n* log *n*) , Worst Case : O(*n*2)**

**Space Complexity: Worst Case : O(log *n*)**

*//Sorting in Ascending order*

**public class** QuickSortAscending {  
 **public static void** quickSort(**int**[] a) {  
 ***quickSort*(a, 0 , a.length - 1 );** }  
  
 **public static void** quickSort(**int**[] a , **int** low , **int** high ) {  
  
 **int** i = low;  
 **int** j = high;  
 **int** pivot = a[low+ (high - low )/2 ];  
  
 **while( i <= j ) {   
 while( a[i] < pivot ) i++;  
  
 while( a[j] > pivot ) j--;  
  
 if( i <= j ) {  
 *swap*(a , i , j);   
 i++;  
 j--;  
 }  
 }  
 if( i < high ) {  
 *quickSort*( a, i , high );  
 }  
 if( j > low )  
 *quickSort*( a,low,j );**  
 }  
  
 **public static void** swap(**int**[] a, **int** i , **int** j ) {  
 **int** temp = a[i];  
 a[i] = a[j];  
 a[j] = temp;  
 }  
  
 **public static void** main(String[] args) {  
 **int**[] a = {24, 3, 45, 20, 56, 75, 2, 56, 99, 53, 12};  
 *quickSort*(a);  
 **for**( **int** i : a )  
 System.***out***.print(i + **"\t"**);  
 }  
}

**Descending Order**

**while (a[i] > pivot) i++;  
  
while (a[j] < pivot) j--;**

**Ascending Order**

**while( a[i] < pivot ) i++;  
  
while( a[j] > pivot ) j--;**

output -> 2 3 12 20 24 45 53 56 56 75 99

# **Quick Sort in Descending Order**

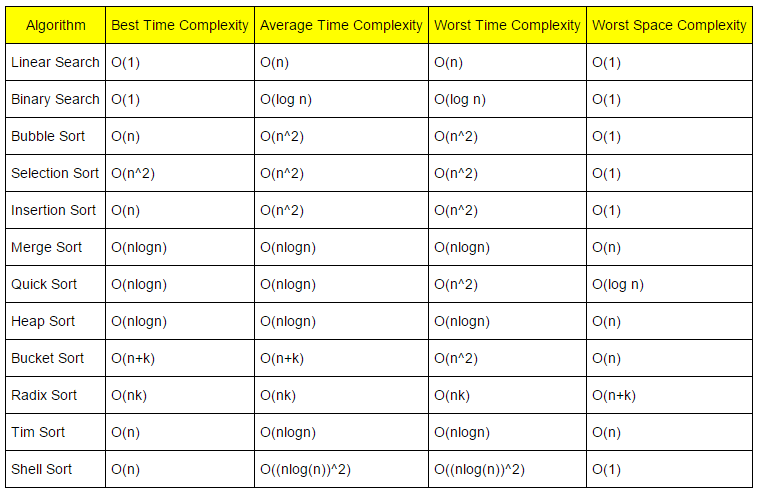
*//Sorting in Descending Order***public class** QuickSortDescending {  
  
 **public static void** quickSort(**int**[] a) {  
  
 *quickSort*(a, 0, a.**length** - 1);  
 }  
  
 **public static void** quickSort(**int**[] a, **int** low, **int** high) {  
  
 **int** i = low;  
 **int** j = high;  
 **int** pivot = a[low + (high - low) / 2];  
  
 **while (i <= j) {  
  
 while (a[i] > pivot) i++;  
  
 while (a[j] < pivot) j--;  
  
 if (i <= j) {  
 *swap*(a, i, j);  
 i++;  
 j--;  
 }  
 }  
 if (i < high) {  
 *quickSort*(a, i, high);  
 }  
 if (j > low)  
 *quickSort*(a, low, j);** }  
  
 **public static void** swap(**int**[] a, **int** i, **int** j) {  
 **int** temp = a[i];  
 a[i] = a[j];  
 a[j] = temp;  
 }  
  
 **public static void** main(String[] args) {  
 **int**[] a = {24, 3, 45, 20, 56, 75, 2, 56, 99, 53, 12};  
 *quickSort*(a);  
 **for** (**int** i : a)  
 System.***out***.print(i + **"\t"**);  
 }  
}

output -> 99 75 56 56 53 45 24 20 12 3 2

**To find the Nth Greatest or Smallest, sort the array using Quick sort and find the index of the array as shown below.**

**public static int getNthSmallest(int[] a , int n) {  
 *quickSort*(a);  
 return a[n];  
}**

# **Order and Space Complexity Table**



# **Merge Sort**

**Time Complexity: Best, Worst, Average Case : O(n log n)**

**Space Complexity : O(n log n)**

**Sorting in Ascending Order**

**else if( a1[j] < a2[k]) a[i] = a1[j++];**

**import** java.util.Arrays;

**public** **class** MergeSort {

**Sorting in Descending Order**

**else if( a1[j] < a2[k]) a[i] = a1[j++];**

**public** **static** **void** mergeSort(**int**[] a) {

**if**(a.length <= 1) **return**;

**int**[] a1 = Arrays.*copyOfRange*(a, 0, a.length/2);

**int**[] a2 = Arrays.*copyOfRange*(a, a.length/2, a.length);

*mergeSort*(a1);

*mergeSort*(a2);

*merge*(a1, a2, a);

}

**public** **static** **void** merge(**int**[] a1, **int**[] a2, **int**[] a) {

**int** j = 0, k = 0;

**for**(**int** i = 0 ; i < a.length ; i++) {

**if(j == a1.length) a[i] = a2[k++];**

else if(k == a2.length) a[i] = a1[j++];

**else if( a1[j] < a2[k]) a[i] = a1[j++];** // Sorting in Ascending order

// **else if( a1[j] > a2[k]) a[i] = a1[j++];** // Sorting in Descending order

**else a[i] = a2[k++];**

}

}

**public** **static** **void** main(String...args) {

**int**[] a = **new** **int**[] {5,7,1,2,3,4,11};

**for**(**int** i : a) System.***out***.print(i+"\t");

*mergeSort*(a);

System.***out***.println("\n"+"After sorting"+"\n");

**for**(**int** i : a) System.***out***.print(i+"\t");

}

}

**Generic Mergesort**

**<T> void merge(T[] a0, T[] a1, T[] a, Comparator<T> c) {**

**{**

**int i0 = 0, i1 = 0;**

**for (int i = 0; i < a.length; i++) {**

**if (i0 == a0.length)**

**a[i] = a1[i1++];**

**else if (i1 == a1.length)**

**a[i] = a0[i0++];**

**else if (compare(a0[i0], a1[i1]) < 0)**

**a[i] = a0[i0++];**

**else**

**a[i] = a1[i1++];**

**}**

**}**

**<T> void mergeSort(T[] a, Comparator<T> c)**

**{**

**if (a.length <= 1) return;**

**T[] a0 = Arrays.copyOfRange(a, 0, a.length/2);**

**T[] a1 = Arrays.copyOfRange(a, a.length/2, a.length);**

**mergeSort(a0, c);**

**mergeSort(a1, c);**

**merge(a0, a1, a, c);**

**}**

# **Merge two already sorted arrays sot that the resulting array, ie third array will be sorted one**

**The techniques used here is merge of Merge Sort**

**public class** MergeTwoSortedArrays1 {  
  
 **public static void** mergeTwoArrays(**int**[] a1 , **int**[] a2 , **int**[] a) {  
  
 **int j = 0, int k = 0;  
  
 for (int i = 0; i < a.length; i++) {  
  
 if( j == a1.length )  
 a[i] = a2[k++];  
 else if( k == a2.length )  
 a[i] = a1[j++];  
 else if(a1[j] < a2[k])  
 a[i] = a1[j++];  
 else  
 a[i] = a2[k++];  
 }**  
 }  
  
 **public static void** main(String[] args) {  
  
 **int**[] a1 = {1,2,3,4,5,5};  
 **int**[] a2 = {7, 8, 9, 10, 14, 17};  
 **int**[] a3 = **new int**[a1.**length** + a2.**length**];  
  
 *mergeTwoArrays*(a1, a2, a3 );  
 **for**( **int** i : a3 )  
 System.***out***.print(i+**"\t"**);  
 }  
}

*//****An alternative technique to merge two arrays***

*//Super technique***public static void** merge(**int**[] a, **int**[] b, **int**[] c) {  
  
 **int** i = a.**length** - 1, j = b.**length** - 1, k = c.**length**;  
  
 **while** (k > 0)  
 c[--k] = (j < 0 || (i >= 0 && a[i] >= b[j])) ? a[i--] : b[j--];  
}

# **Merge two arrays one is small and other big enough to accommodate the small one, the resulting array will be sorted**

It has two different use cases.

1. If the beginning of the big array contains blank spaces like 0.
2. If the end of the big array contains blank spaces like 0.

**package** com.ddlab.rnd.algol;  
*/\*\*  
 \* If the beginning of bigger array has empty spaces like 0.  
 \* int[] b = {0,0,0,0,0,3,4,5,11,34};  
 \*/***public class** MergeTwoSortedArraysCase1 {  
  
 **public static void** merge(**int**[] a , **int**[] b) {  
 *//b[] is bigger* **int** j = 0 ;  
 **for** (**int** i = 0; i < b.**length** && j < a.**length**; i++) {  
 **if**( b[i] == 0 )  
 b[i] = a[j++];  
 }  
  
 *insertionSort*(b);;  
 }  
  
 **public static void** insertionSort( **int**[] a ) {  
 **for**( **int** i = 0 ; i < a.**length** ; i++ ) {  
 **for**( **int** j = i ; j > 0 && a[j] < a[j-1] ; j--)  
 *swap*(a, j , j-1);  
 }  
 }  
  
 **private static void** swap(**int**[] a , **int** i , **int** j ) {  
 **int** temp = a[i];  
 a[i] = a[j];  
 a[j] = temp;  
 }  
  
 **public static void** main(String[] args) {  
 **int**[] a = {1, 2, 5, 8, 9};  
 **int**[] b = {0,0,0,0,0,3,4,5,11,34};  
  
 *merge*(a,b);  
  
 **for**( **int** i : b )  
 System.***out***.print(i+**"\t"**);  
  
 }  
}

*/\*\*  
 \* If the end of the bigger array contains empty spaces like 0  
 \* int[] b = {3,4,5,11,34,0,0,0,0,0};  
 \*/***public class** MergeTwoSortedArraysCase2 {  
  
 **public static void** merge(**int**[] a, **int**[] b) {  
  
 **int** i = a.**length** - 1;  
 **int** j = b.**length** - a.**length** - 1;  
 **int** k = b.**length** - 1;  
  
 **while** (i >= 0 && j >= 0 && k >= 0) {  
 **if** (a[i] > b[j])  
 b[k--] = a[i--];  
  
 **else** b[k--] = b[j--];  
 }  
  
 *//Check the left over from small array* **while** (i >= 0)  
 b[k--] = a[i--];  
  
 *//Check the left over from the big array* **while** (j >= 0)  
 b[k--] = b[j--];  
 }  
  
 **public static void** main(String[] args) {  
 **int**[] a = {1, 2, 5, 8, 9};  
 **int**[] b = {3,4,5,11,34,0,0,0,0,0};  
  
 *merge*(a,b);  
  
 **for**( **int** i : b )  
 System.***out***.print(i+**"\t"**);  
 }  
}

# **Insertion Sort**

**Best Case: O(n), Worst Case and Average Case : O(*n*2)**

**The best case input is an array that is already sorted.**

**Why Insertion Sort is used in Java ?**

<http://stackoverflow.com/questions/3566843/why-does-java-util-arrays-sortobject-use-2-kinds-of-sorting-algorithms>

It's important to note that an algorithm that is O(Nlog N) is not always faster in practice than an O(N^2) algorithm. It depends on the constants, and the range of N involved. Remember that [asymptotic notation](http://en.wikipedia.org/wiki/Big_O_notation) measures relative growth rate, not absolute speed**). For small N, insertion sort in fact does beat merge sort**. It's also faster for almost-sorted arrays.

Although it is one of the elementary sorting algorithms with O(N^2) worst-case time, insertion sort is the algorithm of choice either **when the data is nearly sorted** (because it is adaptive) or **when the problem size is small** (because it has low overhead).

**Insertion sort is best for small or very nearly sorted lists . Insertion sort works well on small or nearly sorted lists.**

**public class** InsertionSort {  
   
 **private static void** insertionSort1(**int** a[]) {   
 **for** (**int** i = 0; i < a.**length**; i++) {  
 **for** (**int** j = i; j > 0 && a[j] < a[j - 1]; j--)  
 *swap*(a, j, j - 1);  
 }  
 }  
  
 **private static void** swap(**int**[] a, **int** i, **int** j) {  
 **int** temp = a[i];  
 a[i] = a[j];  
 a[j] = temp;  
 }  
  
 **public static void** main(String[] args) {  
 **int**[] a = {-1, 67, 8, -1, 1, 3, 4, 10, 05, 07, 9};  
**for** (**int** k : a) System.***out***.print(k + **" "**);  
 System.***out***.println(**"\n"**); *insertionSort1*(a);  
 **for** (**int** k : a)  
 System.***out***.print(k + **" "**);  
 }  
}

*Ascending Order*

**public** **static** **void** insertionSort(**int**[] a) {

**for**(**int** i = 0; i < a.length; i++) {

**for**(**int** j = i; j > 0 && a[j] < a[j-1]; j--)

*swap*(a, j, j-1);

}

}

*Descending Order*

**public** **static** **void** insertionSort(**int**[] a) {

**for**(**int** i = 0; i < a.length; i++) {

**for**(**int** j = i; j > 0 && a[j] > a[j-1]; j--)

*swap*(a, j, j-1);

}

}

# **Selection Sort**

**Best case** : O(n), **Average case**: O(n2) , **Worst case** : O(n2 )

In computer science, a selection sort is a sorting algorithm, specifically an in-place comparison sort. It has O(n2) time complexity, making it inefficient on large lists. It works on the principle that : **first find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted**.

**Best case** : O(n). It occurs when the data is in sorted order. After making one pass through the data and making no insertions, insertion sort exits.

**Average case**: O(n2) since there is a wide variation with the running time.

**Worst case** : O(n2 ) if the numbers were sorted in reverse order. The worst case occurs if the array is already sorted in descending order

**public** **class** SelectionSort {

**public** **static** **void** selectionSort(**int**[] a) {

**for**(**int** i = 0; i < a.length; i++) {

**int** minIndex = i;

**for**(**int** j = i; j < a.length; j++) {

**if**(a[j] < a[minIndex]) **// Ascending Order**

// if(a[j] > a[minIndex]) **// Descending Order**

minIndex = j;

}

*swap*(a, i, minIndex);

}

}

**private** **static** **void** swap(**int**[] a, **int** i , **int** j) {

**int** temp = a[i];

a[i] = a[j];

a[j] = temp;

}

**public** **static** **void** main(String[] args) {

**int**[] a = **new** **int**[] {5,7,1,2,3,4,11};

**for**(**int** i : a) System.***out***.print(i+"\t");

*selectionSort*(a);

System.***out***.println("\n"+"After sorting"+"\n");

**for**(**int** i : a) System.***out***.print(i+"\t");

}

}

# **Binary Search**

In computer science, a **binary search or half-interval search algorithm** finds the position of a specified input value (the search "key") within an array sorted by key value. In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index, or position, is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned. A binary search halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time. A binary search is a dichotomic **divide and conquer search algorithm**.

**Worst case performance O(log n)**

**Best case performance O(1)**

**Average case performance O(log n)**

**Worst case space complexity O(1)**

**Best case - O (1) comparisons**

**In the best case, the item X is the middle in the array A. A constant number of comparisions (actually just 1) are required.**

**Worst case - O (log n) comparsions**

**In the worst case, the item X does not exist in the array A at all. Through each recursion or iteration of Binary Search, the size of the admissible range is halved.**

**Average case - O (log n) comparsions**

**To find the average case, take the sum over all elements of the product of number of comparsions required to find each element and the probability of searching for that element.**

**public** **class** BinarySearch {

**public** **static** **boolean** binarySearch(**int**[] a , **int** b ) {

**int** low = 0;

**int** high = a.length - 1;

**while**(low <= high ) {

**int** mid = low + (high - low) / 2;

**if**( b < a[mid] ) high = mid - 1;

**else** **if**( b > a[mid] ) low = mid+1;

**else** **return** **true**;

}

**return** **false**;

}

**public** **static** **int** binarySearch1(**int**[] a , **int** b ) {

**int** low = 0;

**int** high = a.length - 1;

**while**( low <= high ) {

**int** mid = low + (high - low) / 2;

**if**( b < a[mid] ) high = mid - 1;

**else** **if**( b > a[mid] ) low = mid+1;

**else** **return** mid;

}

**return** -(low+1);

}

**public** **static** **void** main(String[] args) {

**int**[] a = {1,5,9,11,22,43,77};

System.*out*.println("Binary Search found : " + *binarySearch*(a, 11));

System.*out*.println("Binary Search found : " + *binarySearch*(a, 31));

}

}

# **Binary Search Tree**

|  |  |  |
| --- | --- | --- |
| Binary search tree  [http://upload.wikimedia.org/wikipedia/commons/thumb/d/da/Binary_search_tree.svg/200px-Binary_search_tree.svg.png](http://en.wikipedia.org/wiki/File:Binary_search_tree.svg) | | |
| [Type](http://en.wikipedia.org/wiki/List_of_data_structures) | [Tree](http://en.wikipedia.org/wiki/Tree_(data_structure)) | |
| [Time complexity](http://en.wikipedia.org/wiki/Time_complexity) in [big O notation](http://en.wikipedia.org/wiki/Big_O_notation) | | |
|  | Average | Worst case |
| Space | O(n) | O(n) |
| Search | O(log n) | O(n) |
| Insert | O(log n) | O(n) |
| Delete | O(log n) | O(n) |

In [computer science](http://en.wikipedia.org/wiki/Computer_science), a **binary search tree** (**BST**), sometimes also called an **ordered** or **sorted binary tree**, is a [node](http://en.wikipedia.org/wiki/Node_(computer_science))-based [binary tree](http://en.wikipedia.org/wiki/Binary_tree) data structure which has the following properties:

* **The left**[**subtree**](http://en.wikipedia.org/wiki/Tree_(data_structure)#Subtree)**of a node contains only nodes with keys less than the node's key.**
* **The right subtree of a node contains only nodes with keys greater than the node's key.**
* **The left and right subtree each must also be a binary search tree.**
* **There must be no duplicate nodes**.

The major advantage of binary search trees over other [data structures](http://en.wikipedia.org/wiki/Data_structure) is that the related [sorting algorithms](http://en.wikipedia.org/wiki/Sorting_algorithm) and [search algorithms](http://en.wikipedia.org/wiki/Search_algorithm) such as [in-order traversal](http://en.wikipedia.org/wiki/In-order_traversal) can be very efficient.

### **Traversal**

There are two types of traversal . **One is Depth-First and Another is Breadth-First.**

**An in-order traversal of a binary search tree will always result in a sorted list of node items.**

### **Depth-first**

There are three types of depth-first traversal: pre-order, in-order, and post-order.For a binary tree, they are defined as operations recursively at each node, starting with the root node follows:

**Pre-order**

public void **preOrderVisit**(Node node) {  
 if (node == null) return;  
 System.*out*.print(node.data + " ");  
 **preOrderVisit(node.left);  
 preOrderVisit(node.right);**}

1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

public void **inOrderVisit**(Node node) {  
 if (node == null) return;  
 **inOrderVisit(node.left);**  
 System.*out*.print(node.data + " ");  
 **inOrderVisit(node.right);**  
}

**In-order (symmetric)**

1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.

public void **postOrderVisit**(Node node) {  
 if (node == null) return;  
 **postOrderVisit(node.left);  
 postOrderVisit(node.right);** System.*out*.print(node.data + " ");  
}

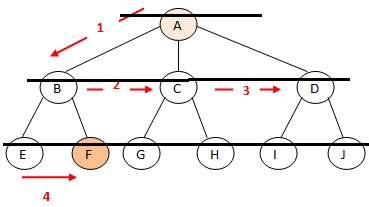
**Post-order**

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.

The trace of a traversal is called a sequentialisation of the tree. No one sequentialisation according to pre-, in- or post-order describes the underlying tree uniquely. Given a tree with distinct elements, either pre-order or post-order paired with in-order is sufficient to describe the tree uniquely. However, pre-order with post-order leaves some ambiguity in the tree structure

### **Breadth-First**

There is only one Breadth-First traversal. The traversal visits nodes by level from top to bottom and from left to right.



* A binary tree is ***height balanced*** (or ***balanced***), if the height of any node’s right subtree differs from the height of the node’s left subtree by no more than 1.

**Perfect binary tree = a binary tree where each level contains the maximum number of nodes every level is completely full of nodes**

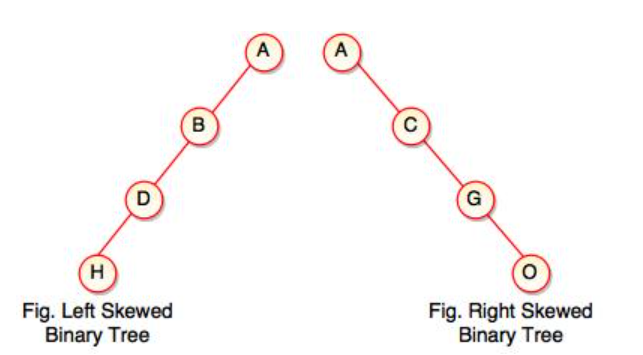
**To understand Binary Search Tree with Visualization**

<https://www.gatevidyalay.com/binary-search-trees-data-structures/>

<https://www.cs.usfca.edu/~galles/visualization/BST.html>

**Skewed Binary Tree**

**If a tree which is dominated by left child node or right child node**, is said to be a Skewed Binary Tree. In a skewed binary tree, all nodes except one have only one child node. The remaining node has no child.



* In a left skewed tree, most of the nodes have the left child without corresponding right child.
* In a right skewed tree, most of the nodes have the right child without corresponding left child

**Usages of Pre-Order, In-Order and Post-Order**

**Pre-order:** Used to create a copy of a tree. For example, if you want to create a replica of a tree, put the nodes in an array with a pre-order traversal. Then perform an *Insert* operation on a new tree for each value in the array. You will end up with a copy of your original tree.

**In-order:** : Used to get the values of the nodes in non-decreasing order in a BST.

**Post-order:** : Used to delete a tree from leaf to root

A **balanced binary tree** is commonly defined as a binary tree in which the depth of the left and right subtrees of every node differ by 1 or less,

The depth of a node M in a tree is the length of the path from the root of the tree to M. The height of a tree is one more than the depth of the deepest node in the tree. All nodes of depth d are at level d in the tree. The root is the only node at level 0, and its depth is 0.

**HEIGHT** is defined as the number of nodes in the longest path from the root node to a leaf node. Therefore: a tree with only a root node has a height of 1 and not 0.

The **LEVEL** of a given node is the distance from the root plus 1. Therefore: The root is on level 1, its child nodes are on level 2 and so on.

**Balanced Binary Tree**

The constraint is generally applied recursively to every subtree. That is, the tree is only balanced if:

1. The left and right subtrees' heights differ by at most one, AND
2. The left subtree is balanced, AND
3. The right subtree is balanced

According to this, the next tree is balanced:

A

/ \

B C

/ / \

D E F

/

G

The next one is **not** balanced because the subtrees of C differ by 2 in their height:

A

/ \

B C <-- difference = 2

/ /

D E

/

G

An AVL tree implements the Map abstract data type just like a regular binary search tree, the only difference is in how the tree performs. To implement our AVL tree we need to keep track of a **balance factor** for each node in the tree. We do this by looking at the heights of the left and right subtrees for each node. More formally, we define the balance factor for a node as the difference between the height of the left subtree and the height of the right subtree.

balanceFactor=height(leftSubTree)−height(rightSubTree)

Using the definition for balance factor given above we say that a subtree is left-heavy if the balance factor is greater than zero. If the balance factor is less than zero then the subtree is right heavy. If the balance factor is zero then the tree is perfectly in balance. For purposes of implementing an AVL tree, and gaining the benefit of having a balanced tree we will define a tree to be in balance if the balance factor is -1, 0, or 1. Once the balance factor of a node in a tree is outside this range we will need to have a procedure to bring the tree back into balance. [Figure 1](http://interactivepython.org/runestone/static/pythonds/Trees/BalancedBinarySearchTrees.html#fig-unbal) shows an example of an unbalanced, right-heavy tree and the balance factors of each node.

**Usage of Binary Search Tree**

A routing table is used to link routers in a network. It is usually implemented with a trie data structure, which is a variation of a binary tree. **The tree data structure will store the location of routers based on their IP addresses. Routers with similar addresses are grouped under a single subtree.**

To find a router to which a packet must be forwarded, we need to traverse the tree using the prefix of the network address to which a packet must be sent. Afterward, the packet is forwarded to the router with the [longest matching prefix](https://www.baeldung.com/spring-cloud-gateway) of the destination address.

**Complete and Correct Binary search tree code is given below.**

**import** java.util.LinkedList;

**import** java.util.Queue;

**public** **class** BinaryTree {

**private** Node root;

**public** **class** Node {

**private** Node left;

**private** Node right;

**private** **int** data;

**public** Node(**int** x) {

data = x;

left = **null**;

right = **null**;

}

}

**public** Node insert(**int** x) {

**return** root = insert(root, x);

}

// BST should not have duplicate

**public** Node insert(Node node, **int** x) {

**if** (node == **null**)

node = **new** Node(x);

**else** {

**if** (x < node.data)

node.left = insert(node.left, x);

**else** **if** (x > node.data)

node.right = insert(node.right, x);

**else**

node.data = x;

}

**return** node;

}

// \*\*\*\*\*\*\*\*\*\*\*\*\* Binary Tree Traversal \*\*\*\*\*\*\*\*\*\*\*\*\*\*

**public** **void** preOrderVisit() {

preOrderVisit(root);

}

**public** **void** preOrderVisit(Node node) {

**if** (node == **null**)

**return**;

System.***out***.print(node.data + " ");

preOrderVisit(node.left);

preOrderVisit(node.right);

}

**public** **void** postOrderVisit() {

postOrderVisit(root);

}

**public** **void** postOrderVisit(Node node) {

**if** (node == **null**)

**return**;

postOrderVisit(node.left);

postOrderVisit(node.right);

System.***out***.print(node.data + " ");

}

**public** **void** inOrderVisit() {

inOrderVisit(root);

}

**public** **void** inOrderVisit(Node node) {

**if** (node == **null**) **return**;

inOrderVisit(node.left);

System.***out***.print(node.data + " ");

inOrderVisit(node.right);

}

**public** **void** depthFirstSearch() {

Queue<Node> q = **new** LinkedList<Node>();

q.offer(root);

**while** (!q.isEmpty()) {

Node node = q.poll();

System.***out***.print("\t" + node.data);

**if** (node.left != **null**)

q.offer(node.left);

**if** (node.right != **null**)

q.offer(node.right);

}

}

**public** **int** size() { // Returns the number of nodes in the tree

**return** size(root);

}

**public** **int** size(Node node) {

**if** (node == **null**) **return** 0;

**else**

**return** (size(node.left) + 1 + size(node.right));

}

**public** **void** mirror() {

mirror(root);

}

**public** **void** mirror(Node node) {

**if** (node != **null**) {

mirror(node.left);

mirror(node.right);

Node temp = node.left;

node.left = node.right;

node.right = temp;

}

}

**public** Node delete(**int** x) {

**return** root = delete(root, x);

}

**public** Node delete(Node node, **int** x) {

**if** (node == **null**)

**throw** **new** RuntimeException("No data available");

**if** (x < node.data)

node.left = delete(node.left, x);

**else** **if** (x > node.data)

node.right = delete(node.right, x);

**else** {

**if** (node.left == **null**)

**return** node.right;

**else** **if** (node.right == **null**)

**return** node.left;

**else** {

node.data = node.left.data;

node.left = delete(node.left, node.data);

}

}

**return** node;

}

**public** **boolean** search(**int** x) {

**return** search(root, x);

}

**public** **boolean** search(Node node, **int** x) {

**if** (node == **null**)

**return** **false**;

**if** (x < node.data)

**return** search(node.left, x);

**else** **if** (x > node.data)

**return** search(node.right, x);

**else**

**return** **true**;

}

**public** **int** findMin() { // Another way of writing minimum

**int** val = 0;

**for** (Node p = root; p != **null**; p = p.left)

val = p.data;

**return** val;

}

**public** **int** findMax() { // Another way of writing maximum

**int** val = 0;

**for** (Node p = root; p != **null**; p = p.right)

val = p.data;

**return** val;

}

// Height of Binary search tree

**public** **int** getHeight() {

**return** getHeight(root);

}

**public** **int** getHeight(Node node) {

**if** (node == **null**)

**return** -1;

**else**

**return** 1 + Math.*max*(getHeight(node.left), getHeight(node.right));

}

// Find minimum value

**public** **int** findMinimum() {

**if** (root == **null**) **return** 0;

Node currNode = root;

**while** (currNode.left != **null**) {

currNode = currNode.left;

}

**return** currNode.data;

}

// Find Maximum

**public** **int** findMaximum() {

**if** (root == **null**) {

**return** 0;

}

Node currNode = root;

**while** (currNode.right != **null**) {

currNode = currNode.right;

}

**return** currNode.data;

}

**public** **int** minimumDepth() {

**return** minimumDepth(root);

}

**public** **int** minimumDepth(Node root) {

**if** (root == **null**)

**return** 0;

**if** (root.left == **null** && root.right == **null**)

**return** 1;

**if** (root.left == **null**)

**return** minimumDepth(root.right) + 1;

**if** (root.right == **null**)

**return** minimumDepth(root.left) + 1;

**return** 1 + Math.*min*(minimumDepth(root.left), minimumDepth(root.right));

}

**public** **int** minDepth2() {

**return** minDepth2(root);

}

**public** **int** minDepth2(Node node) {

**if** (node == **null**)

**return** 0;

**int** left = minDepth2(node.left);

**int** right = minDepth2(node.right);

**if** (0 == left && 0 == right)

**return** 1;

**if** (0 == left || 0 == right)

**return** Math.*max*(left, right) + 1;

**return** Math.*min*(left, right) + 1;

}

**public** **int** maxDepth1() {

**return** maxDepth(root);

}

**public** **int** maxDepth1(Node node) {

**if** (node == **null**)

**return** 0;

**return** 1 + Math.*max*(maxDepth1(node.left), maxDepth1(node.right));

}

**public** **int** maxDepth() {

**return** maxDepth(root);

}

**public** **int** maxDepth(Node node) {

**if** (node == **null**) **return** 0;

**else** {

**int** lDepth = maxDepth(node.left);

**int** rDepth = maxDepth(node.right);

**return** 1 + Math.*max*(lDepth, rDepth);

}

}

**public** **boolean** isBST() {

**return** isBST(root, Integer.***MIN\_VALUE***, Integer.***MAX\_VALUE***);

}

**public** **boolean** isBST(Node node, **int** min, **int** max) {

**if** (node == **null**)

**return** **true**;

**if** (node.data < min || node.data > max)

**return** **false**;

**return** isBST(node.left, min, node.data - 1) && isBST(node.right, node.data + 1, max);

}

**public** **boolean** isValidBST() { // Another way of writing

**return** isValidBST(root);

}

**int** lastValue = Integer.***MIN\_VALUE***;

**public** **boolean** isValidBST(Node root) {

**if** (root == **null**)

**return** **true**;

**if** (!isValidBST(root.left))

**return** **false**;

**if** (root.data < lastValue)

**return** **false**;

lastValue = root.data;

**return** isValidBST(root.right);

}

// Balanced Binary search

**public** **boolean** isBalanced() {

**return** isBalanced(root);

}

**public** **boolean** isBalanced(Node node) {

**return** (maxDepth(root) - minimumDepth(root) <= 1);

}

**public** **static** **void** main(String[] args) {

**int**[] a = **new** **int**[]{11,7,13,20,21,10};

BinaryTree bt = **new** BinaryTree();

**for**(**int** i : a)

bt.insert(i);

System.***out***.println("minimumDepth : " + bt.minimumDepth());

System.***out***.println("minDepth 2 : " + bt.minDepth2());

System.***out***.println("MaximumDepth : " + bt.maxDepth());

System.***out***.println("MaximumDepth 1 : " + bt.maxDepth1());

System.***out***.println("Get Height : "+bt.getHeight());

System.***out***.println("Is Balanced : "+ bt.isBalanced());

}

}

Breadth-First Traversal in BST

* Push level wise nodes into the queue at every iteration.
* Pop the node from queue at every iteration and print its value.
* And at every loop element from queue is removed in FIFO manner.

public void breadthFirstNonRecursive() {

    Queue<Node> queue = new LinkedList<>();

    queue.offer(root);

    while (!queue.isEmpty()) {

        Node node = queue.poll();

        System.out.println(node.data);

        if (node.left != null)

            queue.offer(node.left);

        if (node.right != null)

            queue.offer(node.right);

    }

}

**20**

**15              25**

**12      18      22      28**

--------------------------------

20 | 15 | 25 | 12 | 18 | 22 | 28

--------------------------------

And at every loop element from queue is removed in FIFO manner

**Output:**20, 15, 25, 12, 18, 22, 28

**Correct Implementation of MinDepth and MaxDepth**

**Min Depth**

public int minimumDepth() {  
 return minimumDepth(root);  
}  
  
public int minimumDepth(Node root) {  
 if (root == null) return 0;  
 if (root.left == null && root.right == null) return 1;  
  
 if (root.left == null)  
 return minimumDepth(root.right) + 1;  
  
 if (root.right == null)  
 return minimumDepth(root.left) + 1;  
  
 return 1 + Math.*min*(minimumDepth(root.left),  
 minimumDepth(root.right)) ;  
}

Another way of writing

public int minDepth2() {  
 return minDepth2(root);  
}  
  
public int minDepth2(Node node) {  
 if (node == null)  
 return 0;  
  
 int left = minDepth2(node.left);  
 int right = minDepth2(node.right);  
 if (0 == left && 0 == right) return 1;  
 if (0 == left || 0 == right)  
 return Math.*max*(left, right) + 1;  
  
 return Math.*min*(left, right) + 1;  
}

**Max Depth**

public int maxDepth1() {  
 return maxDepth(root);  
}  
  
public int maxDepth1(Node node) {  
 if (node == null) return 0;  
 return 1 + Math.*max*(maxDepth(node.left), maxDepth(node.right)) ;  
}

Another way of writing

public int maxDepth() {  
 return maxDepth(root);  
}  
  
public int maxDepth(Node node) {  
 if (node == null) return 0;  
 else {  
 int lDepth = maxDepth(node.left);  
 int rDepth = maxDepth(node.right);  
 return (Math.*max*(lDepth, rDepth) + 1);  
 }  
}

Correct implementation of deleting a node from Binary Search Tree

public void deleteNode(int x) {  
 root = deleteRec(root, x);  
 }  
  
 public Node deleteRec(Node node, int x) {  
 */\* Base Case: If the tree is empty \*/* if (node == null)  
 return node;  
  
 */\* Otherwise, recur down the tree \*/* if (x < node.data)  
 node.left = deleteRec(node.left, x);  
 else if (x > node.data)  
 node.right = deleteRec(node.right, x);  
  
 *// if key is same as root's key, then This is the node to be deleted* else {  
 *// node with only one child or no child* if (node.left == null)  
 return node.right;  
 else if (node.right == null)  
 return node.left;  
  
 *// node with two children: Get the inorder  
 // successor (smallest in the right subtree)  
// root.data = minValue(node.right); 🡺 do not use* **node.data = minVal(node.right);**  
 *// Delete the inorder successor* **node.right = deleteRec(node.right, node.data);**  
 }  
  
 return node;  
 }  
  
 public int minVal(Node node) {  
 int minv = node.data;  
 for (Node p = node; p != null; p = p.left)  
 minv = p.data;  
 return minv;  
 }

The older version code can be written like below.

public Node delete(Node node, int x) {  
 if (node == null) return node;  
 if (x < node.data)  
 node.left = delete(node.left, x);  
 else if (x > node.data)  
 node.right = delete(node.right, x);  
 else {  
 if (node.left == null)  
 return node.right;  
 else if (node.right == null)  
 return node.left;  
 else {  
 **node.data = minVal(node.right);** *//node.left.data;* **node.right = delete(node.right, node.data);**  
 }  
 }  
 return node;  
 }