Aryabhatta Knowledge University (AKU)

Information Technology (IT)

Data Structure and Algorithm

Solved Exam Paper 2019

Question 1. What is a hash table, and what is the average case and worst case time for each of its operations? How can we use this structure to find all anagrams in a dictionary?

Answer: The difficulty with direct addressing is obvious: if the universe U is large, storing a table T of size |U| may be impractical, or even impossible, given the memory available on a typical computer. Furthermore, the set K of keys actually stored may be so small relative to U that most of the space allocated for T would be wasted.

When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct-address table. Specifically, the storage requirements can be reduced to $\Theta(|K|)$, even though searching for an element in the hash table still requires only O(1) time. (The only catch is that this bound is for the average time, whereas for direct addressing it holds for the worst-case time.)

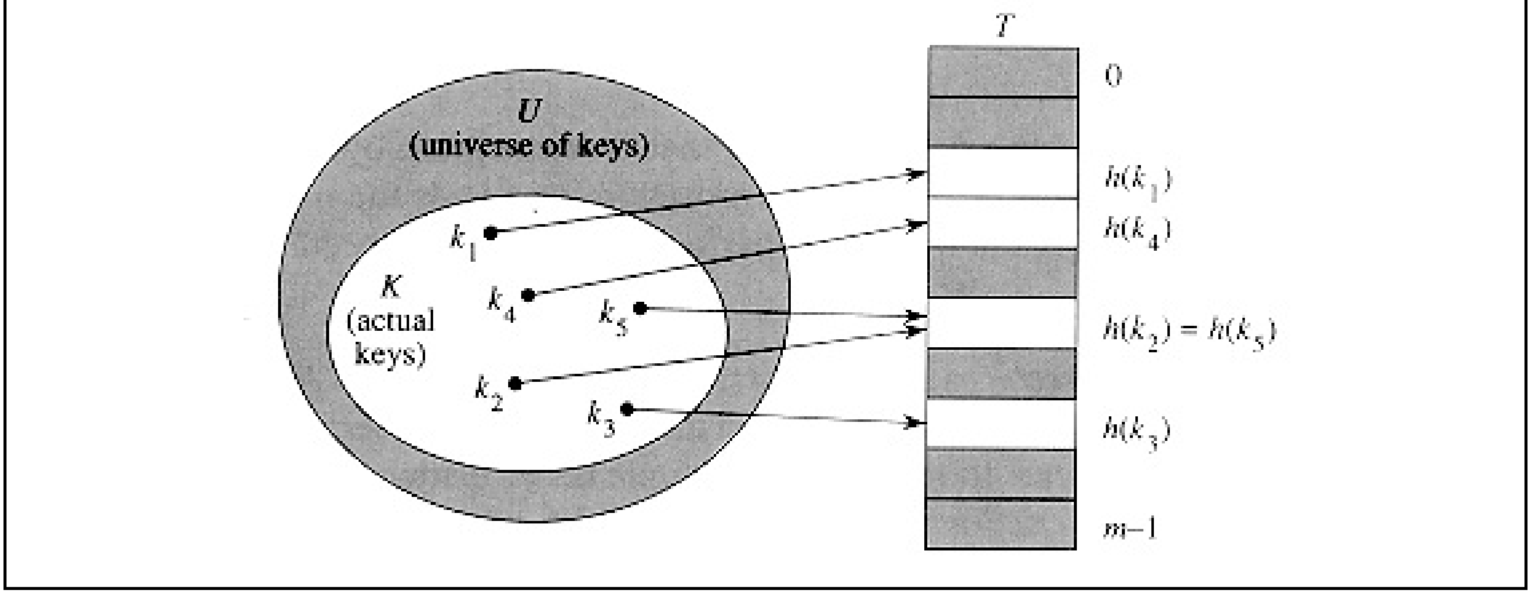


Figure: Using a hash function h to map keys to hash-table slots. Keys k_2 and k_5 map to the same slot, so they collide.

With direct addressing, an element with key k is stored in slot k. With hashing, this element is stored in slot h(k); that is, a **hash function** h is used to compute the slot from the key k. Here h maps the universe U of keys into the slots of a **hash table** T[0 . . m - 1]:

$h: U \rightarrow \{0,1, . . . , m - 1\}$.

We say that an element with key k **hashes** to slot h(k); we also say that h(k) is the **hash value** of key k. Figure illustrates the basic idea. The point of the hash function is to reduce the range of array indices that need to be handled. Instead of |U| values, we need to handle only m values. Storage requirements are correspondingly reduced.

The fly in the ointment of this beautiful idea is that two keys may hash to the same slot--a **collision.** Fortunately, there are effective techniques for resolving the conflict created by collisions.

Of course, the ideal solution would be to avoid collisions altogether. We might try to achieve this goal by choosing a suitable hash function h. One idea is to make h appear to be "random," thus avoiding collisions or at least minimizing their number. The very term "to hash," evoking images of random mixing and chopping, captures the spirit of this approach. (Of course, a hash function h must be deterministic in that a given input k should always produce the same output h(k).) Since |U| > m, however, there must be two keys that have the same hash value; avoiding collisions altogether is therefore impossible. Thus, while a well-designed, "random"- looking hash function can minimize the number of collisions, we still need a method for resolving the collisions that do occur.

The remainder of this section presents the simplest collision resolution technique, called chaining. Section introduces an alternative method for resolving collisions, called open addressing.

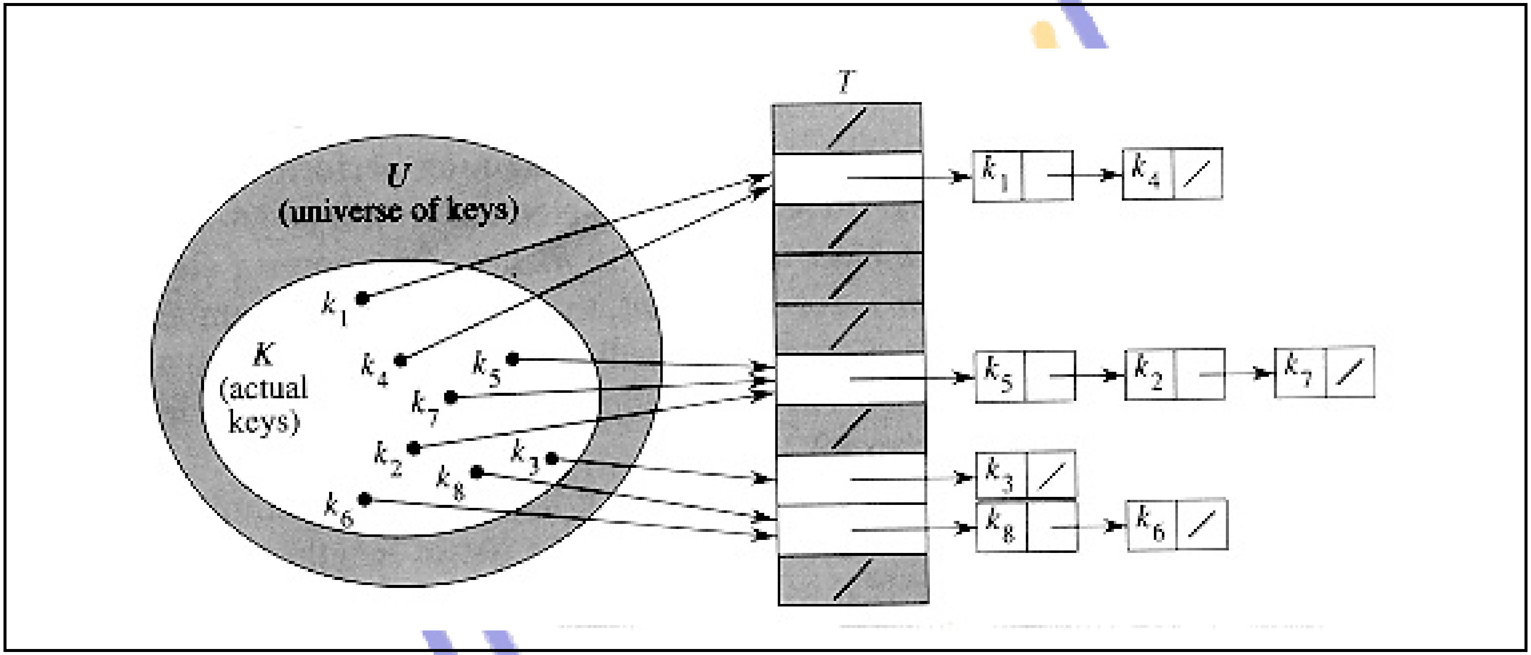


Figure: Collision resolution by chaining. Each hash-table slot T[j] contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.

Since, we will be performing multiple anagram queries, our first step is to load all of the (25,000) words in the dictionary into an appropriate data structure. A primary requirement is that one must be able to efficiently search this data structure to look for anagrams of a given word. A clever trick that we will use to facilitate this is to first sort the letters of every word we insert into our data structure (you may use any sort you wish to produce a key for each word. For example, the key for the string "toxic" is "ciotx", similarly the key for both "star" and "rats" is "arst". We will then use a hash table to store pairs of strings, where the pair consists of the original word and its key. When performing insertions into the hash table, we will compute the hash of the key of the word to compute the correct bucket (location in the hash table). This approach guarantees that all words which are anagrams of one another are stored in the same bucket of the hash table. Similarly, when we are searching for anagrams, we will first compute the key of the word we are searching for, then hash the key, then search that bucket for anagram matches. You should feel free to use any appropriate hash function for hashing strings (but please cite any source which you use). Also, make sure your function is efficient and does not hash completely unrelated sets of anagrams to the same bucket if possible. If it does, handle the collisions as you see fit (e.g. linked processing). Also note that if you must probe for a given set of anagrams in time greater than or equal to O(logn), then you must revise your hash function. You will be graded heavily on the performance of the efficiency of your function.

The hash table code which you provide only needs to have the minimum functionality needed to solve this problem. You may fix a size for your hash table for efficient searching. It is recommended that the final hash table you submit contain at least 25,000 buckets. (For debugging your code, it is suggested that you work with a much smaller practice dictionary, perhaps 10 words, and a much smaller hash table, perhaps 8-10 buckets (depending on whether or not there are any anagrams in the dictionary). Make sure your table size is prime to help reduce collisions. Remember it is ok to sacrifice space for speed -- that is what hashing is all about. That said, your table should not be bigger than 200,000. 1You may disregard any words in the dictionary which contain any punctuation characters. Also, you should convert any uppercase characters to lowercase (thus you are only representing words that contain all lower case characters). Your program should read anagram queries from an input file (“input.txt”).

Each query in the file will be on its own line and will simply consist of a string. The output file (“output.txt”) should contain the original string, then the number of matching anagrams, followed by those anagrams. An example input file and the resulting output file have been provided. Your output file should match this format exactly, except that the matching anagrams you output may be ordered differently.

Question 2. Describe insertion in max heap tree with example from the following list of number: 35 33 42 10 14 19 27 44 26 31

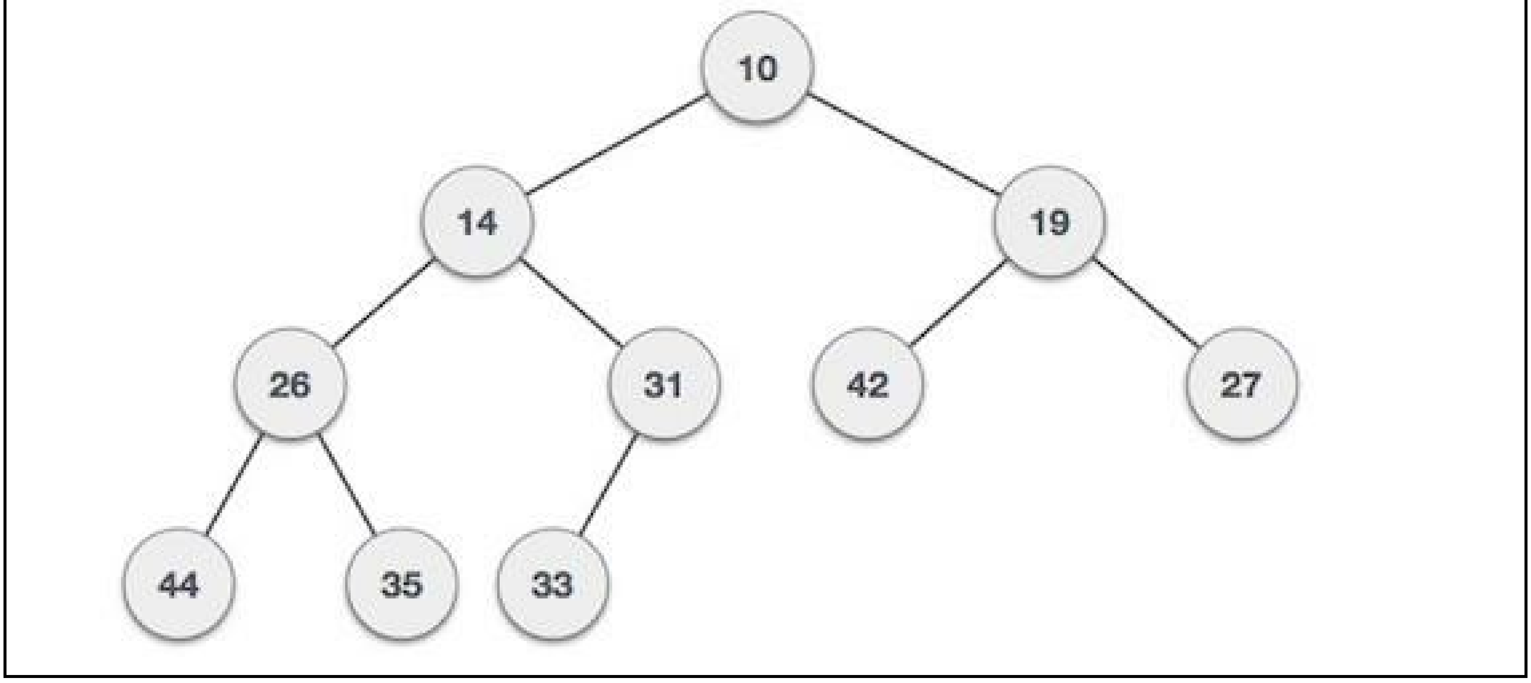
Answer: Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If α has child node β then −

key(α) ≥ key(β)

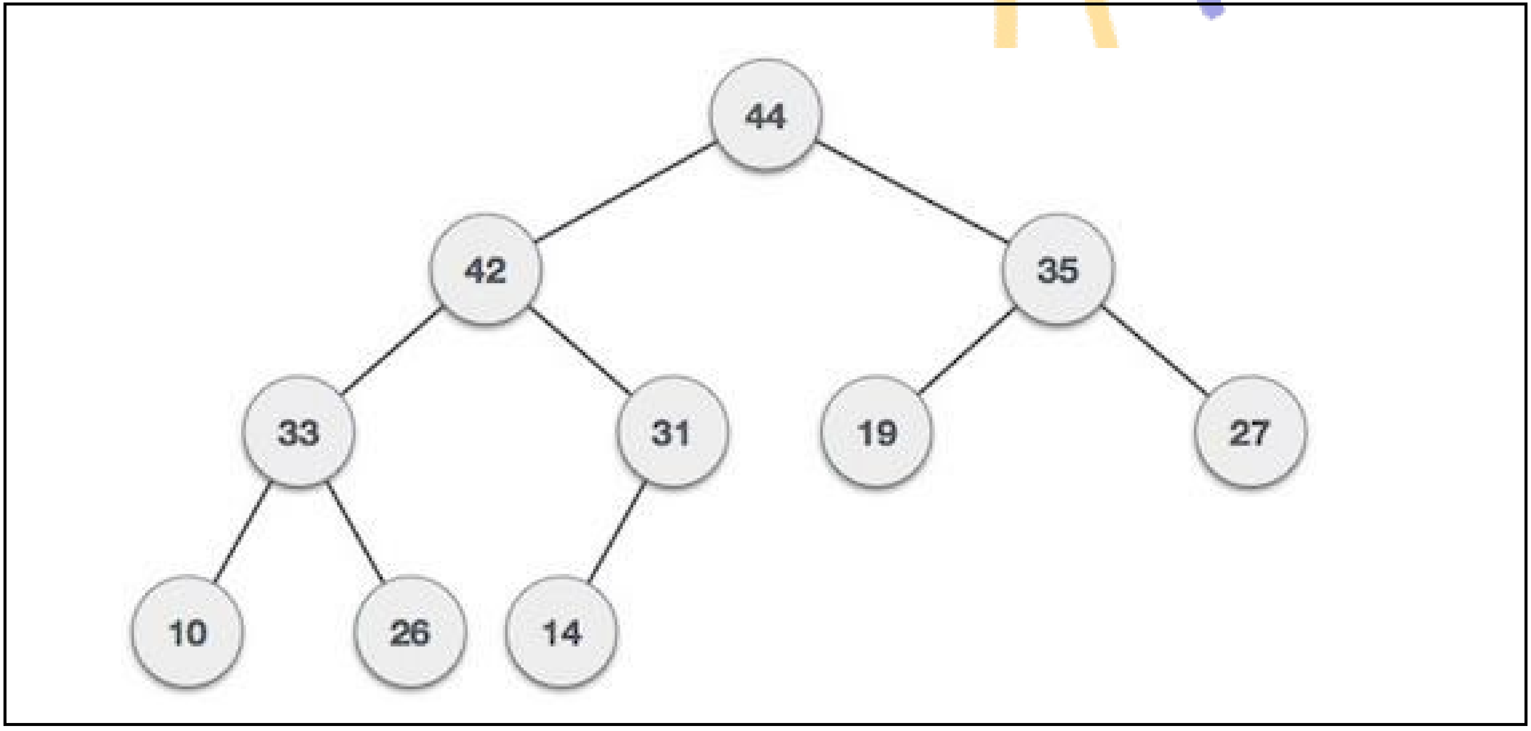
As the value of parent is greater than that of child, this property generates Max Heap. Based on this criteria, a heap can be of two types −

For Input → 35 33 42 10 14 19 27 44 26 31

Min-Heap − Where the value of the root node is less than or equal to either of its children.



Max-Heap − Where the value of the root node is greater than or equal to either of its children.



Both trees are constructed using the same input and order of arrival.

Max Heap Construction Algorithm

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

Step 1 − Create a new node at the end of heap.

Step 2 − Assign new value to the node.

Step 3 − Compare the value of this child node with its parent.

Step 4 − If value of parent is less than child, then swap them.

Step 5 − Repeat step 3 & 4 until Heap property holds.

Note − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Question. Sort the given values using quicksort and write time complexity of algorithm: 65, 70, and 75,80,85,60,55,50,45

Answer: Sorting takes place from the pivot value, which is the first value of the given elements, this is marked bold. The values at the left pointer and right pointer are indicated using L and R respectively.

65 70L 75 80 85 60 55 50 45R

Since pivot is not yet changed the same process is continued after interchanging the values at L and R positions

65 45 75 L 80 85 60 55 50 R 70

65 45 50 80 L 85 60 55 R 75 70

65 45 50 55 85 L 60 R 80 75 70

65 45 50 55 60 R 85 L 80 75 70

When the L and R pointers cross each other the pivot value is interchanged with the value at right pointer. If the pivot is changed it means that the pivot has occupied its original position in the sorted order (shown in bold italics) and hence two different arrays are formed, one from start of the original array to the pivot position-1 and the other from pivot position+1 to end.

60 L 45 50 55 R 65 85 L 80 75 70 R

55 L 45 50 R 60 65 70 R 80 L 75 85

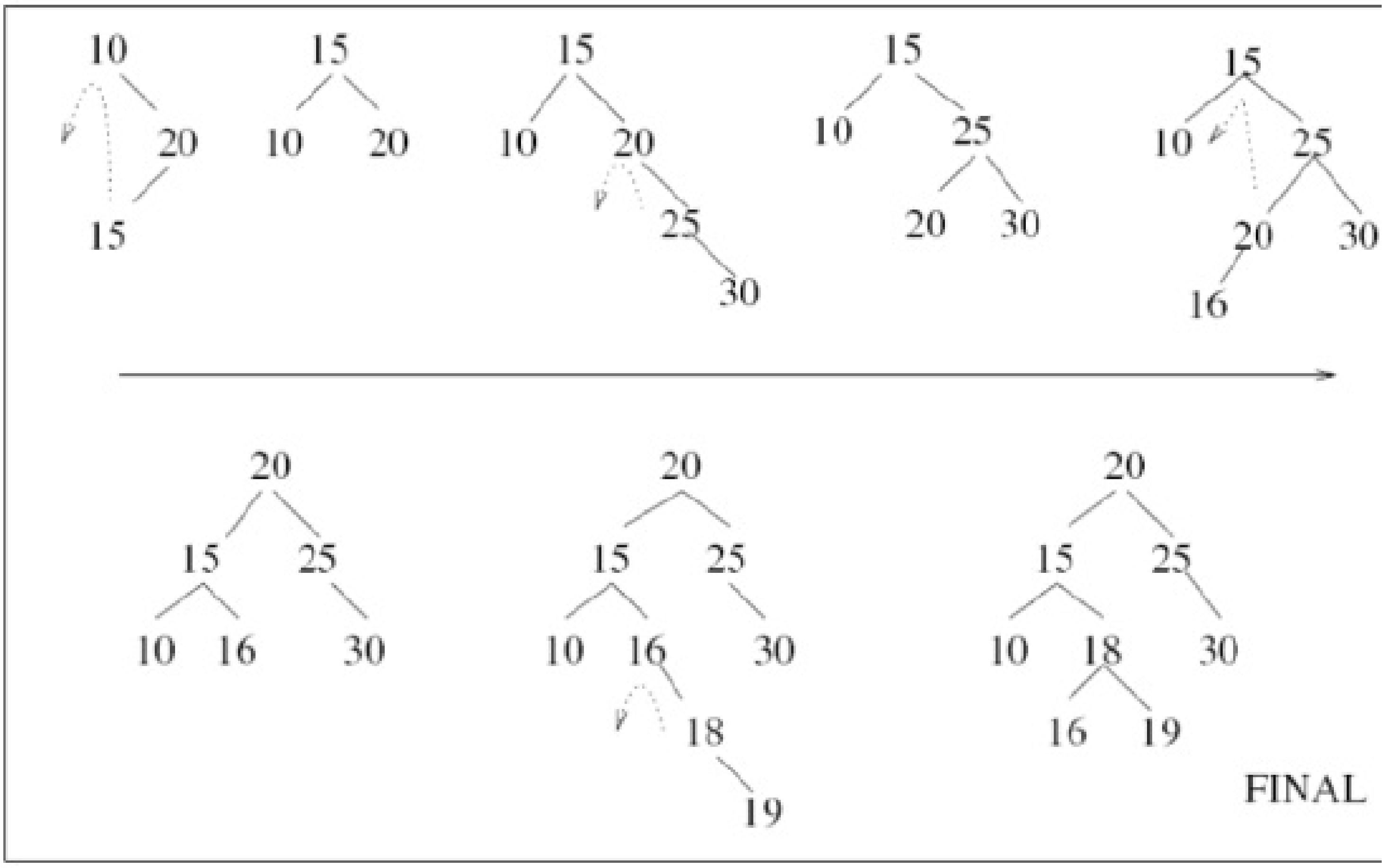
50 L 45 R 55 60 65 70 80 L 75 R 85

In the next pass we get the sorted form of the array.

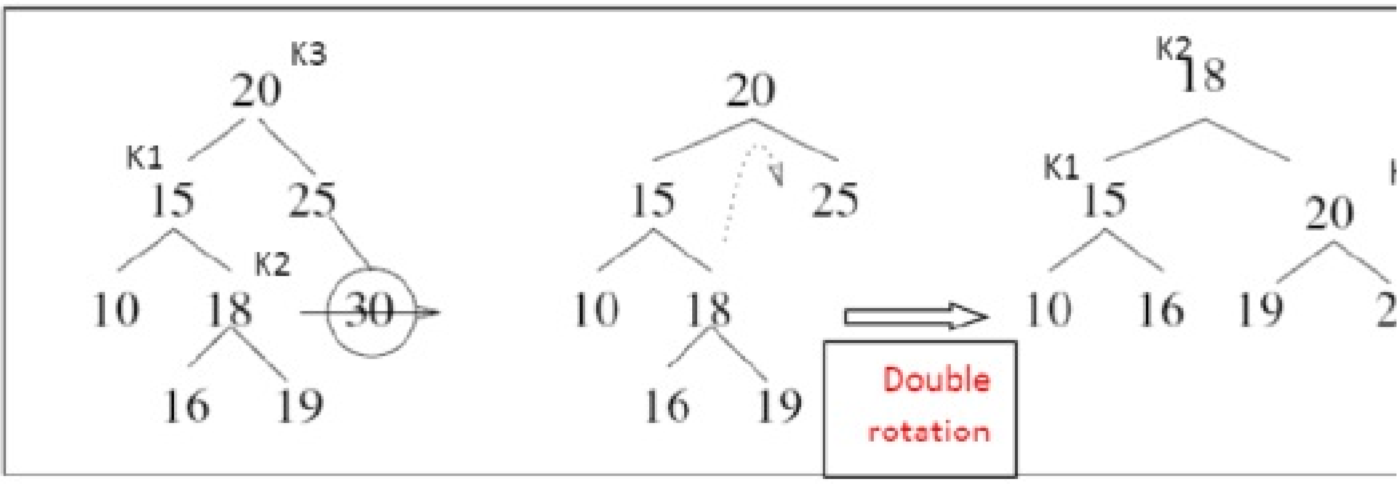
45 50 55 60 65 70 75 80 85

Question. Insert the following sequence of elements into an AVL tree, starting with the empty tree: 10, 20,15,25,30,16,18,19

Answer:



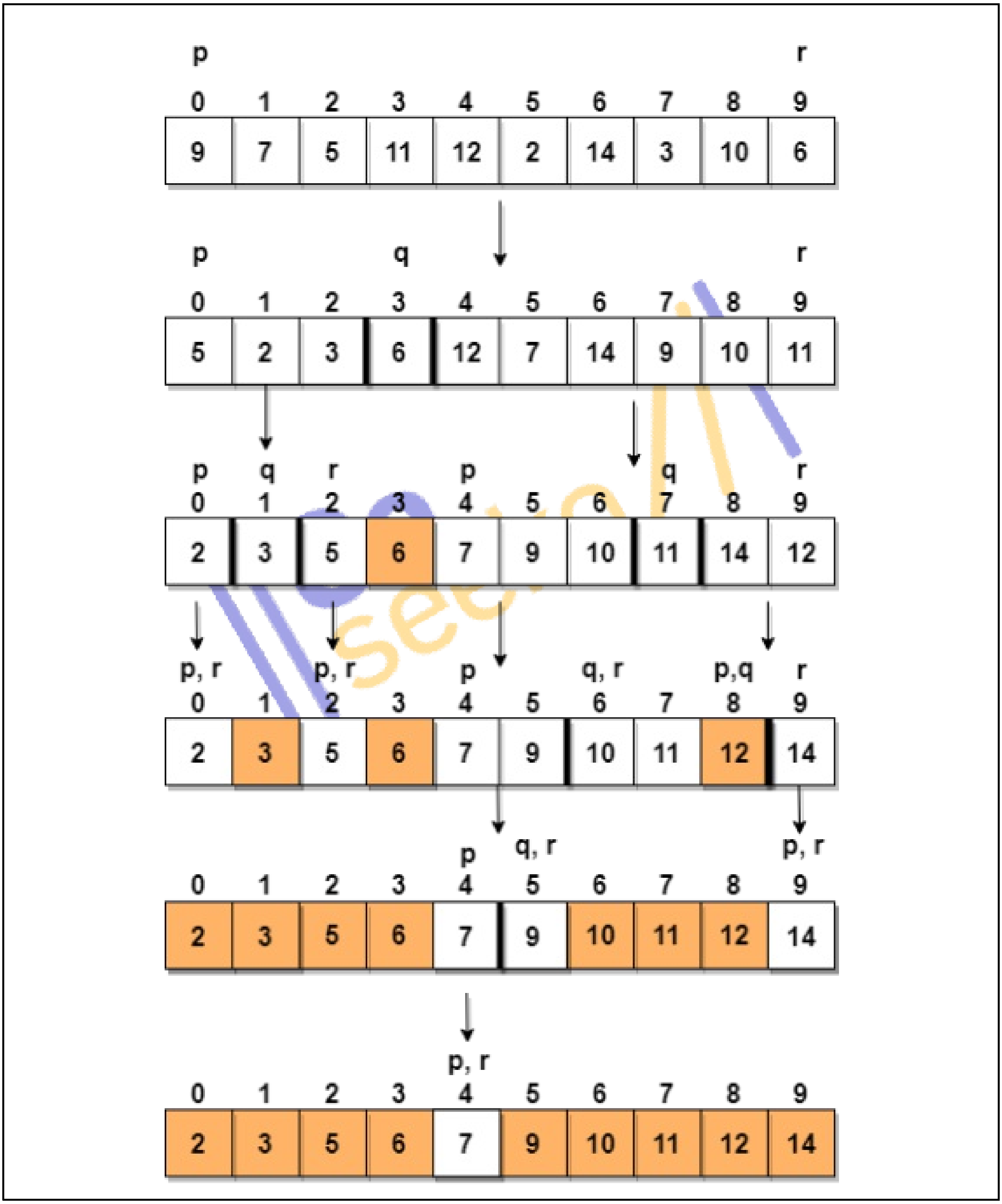
Delete 30 in the AVL tree that you got.



Question. Write algorithm for quicksort and mention time and space complexity in each case

Answer: Let's consider an array with values {9, 7, 5, 11, 12, 2, 14, 3, 10, 6}

Below, we have a pictorial representation of how quick sort will sort the given array.



In step 1, we select the last element as the pivot, which is 6 in this case, and call for partitioning, hence re-arranging the array in such a way that 6 will be placed in its final position and to its left will be all the elements less than it and to its right, we will have all the elements greater than it.

Then we pick the subarray on the left and the subarray on the right and select a pivot for them, in the above diagram, we chose 3 as pivot for the left subarray and 11 as pivot for the right subarray.

And we again call for partitioning.

For an array, in which partitioning leads to unbalanced subarrays, to an extent where on the left side there are no elements, with all the elements greater than the pivot, hence on the right side.

And if keep on getting unbalanced subarrays, then the running time is the worst case, which is O(n2)

Whereas if partitioning leads to almost equal subarrays, then the running time is the best, with time complexity as O(n*log n).

Worst Case Time Complexity [Big-O]: O(n2)

Best Case Time Complexity [Big-omega]: O(n*log n)

Average Time Complexity [Big-theta]: O(n*log n)

Space Complexity: O(n*log n)

As we know now, that if subarrays partitioning produced after partitioning are unbalanced, quick sort will take more time to finish. If someone knows that you pick the last index as pivot all the time, they can intentionally provide you with array which will result in worst-case running time for quick sort.

To avoid this, you can pick random pivot element too. It won't make any difference in the algorithm, as all you need to do is, pick a random element from the array, swap it with element at the last index, make it the pivot and carry on with quick sort.

Space required by quick sort is very less, only O(n*log n) additional space is required.

Quick sort is not a stable sorting technique, so it might change the occurence of two similar elements in the list while sorting.

Question. Define collision in hashing. What are the different methodologies to resolve collision? Explain briefly

Answer: Hash functions are there to map different keys to unique locations (index in the hash table), and any hash function which is able to do so is known as the perfect hash function. Since the size of the hash table is very less comparatively to the range of keys, the **perfect hash function** is practically impossible. What happens is, more than one keys map to the same location and this is known as a **collision**. A good hash function should have less number of collisions.

Collision resolution is finding another location to avoid the collision. The most popular resolution techniques are,

i. Separate chaining
ii. Open addressing

Open addressing can be further divided into,

i. Linear Probing
ii. Quadratic Probing
iii. Double hashing

Open Addressing: In this technique a hash table with per-identified size is considered. All items are stored in the hash table itself. In addition to the data, each hash bucket also maintains the three states: EMPTY, OCCUPIED, DELETED. While inserting, if a collision occurs, alternative cells are tried until an empty bucket is found. For which one of the following technique is adopted.

- 1.Liner Probing(this is prone to clustering of data + Some other constrains)
- 2.Quadratic probing
- 3.Double hashing(in short in case of collision another hashing function is used with the key value as an input to identify where in the open addressing scheme the data should actually be stored.)

Chaining: Open Hashing, is a technique in which the data is not directly stored at the hash key index (k) of the Hash table. Rather the data at the key index (k) in the hash table is a pointer to the head of the data structure where the data is actually stored. In the most simple and common implementations the data structure adopted for storing the element is a linked-list.

Linear probing
In this, when the collision occurs, we perform a linear probe for the next slot, and this probing is performed until an empty slot is found. In linear probing, the worst time to search for an element is O (table size). The linear probing gives the best performance of the cache but its problem is clustering. The main advantage of this technique is that it can be easily calculated.
Quadratic probing
In this, when the collision occurs, we probe for **i2th** slot in **ith** iteration, and this probing is performed until an empty slot is found. The cache performance in quadratic probing is lower than the linear probing. Quadratic probing also reduces the problem of clustering.

Double hashing
In this, you use another hash function, and probe for (i * hash 2(x)) in the ith iteration. It takes longer to determine two hash functions. The double probing gives the very poor the cache performance, but there has no clustering problem in it.

Question. Write an algorithm to count leaf node in a binary tree and check whether the tree is balanced or not.

Answer: The algorithm to count the total number of leaf nodes is very similar to the earlier problem about printing a leaf node.

Here are the actual steps to follow:

1) If the node is null return 0, this is also the base case of our recursive algorithm

2) If a leaf node is encountered then return 1

3) Repeat the process with left and right subtree

4) Return the sum of leaf nodes from both left and right subtree

The exact steps of the iterative algorithm to get a total number of leaf nodes of a binary tree:

1) If the root is null then return zero.

2) Start the count with zero

3) Push the root into Stack

4) loop until Stack is not empty

5) Pop the last node and push left and right children of the last node if they are not null.

6) Increase the count

A tree where no leaf is much farther away from the root than any other leaf. Different balancing schemes allow different definitions of “much farther” and different amounts of work to keep them balanced.

Consider a height-balancing scheme where following conditions should be checked to determine if a binary tree is balanced.

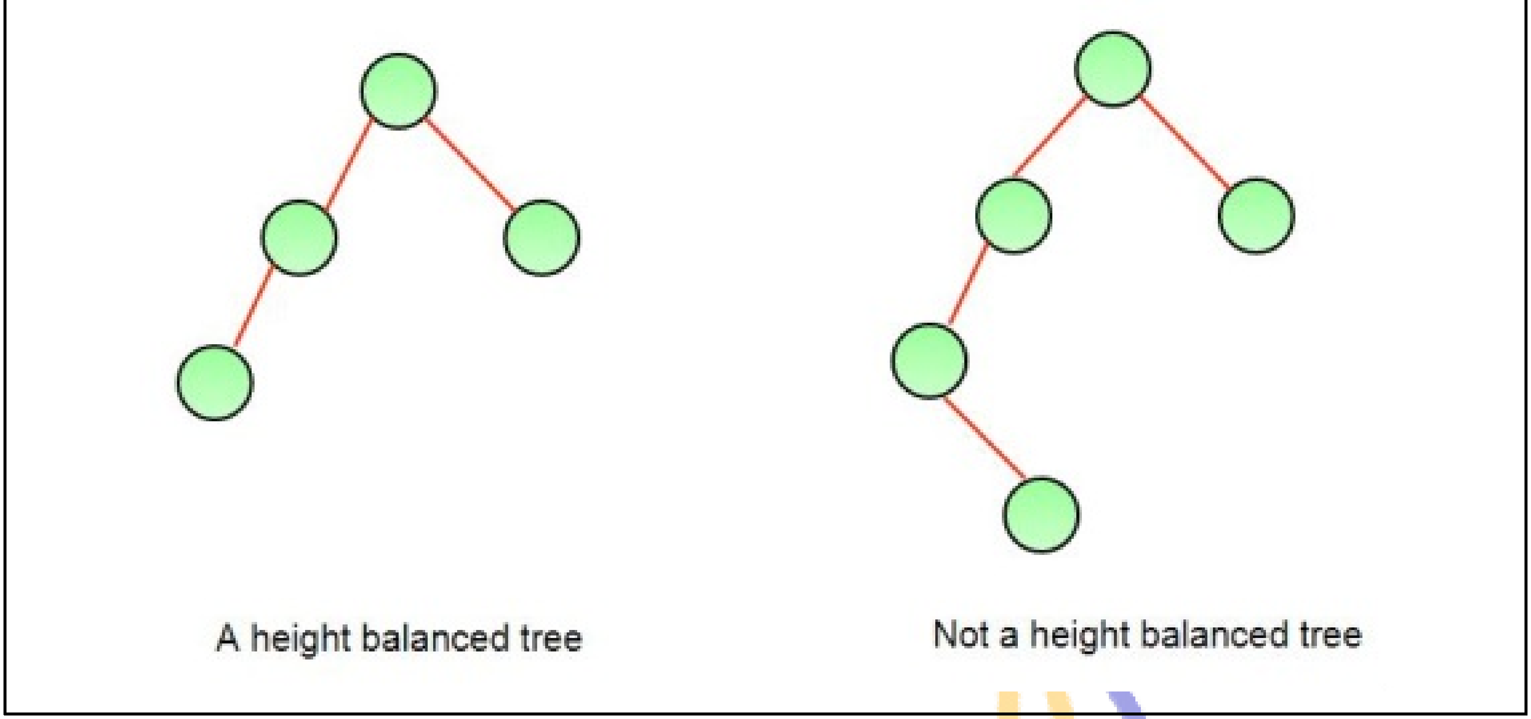
An empty tree is height-balanced. A non-empty binary tree T is balanced if:

1) Left subtree of T is balanced

2) Right subtree of T is balanced

3) The difference between heights of left subtree and right subtree is not more than 1.

The above height-balancing scheme is used in AVL trees. The diagram below shows two trees, one of them is height-balanced and other is not. The second tree is not height-balanced because height of left subtree is 2 more than height of right subtree.



Question. Write a recursive and iterative version of insertion sort algorithm and mention time complexity

Answer: Recursive Insertion Sort has no performance/implementation advantages, but can be a good question to check one's understanding of Insertion Sort and recursion.

If we take a closer look at Insertion Sort algorithm, we keep processed elements sorted and insert new elements one by one in the inserted array.

Algorithm

1. Base Case: If array size is 1 or smaller, return.
2. Recursively sort first n-1 elements.
3. Insert last element at its correct position in sorted array.

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm

To sort an array of size n in ascending order:

1: Iterate from arr [1] to arr [n] over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Below are the detailed example to illustrate the difference between the two:

1. **Time Complexity:** Finding the Time complexity of Recursion is more difficult than that of Iteration.

   - **Recursion**: Time complexity of recursion can be found by finding the value of the nth recursive call in terms of the previous calls. Thus, finding the destination case in terms of the base case, and solving in terms of the base case gives us an idea of the time complexity of recursive equations. Please see Solving Recurrences for more details.

   - **Iteration**: Time complexity of iteration can be found by finding the number of cycles being repeated inside the loop.

2. **Usage:** Usage of either of these techniques is a trade-off between time complexity and size of code. If time complexity is the point of focus, and number of recursive calls would be large, it is better to use iteration. However, if time complexity is not an issue and shortness of code is, recursion would be the way to go.

   - **Recursion**: Recursion involves calling the same function again, and hence, has a very small length of code. However, as we saw in the analysis, the time complexity of recursion can get to be exponential when there are a considerable number of recursive calls. Hence, usage of recursion is advantageous in shorter code, but higher time complexity.

- **Iteration**: Iteration is repetition of a block of code. This involves a larger size of code, but the time complexity is generally lesser than it is for recursion.

3. **Overhead:** Recursion has a large amount of Overhead as compared to Iteration.

- **Recursion**: Recursion has the overhead of repeated function calls, that is due to repetitive calling of the same function, the time complexity of the code increases manifold.
- **Iteration**: Iteration does not involve any such overhead.

4. **Infinite Repetition:** Infinite Repetition in recursion can lead to CPU crash but in iteration, it will stop when memory is exhausted.

- **Recursion**: In Recursion, Infinite recursive calls may occur due to some mistake in specifying the base condition, which on never becoming false, keeps calling the function, which may lead to system CPU crash.
- **Iteration**: Infinite iteration due to mistake in iterator assignment or increment, or in the terminating condition, will lead to infinite loops, which may or may not lead to system errors, but will surely stop program execution any further.

Question. Write short notes on:

a) BFS

b) DFS

c) Binary search tree

d) Balance factor

S.NO	BFS	DFS

1.	BFS stands for Breadth First Search.	DFS stands for Depth F Search.
2.	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search uses Stack data structu
3.	BFS can be used to find single source shortest path in an unweighted graph, because in BFS, we reach a vertex with minimum number of edges from a source vertex.	In DFS, we might trave through more edges to a destination vertex fro source.
4.	BFS is more suitable for searching vertices which are closer to the given source.	DFS is more suitable w there are solutions awa from source.
5.	BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles.	DFS is more suitable fo game or puzzle problem We make a decision, the explore all paths throug this decision. And if this decision leads to win situation, we stop.
6.	The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when	The Time complexity o is also O(V + E) Adjacency List is used O(V^2) when Adja

	Adjacency Matrix is used, where V stands for vertices and E stands for edges.	Matrix is used, whe stands for vertices a stands for edges.

Binary Search Tree

A Binary Search Tree (BST) is a tree in which all the nodes follow the below mentioned properties −

- The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
- The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

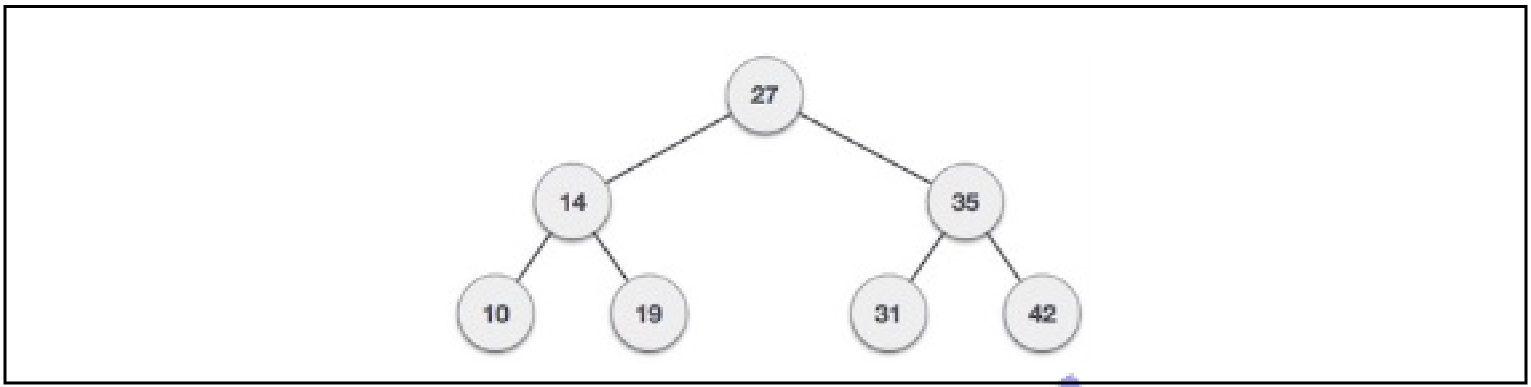
left_subtree (keys) < node (key) ≤ right_subtree (keys)

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser 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

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST −



Observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations

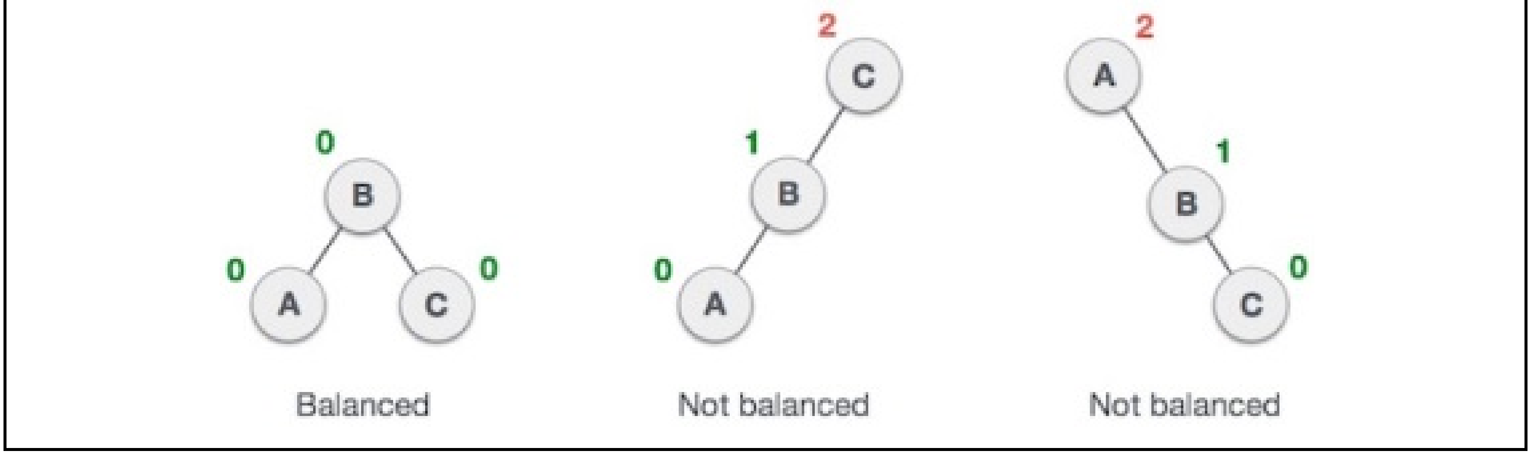
Following are the basic operations of a tree −

- Search − Searches an element in a tree.
- Insert − Inserts an element in a tree.
- Pre-order Traversal − Traverses a tree in a pre-order manner.
- In-order Traversal − Traverses a tree in an in-order manner.
- Post-order Traversal − Traverses a tree in a post-order manner.

Balance Factor

AVL trees are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the Balance Factor.

Here we see that the first tree is balanced and the next two trees are not balanced −



Unbalanced AVL Trees

In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of A has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

BalanceFactor = height (left-sutree) − height(right-sutree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

To balance itself, an AVL tree may perform the following four kinds of rotations −

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.