Data Structures

(revision material)

In every computational task we try to perform in our day to day life is nothing but doing some operation on data. For example, a simple mouse clicks on computer or tapping on our mobile touch screen invokes some operations that are associated with that region of screen/monitor like selecting text over there or changing the brightness of the screen or anything for that matter involves some operations on data relevant to it. All these aspects of computations finally boil down to accessing or manipulating data. How fast and how easily we can perform these data manipulations is crucial. Here we try to discuss some data structures that can develops some intuition of this concept.

HASH TABLE

In C language we have arrays which can arrange similar type of data elements in consecutive memory locations. This feature allows us to access any element of the array, by its index, just in constant time, O(1). At the same time C arrays are not elastic in size, one can't change their size once it is declared. On the other hand, C allows us to implement linked lists using structures and pointers using which one can arrange collection of data elements dynamically expandable in size.

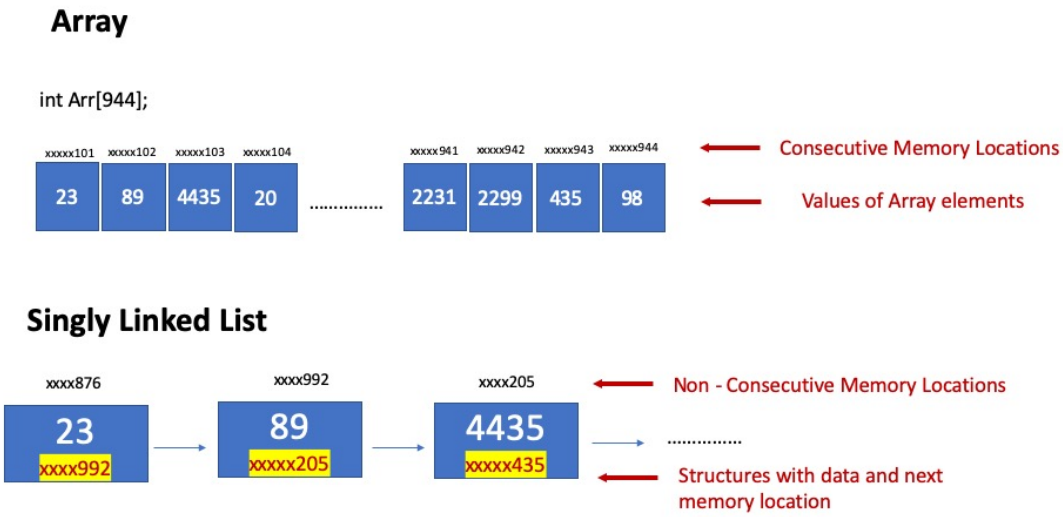


Figure 1: Array & Linked list

Hash tables are combined representation of both Arrays and linked lists. When we have data of indefinite size but can be categorized into definite number of groups basing on some function Hash table representation is very useful. The function that categorizes the data into non-overlapping is called hash function. Groups are represented using array as its size is fixed. Groups of data elements are stored in linked lists whose first address is stored in the array element that representing the group.

English (any language for that matter) dictionary can be represented using hash table as shown in *Figure 2*. All the words in the dictionary in grouped basing on their starting letter so number of possible groups is fixed now so it can be implemented using an array where each array element holds the starting address of corresponding linked list. In the Figure 2 one can observe array (orange like color) of singly linked lists(blue). Each array element stores the address of starting point of the corresponding linked list. This representation of dictionary data helps us in many operations such as

adding a new word to our dictionary. First, we try to find out what is the first letter of that word and corresponding linked lists starting address. In the next step the new word will be added in its appropriate position of the linked list. The complexity of accessing the starting address of relevant linked list is ~ O(1). And the complexity of adding the word to its linked list is always very less than O(n) where n is the total number of words in the dictionary. The complexity is same for other operations like editing a word, deleting a word or searching word.

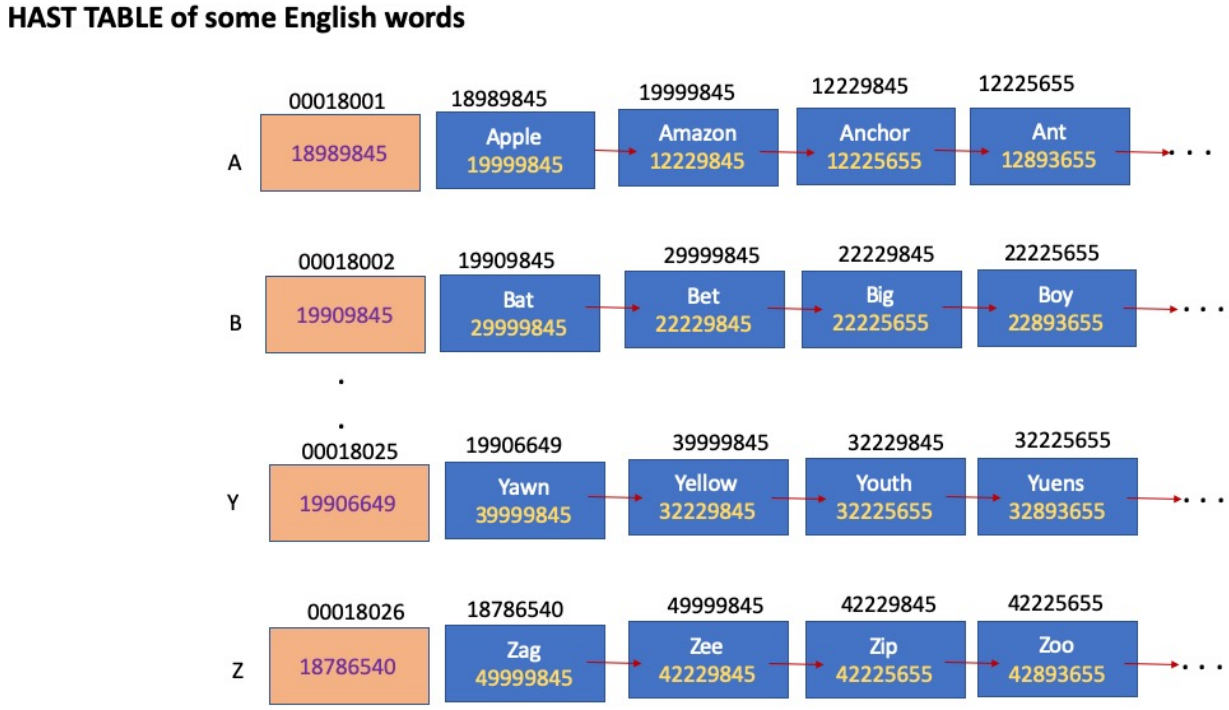


Figure 2 Hash table representation of collection of some English words

Some important aspects of Hash tables:

Hashing function that categorizes the data into non-overlapping groups. In the above case hashing function just finds the first character of the given word and maps it to the array index of corresponding linked list. Data collection need not always be collection of single items like our dictionary example. Data items can be anything, student's information, employee information, customer information, company's information like in stock markets, properties of molecules, atom co-ordinates in protein structures etc. One has to choose proper key that represents the item and its category. If students' records are our data of interest, each student record may consist lot of information such as his data of birth, institute joining date, his father's name etc. out all that student year of joining can be our key of categorization. If students with same year of joining are chained form a linked list their starting address should be kept in an array whose index can be obtained by hashing function for the given student record.

Activity:

1. What will be complexity if dictionary, like above, is implemented using just an array or just a linked list?
2. Think of some other collections of data that can be represented as hash table and how?
3. How a hash function can be chosen for a given set of data?
4. Write an algorithm/ pseudo code to implement hash table for given set of data in C.

Tree Structures

As we have seen in hash tables, categorizing data for organizing in memory help us in efficiently access it and manipulate it. One of the main goals of data structuring is to minimize the complexity of accessing and manipulating the data as per our requirement. The requirements may vary from problem to problem. In some cases, we may wish to access only the highest value in one particular dimension in others it may be the smallest. So, data structuring also varies from one problem to other.

But what is the generalized form of data organization?

Nature has given us the solution in the form of trees, circulatory systems animals, where very complex activities are being carried out. In a very minimalist form, we try to use these tree structures for our purpose of data organization and see how it helps us.

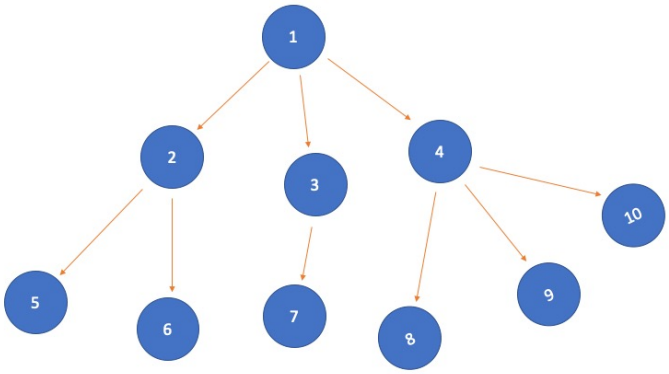


Figure 3: Tree structure

A generalized tree structure consists of nodes and branches as shown in figure 3. Node 1 is called the root node, nodes 5,6,7,8,9,10 are leaf nodes. Nodes 5&6 are child nodes of node 2 which is having root node as its parent. This nomenclature is often used in our discussions further. Nodes hold some data and represented by a key. In our example the numbers 1 to 10 are keys corresponding to each node. Usually keys are unique and collection of keys represents a set. If students' records are saved in tree structure each node consists of data corresponding to each student and its key can be anything that uniquely represents the student such as his roll number or rank in some entrance exam.

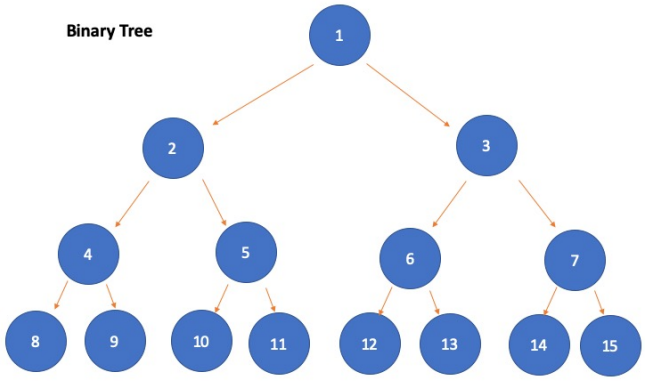


*Figure 4: In a **binary tree** any node can have at most two child nodes. It is also a complete binary tree as every node of it has two child nodes except leaf node.*

Heap/ Min-max tree/ priority queue:

A complete binary tree where key value of every node is greater than both of its child nodes'. There is no condition imposed on left and right child nodes. This kind of representation guarantees that root node will be always greater than all the nodes of the tree there by one can access the maximum value in constant time, O(1).

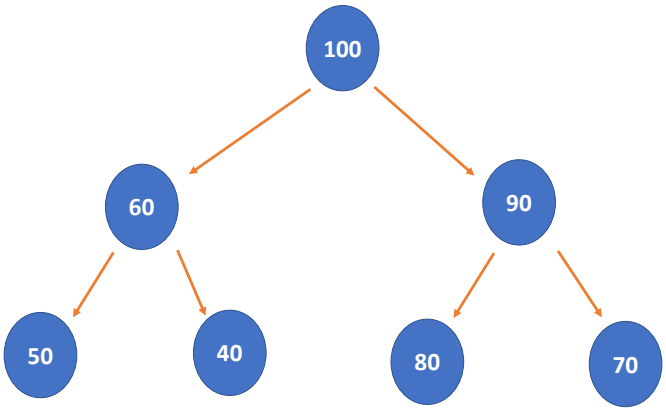


Figure 5 Heap

Adding a node to heap:

1. Heap always tries to be complete binary tree. Each level is filled from left to right.
2. Once it is added it checks the condition of heap (if the parent node of it is greater or not)
3. If the heap condition is not satisfied new node will be swapped with its parent node. This process of swapping continued till the heap condition is satisfied.

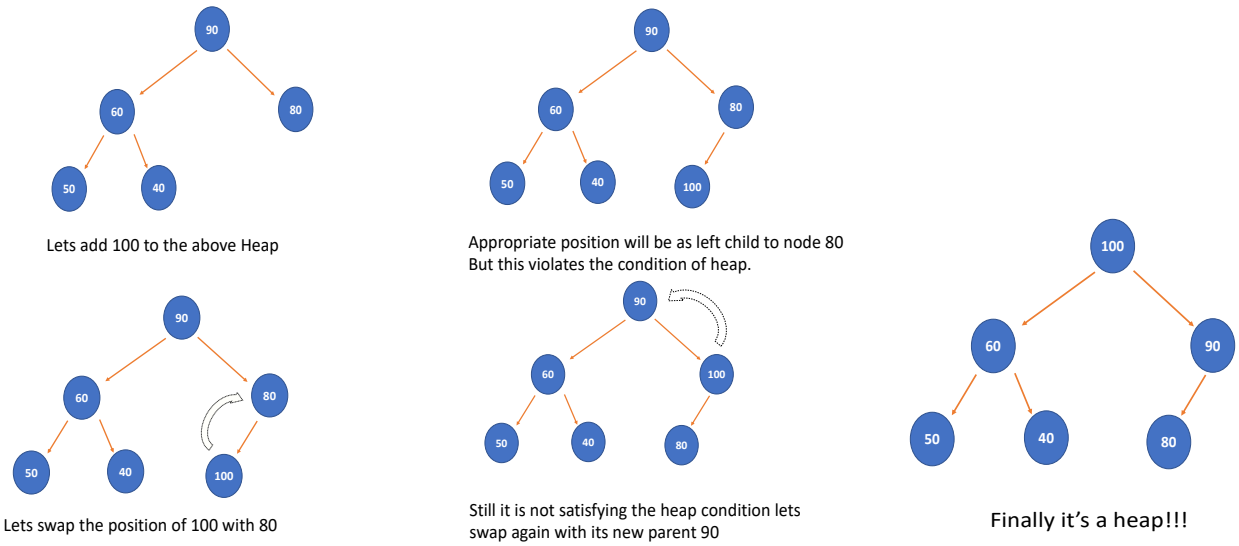
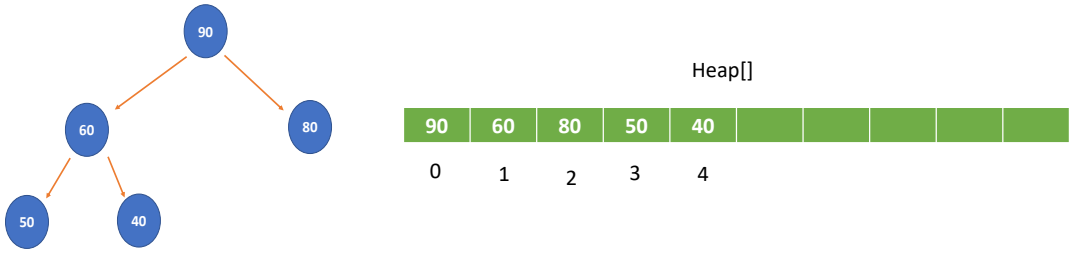


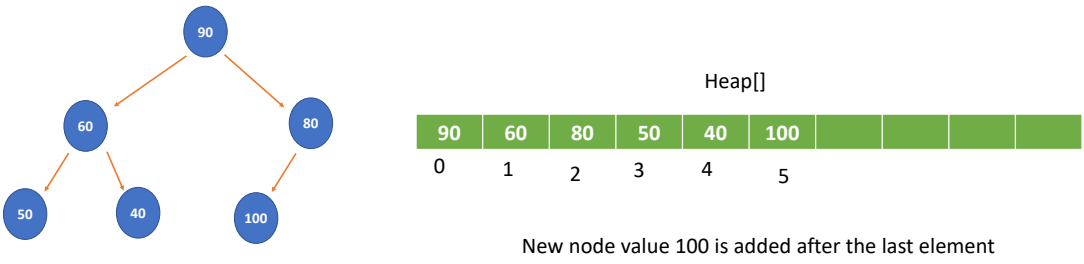
Figure 6: Adding a new node to heap

C implementation of Heap:

The best data structure we can use to implement heap in C is array.



Heap elements are represented in array. Root node as 0th element of array and other node values are filled level wise top to bottom and elements of the same level are filled from left to right.



New node value 100 is added after the last element

In order to perform swapping of nodes for satisfying heap condition we need to know with which element it is required to swap. Following are some general relations.

Kth nodes –

Left child will be at 2k+1

Right child at 2k+2

Parent is at (k-1)/2

100 is at 5th location. Its parent 80 is at (5-1)/2.

80 < 100 which violates the heap condition.

Heap [5] is swapped with Heap [2] still it violates the heap condition.

New Heap [2] is again swapped with its parent at Heap [0] which is root.

Activity:

1. Write a C program or pseudo code for implementing them.
2. All that discussed above is for Max heap. How implement Min heap?
3. Applications?

Binary search Tree

1. It's a binary tree
2. Key value of any node is always less than its right child nodes' and greater than its left child nodes'.

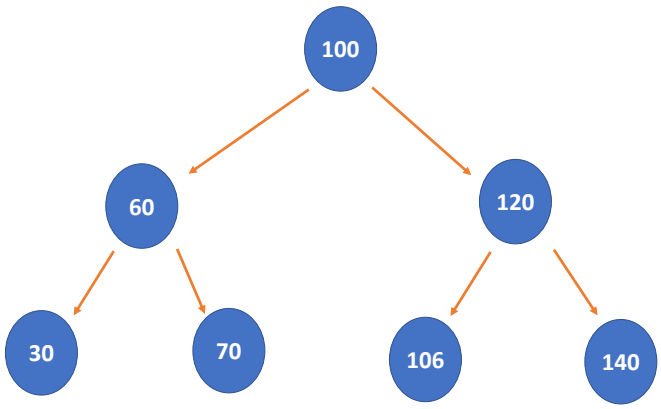


Figure 7: Binary Search Tree

The main purpose of binary search tree is to do major operations on data like adding, deleting, searching with complexity less than O(n). i.e. we should not check with all the elements in our collection. When we organize the data as per the BST criteria these operations can be performed with the complexity ~ O(log n).

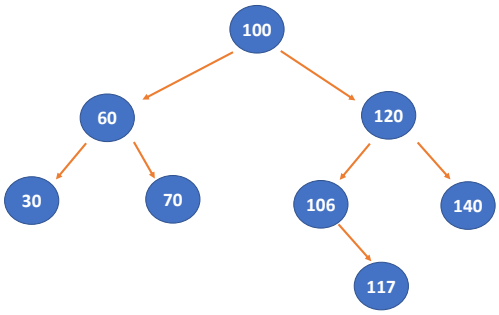
E.g. If we want to access the node 30. First start from the root node with key value 100 which is greater than 30. Now we confine to left subtree of 100 thereby we can eliminate half of the elements.

We only need to compare 30 with 100, 60 and finally 30. For seven elements we only made 3 comparisons.

Adding new node:

For example, if we want add a new node with key value 117,

1. We start from root where node value is 100 which is less than 117. So, we travel towards right subtree.
2. 120 > 117 so we have to change our direction towards left.
3. 106 < 117 and it has no elements attached to it. So, we can add 117 to right of it.

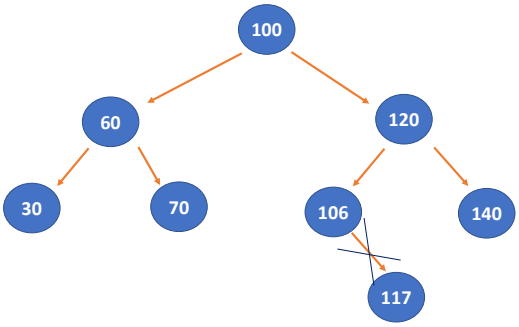


BST allows us to add elements with complexity ~ O (log n)
What will be the complexity if the tree grows in one particular branch lengthy?

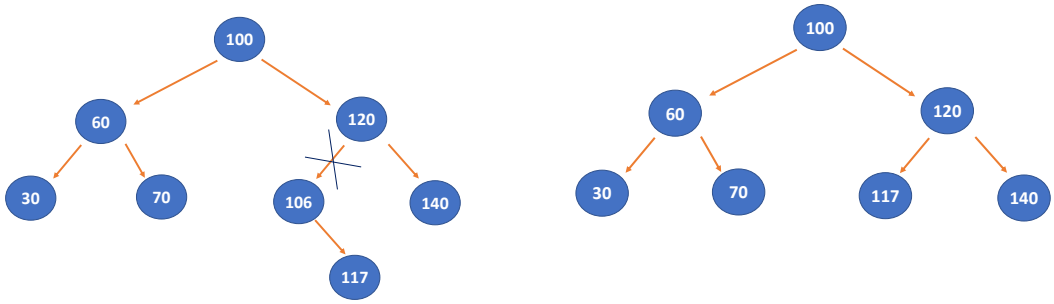
Deleting Node:
Deleting node is not as straight forward as adding a node. Still we can have similar complexity as of adding.
There are three cases we have to consider while deleting any node.

1. Deleting a node that does not have anything attached as child.
2. Deleting a node that has child node only in one direction (either left or right)
3. Deleting a node that has child nodes in both left and right directions.

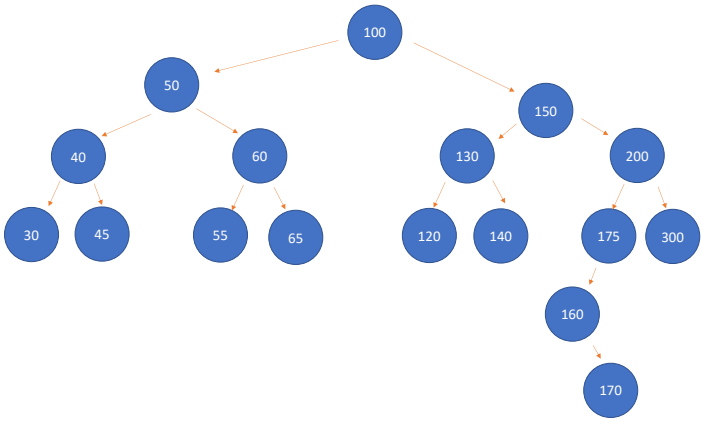
Case 1: We can detach from its parent node and make it free.
If we want to delete node 117



Case 2: We detach the it from its parent node and its child can be attached in that place.
If we want to delete 106

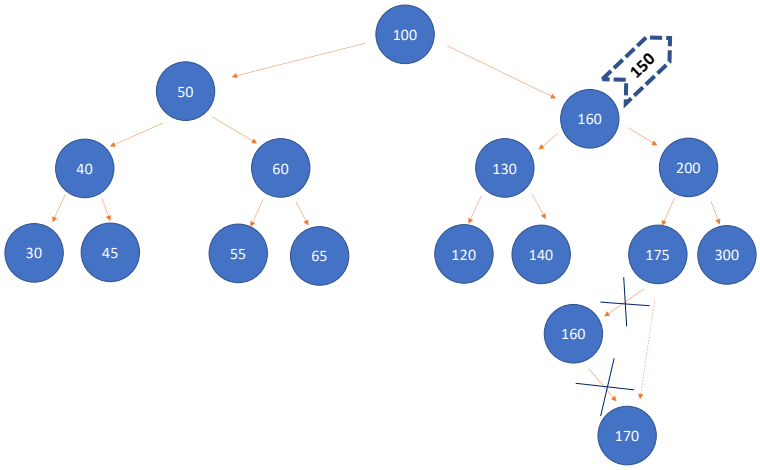


Case 3: Let's consider a bigger tree BST for discussing this case.
How can we delete 150? It has both left sub-tree and right.



As per BST condition for any node we should have its left child lesser value and right child higher value. If we delete 150 what will be the right node to take this position? How about node that has next greater value. All the values greater than 150 are only found in its right sub tree. So next greater value of 150 should be the minimum value of its right sub tree. Minimum value of any BST can be found at left most leaf node. In this case 160 is the left most leaf node. If we make it to replace 150 BST

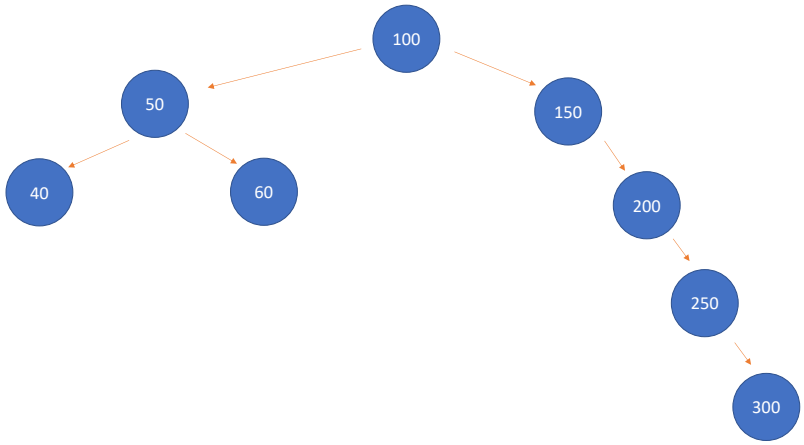
conditions are not disturbed. But what about the nodes attached to it. Here there is only one node, 170, attached to on its right. So, we can make 170 get attached to parent of 160 i.e175.



Is there any possibility of node of minimum value having left child?
No if at all there is any that will be the minimum.

Self-Balancing BST:

Problem with normal BST is there is always a possibility that growth happening in only one direction as shown below.



It is satisfying the BST condition but its right sub-tree is bigger compared to left. To find 300 one has to make five comparisons > log(n=8). So, it should be balanced such that both left and right sub-tree are of same height. Such BSTs are called self-balancing BSTs. One variety of self-balancing binary search trees are AVL trees.

AVL Trees:

1. It's a BST
2. Height of left subtree – Height of right subtree $\in \{-1,0,1\}$

Adding and deleting operation are same as BST. But every time we need to rebalance the tree such that the second condition is meat.