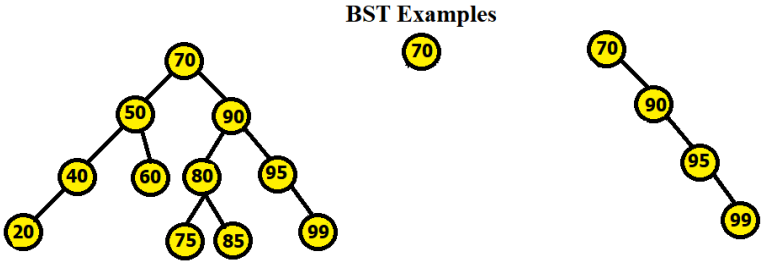
Chapter 6: Binary Search Tree (BST) and Heap

6.1 Characteristics of a BST

Binary Search Tree is a binary tree data with the following fundamental properties:

1. The left subtree of a node contains only nodes with keys lesser than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. The left and right subtree each must also be a binary search tree.
4. Each node must have a distinct key, which means no duplicate values are allowed.

The goal of using BST data structure is to search any element within O(log(n)) time complexity.

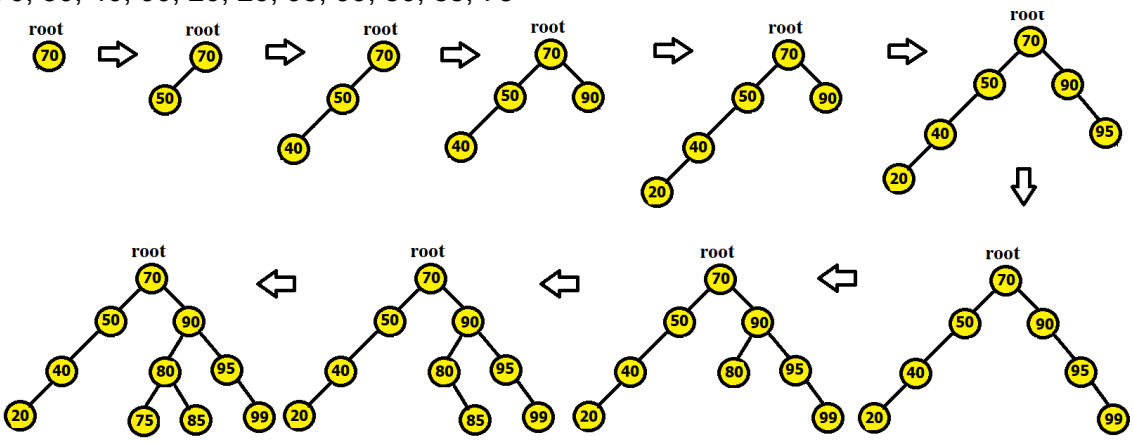


6.2 Basic Operations on a BST

Any operation done in a BST must not violate the fundamental properties of a BST.

6.2.1 Creation of a BST

Draw the BST by inserting the following numbers from left to right:
70, 50, 40, 90, 20, 20, 95, 99, 80, 85, 75



Here second 20 is ignored as it is duplicate value.

6.2.2 Inserting a Node

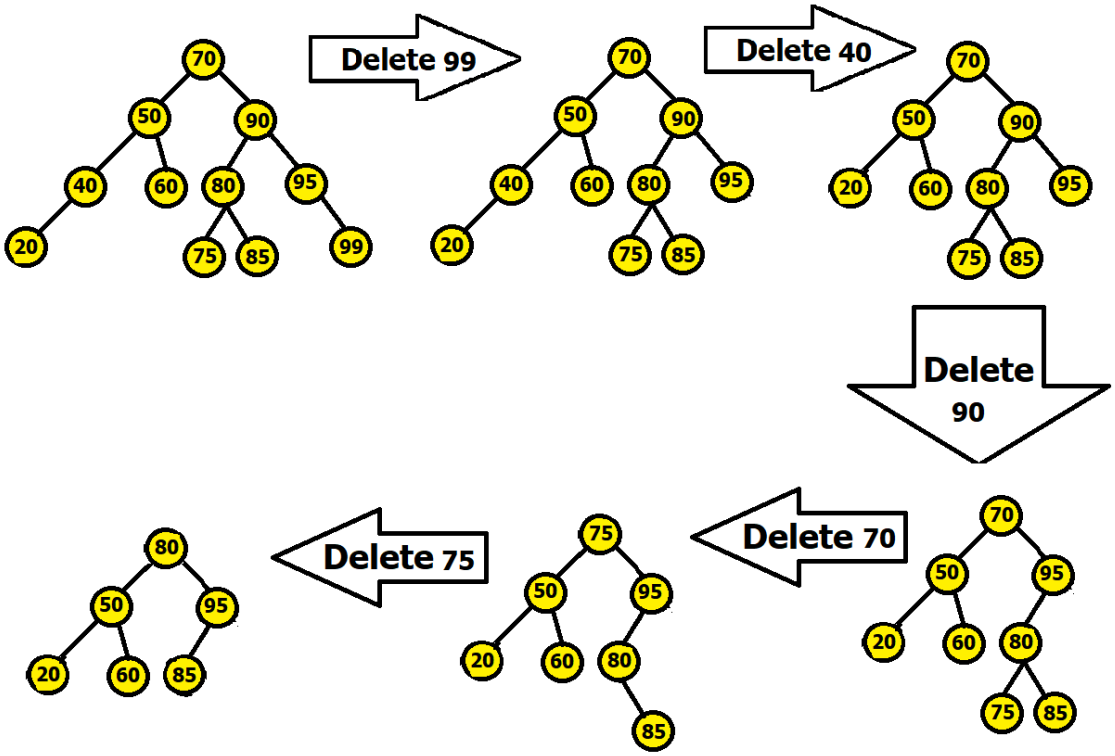
While inserting a node to an existing BST, the process is similar to that of creating a BST. We take the new node data which must not be repetitive or already present in the BST. We start comparing it with the root node, if the new node data is greater we go towards the right subtree of the root, if smaller we go towards the left subtree. We keep on going like this until we find a

free space, and then make a node using the new node data and attach the new node at the vacant place. Note that, after insertion, a balanced BST may become **unbalanced** and therefore any searching operation done on it may take a lot longer than O(log(n)). We have to balance the tree if it becomes unbalanced.

6.2.3 Removing a Node

3 possible cases can occur while deleting a node:

1. *Case 1* | No subtree or children: This one is the easiest one. You can simply just delete the node, without any additional actions required.
2. *Case 2* | One subtree (one child): You have to make sure that after the node is deleted, its child is then connected to the deleted node's parent.
3. *Case 3* | Two subtrees (two children): You have to find and replace the node you want to delete with its leftmost node in the right subtree (inorder successor) or rightmost node in the left subtree (inorder predecessor).



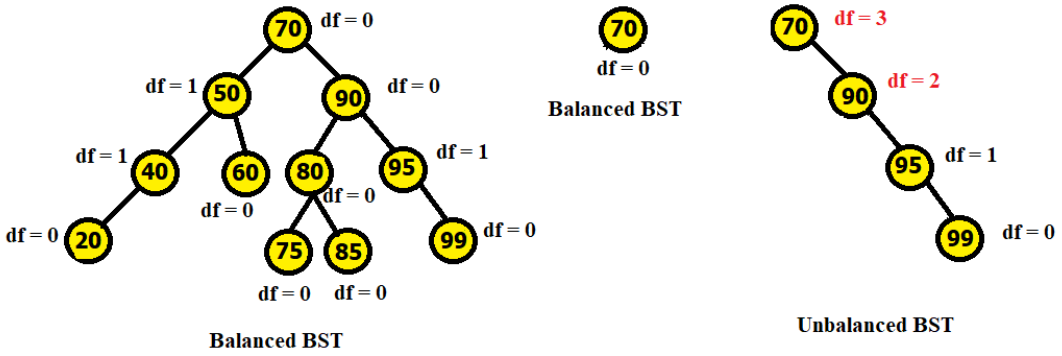
Here deletion of 99 falls under case 1, deletion of 40 falls under case 2, and deletion of 90, 70 and 75 fall under case 3. While deleting 75, we replaced 75 with its leftmost child from its right subtree, 80. After that 85 was put in the 80's previous place.

Note that, after deletion, a balanced BST may become **unbalanced** and therefore any searching operation done on it may take a lot longer than O(log(n)). We have to balance the tree if it becomes unbalanced.

6.3 Balanced vs Unbalanced BST

If the height difference between left and right subtree of any node in the BST is more than one, it is an unbalanced BST. Otherwise, it is a balanced one. In a balanced BST, any searching operation can take upto **O(log(n))** time complexity. In an unbalanced one, it may take upto **O(n)** time complexity, which renders the usage of BST obsolete. Therefore, we should only work with balanced BST and after conducting any operation on a BST, we must first check if it became unbalanced or not. If it does become unbalanced, we have to balance it.

df = | height of left child - height of right child |

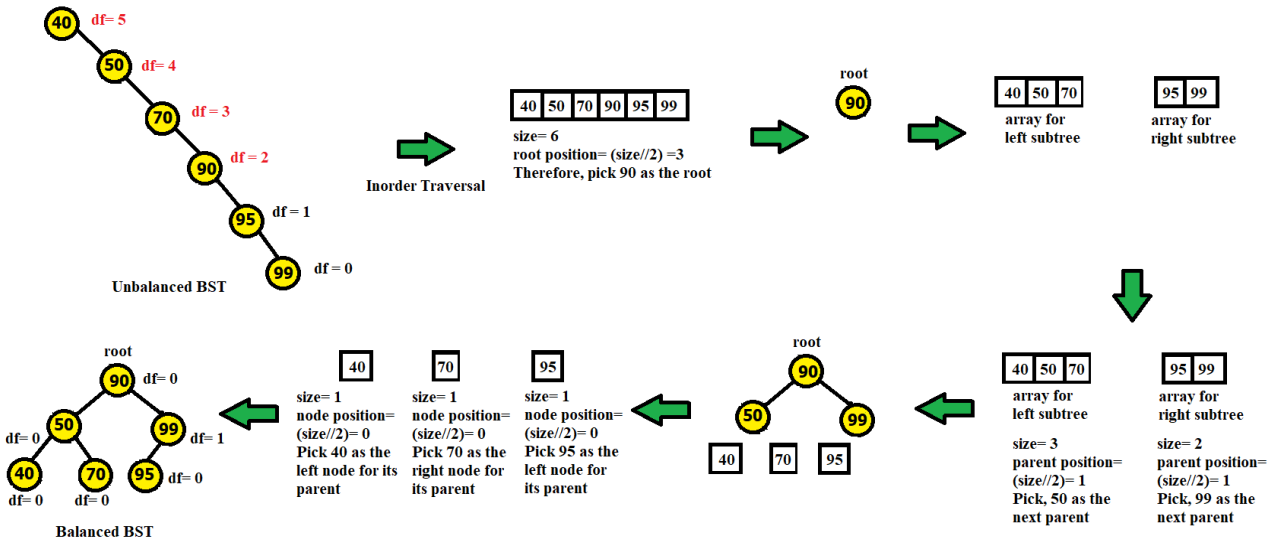


How to convert an unbalanced BST into a balanced BST?

I. Traverse given BST in inorder and store result in an array. This will give us the ascending sorted order of all the data.

II. Now take the data in the (size//2) position in the array and make it the root. Now the left subtree of the root will be all the data residing in from 0 to (size//2)-1 positions of the array. The right subtree of the root will be all the data residing in from (size//2)+1 to (size-1) positions of the array.

III. Now again choose the middlemost values from the left subtree and right subtree and connect these to the root. Keep on repeating the process until all the elements of the array have been taken.



6.4 BST Coding

6.4.1 Creating a BST / Inserting a Node

Pseudocode

```
FUNCTION insert(node, key):
   IF node IS NULL:
      RETURN new Node(key)

   IF key < node.key:
      node.left = insert(node.left, key)
   ELSE IF key > node.key:
      node.right = insert(node.right, key)

   RETURN node
```

6.4.2 BST Traversal: Pre-order, In-order, Post-order

Pre-Order	In-Order	Post-Order

function preorder(root)
  if root is null
  return
  print root.data
  preorder(root.left)
  preorder(root.right)
	function inorder(root)
	  if root is null
	  return
	  inorder(root.left)
	  print(root.data)
	  inorder(root.right)
		function postorder(root)
		  if root is null
		  return
		  postorder(root.left)
		  postorder(root.right)
		  print(root.data)

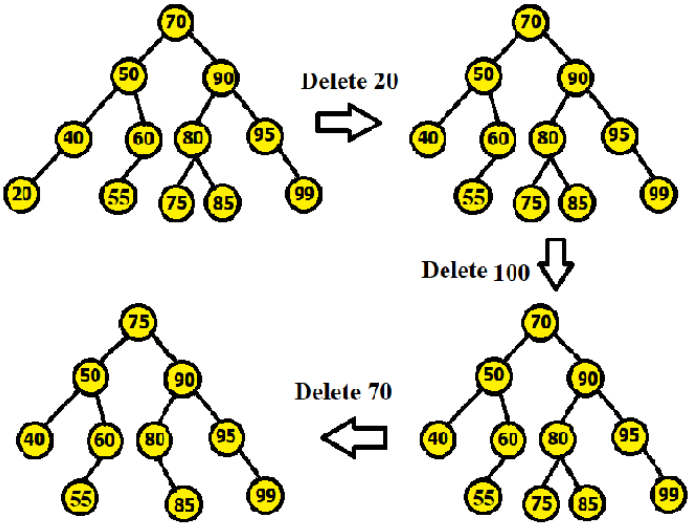
6.4.3 Searching for an element

```
FUNCTION search(node, key):
   IF node IS NULL:
       RETURN False

   IF node.key == key:
      RETURN True

   IF key < node.key:
      RETURN search(node.left, key)
   ELSE:
      RETURN search(node.right, key)
```

6.4.4 Removing a Node

```
FUNCTION deleteUsingSuccessor(node, key):
    IF node IS NULL:
        RETURN NULL

    IF key < node.key:
        node.left = deleteUsingSuccessor(node.left, key)
    ELSE IF key > node.key:
        node.right = deleteUsingSuccessor(node.right, key)
    ELSE:
        // Node with only one child or no child
        IF node.left IS NULL:
            RETURN node.right
        ELSE IF node.right IS NULL:
            RETURN node.left

        // Node with two children:
        // Get inorder successor (smallest in the right subtree)
        successor = findMin(node.right)
        node.key = successor.key
        node.right = deleteUsingSuccessor(node.right, successor.key)

    RETURN node

FUNCTION findMin(node):
    WHILE node.left IS NOT NULL:
        node = node.left
    RETURN node



FUNCTION deleteUsingPredecessor(node, key):
    IF node IS NULL:
        RETURN NULL

    IF key < node.key:
        node.left = deleteUsingPredecessor(node.left, key)
    ELSE IF key > node.key:
        node.right = deleteUsingPredecessor(node.right, key)
    ELSE:
        // Node with only one child or no child
        IF node.left IS NULL:
            RETURN node.right
        ELSE IF node.right IS NULL:
            RETURN node.left

        // Node with two children:
        // Get inorder predecessor (largest in the left subtree)
```

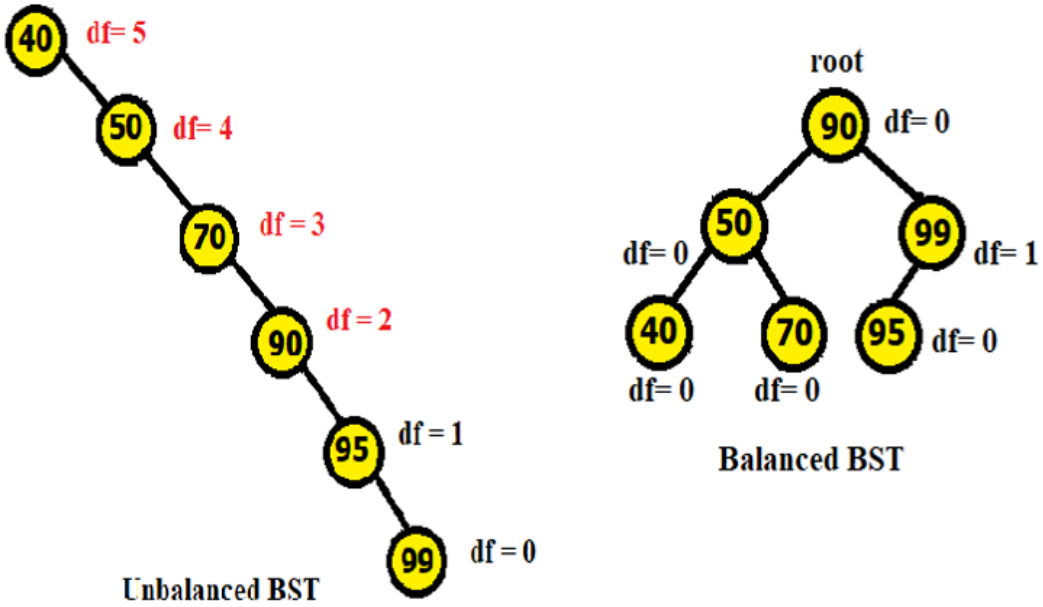
```
        predecessor = findMax(node.left)
        node.key = predecessor.key
        node.left = deleteUsingPredecessor(node.left, predecessor.key)

    RETURN node

FUNCTION findMax(node):
    WHILE node.right IS NOT NULL:
        node = node.right
    RETURN node
```



6.4.5 Balancing BST

```
FUNCTION storeInorder(node, array, index):
    IF node IS NULL:
        RETURN index
    index = storeInorder(node.left, array, index)
    array[index] = node.key
    index = index + 1
    index = storeInorder(node.right, array, index)
    RETURN index

FUNCTION buildBalancedBST(array, start, end):
    IF start > end:
```

```
    RETURN NULL
  mid = (start + end) / 2
  node = new Node(array[mid])
  node.left = buildBalancedBST(array, start, mid - 1)
  node.right = buildBalancedBST(array, mid + 1, end)
  RETURN node

FUNCTION balanceBST(root):
  size = countNodes(root)
  array = new Array of size
  storeInorder(root, array, 0)
  RETURN buildBalancedBST(array, 0, size - 1)
```

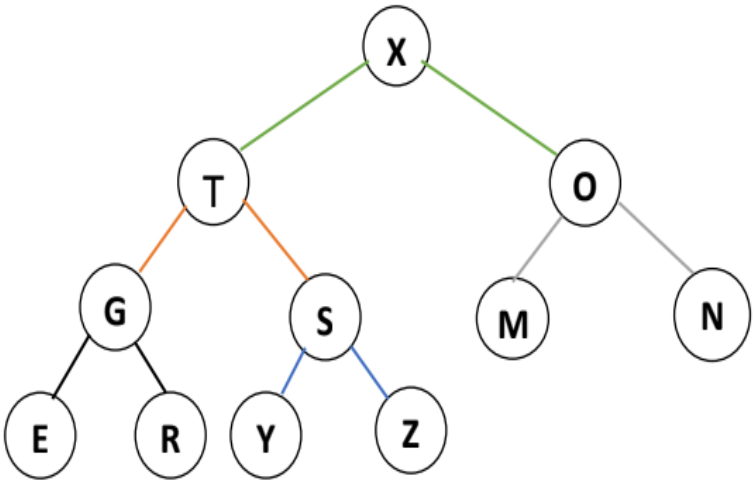


6.5 Heap

Heap is an ADT for storing values. A heap is expressed as a special binary tree pictorially and as its underlying data structure it uses an array. The tree gives heap an advantage to manipulate using a pen and paper quite easily which we will see as we progress. Let me break down the "special tree" as mentioned. A heap has to be a complete binary tree and it must satisfy the heap property. In previous chapter we have learned about Queue which maintains the FIFO property. If we want to maintain a priority queue where we will maintain FIFO property based on a priority, heap is our underlying data structure.

6.6 Heap Property

The value of the parent must be greater than or equal to the values of the children. (Max heap). or the value of the parent must be smaller than or equal to the values of the children. (Min heap). There are two types of heaps. Max heap is mostly used. A heap can be either a max heap or a min heap but can't be both at the same time.



The above tree is a heap. Below is the array representation of the above heap. Please note that the tree is used for efficient tracing. While programming the data structure is a simple array. Below is the array representation of the above heap. We start the index from 1.

1	2	3	4	5	6	7	8	9	10	11	12
X	T	O	G	S	M	N	E	R	Y	Z	C

For i^{th} NODE, its parent NODE is i/2.
For i^{th} NODE, its children are 2i and 2i+1.
Note: 2i is the LEFT child and 2i+1 is the RIGHT child.

Benefit of using ARRAY for Heap rather than Linked List

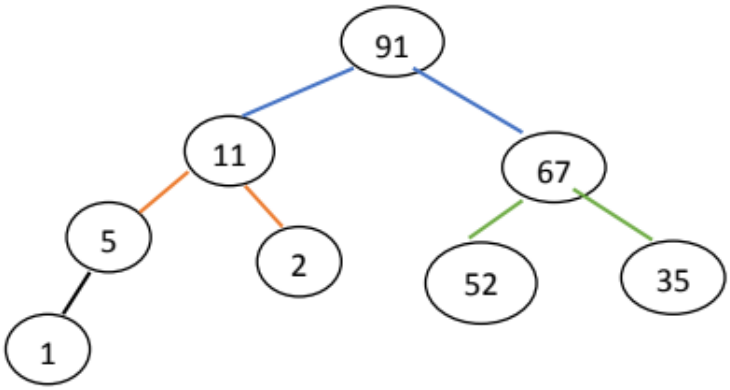
ARRAYS give you random access to its elements by indices. You can just pick any element from the ARRAY by just calling the corresponding index. Finding a parent and its children is trivial. Linked List is sequential. This means you need to keep visiting elements in the linked

list unless you find the element you are looking for. Linked List does not allow random access as ARRAY does. On the other hand each linked list must have three (3) references to traverse the whole Tree (Parent, left, Right).

6.7 Operations on Heap

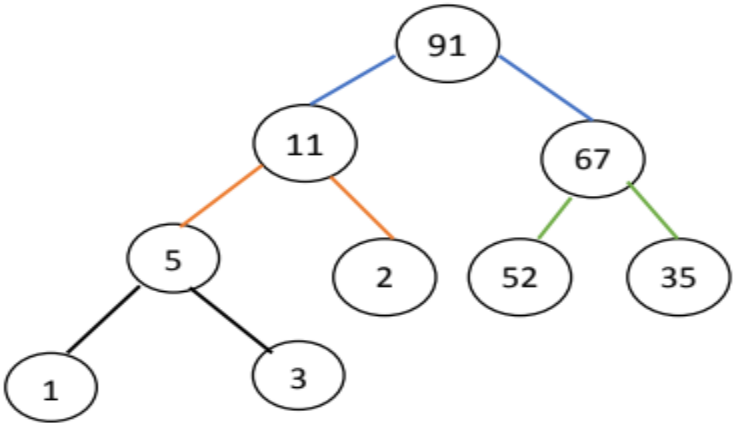
6.7.1 Insert

Inserts an element at the bottom of the Heap. Then we must make sure that the Heap property remains unchanged. When inserting an element in the Heap, we start from the left available position to the right.



Consider the above Heap. If we want to insert an element 3, we start left to right at the bottom level. Therefore 3 will be added as a child of 5.

Then the new Heap will look like this:

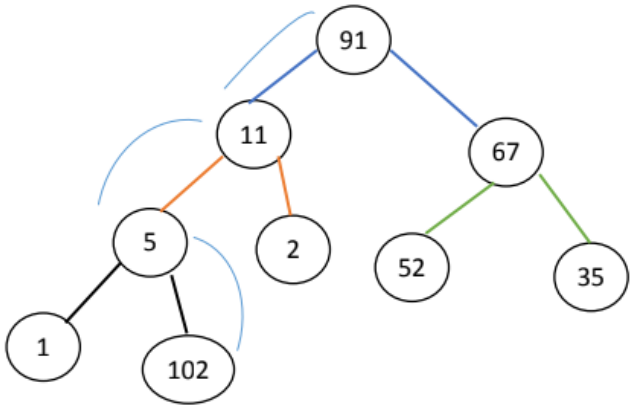


Look carefully five (5) is 3's parent and it is larger. Hence Heap property is kept intact.

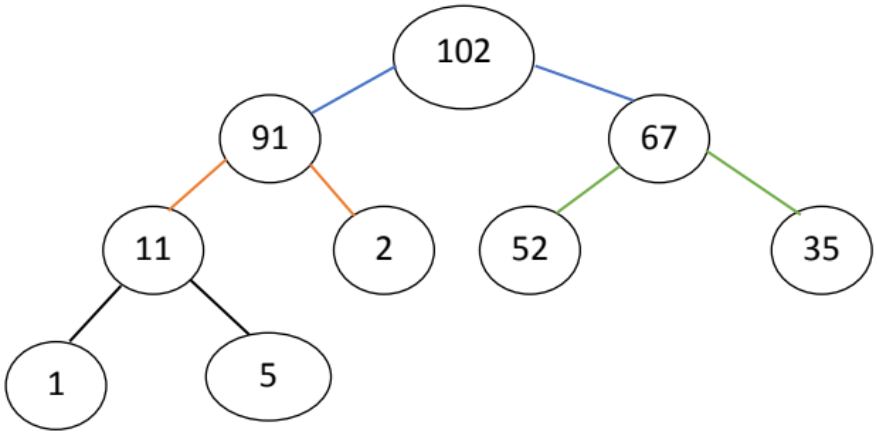
What if we want to insert 102 instead of 3?

Let's say we want to insert 102 at the existing Heap. 102 will be added as a child of 5. Now is the Heap property hold intact? Therefore we need to put 102 in its correct position. How we going to do it? The methodology is called HeapIncreaseKey () or swim().

HeapIncreaseKey ()/swim(): Let the new NODE be 'n' (in this case it is the node that contains 102). Check 'n' with its parent. If the parent is smaller (n > parent) than the node 'n', replace 'n' with the parent. Continue this process until n is its correct position.



After the swim() operation the Heap will look like this:



Time Complexity: Best case O(1) when a key is inserted in the correct position at the first go. Worst case is when the newest node needs to climb up to the root. We have learnt that the distance from the leaf node to the root is lg(n) (height of the tree). Hence this is the worst case complexity. O(1) [insertion] + O(lgn) [swim]

Thus insert is a combination of two functions: insert() and swim().

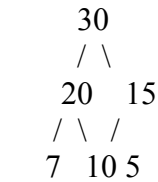
Pseudocode:

```
function insert(heap, value):
   append value to heap
   swim(heap, size of heap - 1)
   inc heap size

function swim(heap, index):
   while index > 0 and heap[index] > heap[parent(index)]:
      swap heap[index] with heap[parent(index)]
      index = parent(index)

function parent(index):
   return (index - 1) // 2
```

6.7.2 Delete

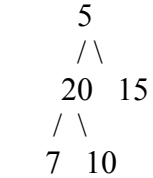
In heap you cannot just cannot randomly delete an item. Deletion is done by replacing the root with the last element. The Heap property will be broken 100%. Small value will be at the top (root) of the Heap. Therefore we must put it in a right place which is definitely somewhere down the Tree.

Initial Max Heap: [30, 20, 15, 7, 10, 5]



Step 1: Swap root with last (5), delete last

After swap: [5, 20, 15, 7, 10] (heapify to follow)



Step 2: Heapify from index 0

After heapify:

```
    20
    / \
  10   15
 /
7
```

Final Heap: [20, 10, 15, 7, 5]

This process of putting a node in its correct place by traveling downward is called sink() or MaxHeapify(). The time complexity of sink is lgn as it might have to sink down to the end of the tree.

Delete is a combination of delete() and sink() hence the worst time complexity is lgn too.

Pseudocode:

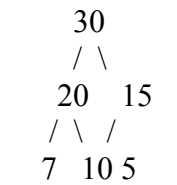
```
function deleteMax(heap):
    if heap is empty:
        return error
    maxValue = heap[0]
    swap heap[0] with heap[last index]
    remove last element from heap
    dec heap size
    heapify(heap, 0)
    return maxValue

function heapify(heap, index):
    size = size of heap
    largest = index
    left = 2 * index + 1
    right = 2 * index + 2

    if left < size and heap[left] > heap[largest]:
        largest = left
    if right < size and heap[right] > heap[largest]:
        largest = right

    if largest != index:
        swap heap[index] with heap[largest]
        heapify(heap, largest)
```

6.8 Heap Sort

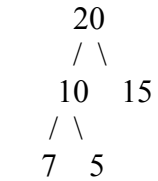
Delete all the nodes of the heap.

Initial Array: [30, 20, 15, 7, 10, 5]

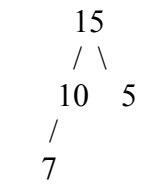
Initial Heap:



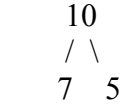
Step 1: Swap 30 and 5 → [5, 20, 15, 7, 10, 30], then heapify



Step 2: Swap 20 and 5 → [5, 10, 15, 7, 20, 30], then heapify



Step 3: Swap 15 and 7 → [7, 10, 5, 15, 20, 30], then heapify



Step 4: Swap 10 and 5 → [5, 7, 10, 15, 20, 30], then heapify

Step 5: Swap 7 and 5 → [5, 7, 10, 15, 20, 30]

Build Max Heap: You are given an arbitrary array and you have been asked to built it a heap.

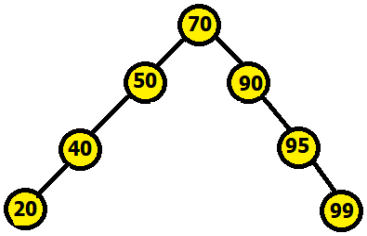
This will take O(nlgn).

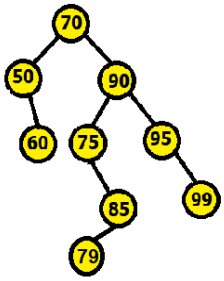
Pseudocode:

```
function heapsort(array):
   buildMaxHeap(array)

   for i from size of array - 1 down to 1:
      swap array[0] with array[i]
      heapify(array, 0, i)

function buildMaxHeap(array):
   for i from (size of array // 2) - 1 down to 0:
      heapify(array, i, size of array)

function heapify(array, index, size):
   largest = index
   left = 2 * index + 1
   right = 2 * index + 2

   if left < size and array[left] > array[largest]:
      largest = left
   if right < size and array[right] > array[largest]:
      largest = right

   if largest != index:
      swap array[index] with array[largest]
      heapify(array, largest, size)
```

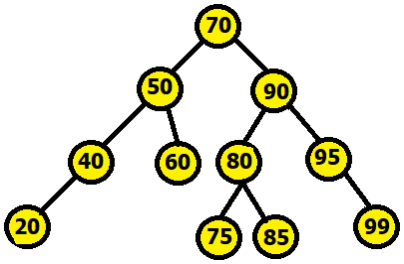
Exercises

6.1: Convert the following unbalanced BSTs into balanced BSTs. Show simulation.

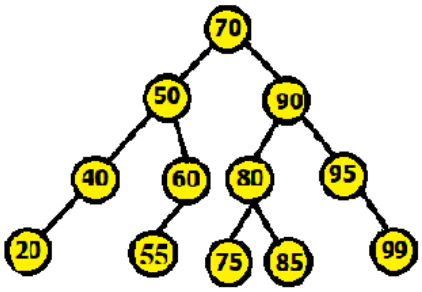




6.2: Insert keys 65, 105, 69 into the following BST and show the steps. Show simulation and code.

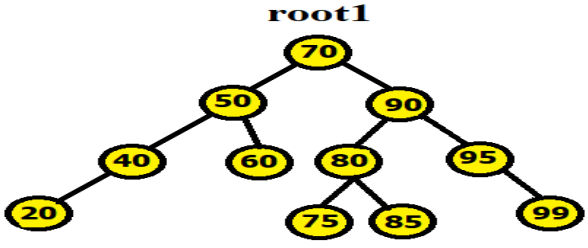


6.3: Delete keys 20, 95, 50, 70, 75 into the following BST and show the steps. Show simulation and code..



6.4: How can you print the contents of a tree in descending order with and without using stack? Solve using code.

6.5: Write a python program that takes the root of a tree and finds its inorder successor and predecessor.



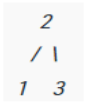
Output: In-order Successor: 75
In-order Predecessor: 60

6.6: Given a sorted array, write a function that creates a Balanced Binary Search Tree using array elements. Follow the steps mentioned below to implement the approach:

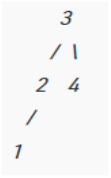
1. Set The middle element of the array as root.
2. Recursively do the same for the left half and right half.
3. Get the middle of the left half and make it the left child of the root created in step 1.
4. Get the middle of the right half and make it the right child of the root created in step 1.
5. Print the preorder of the tree.

Given Array#1: [1, 2, 3]
Output: Pre-order of created BST: 2 1 3

BST#1



BST#2

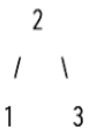


Given Array#2: [1, 2, 3,4]
Output: Pre-order of created BST: 3 2 1 4

6.7: Given the root of a binary tree, check whether it is a BST or not. A BST is defined as follows:

A. The left subtree of a node contains only nodes with keys less than the node's key.
B. The right subtree of a node contains only nodes with keys equal or greater than the node's key.
C. Both the left and right subtrees must also be binary search trees.

Input:



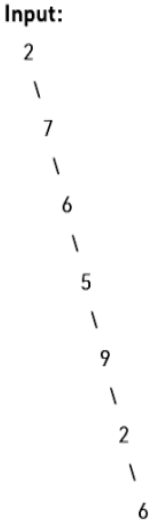
Output: 1

Explanation:

The left subtree of root node contains node with key lesser than the root nodes key and the right subtree of root node contains node with key greater than the root nodes key.

Hence, the tree is a BST.

Input:



Output: 0

Explanation:

Since the node with value 7 has right subtree nodes with keys less than 7, this is not a BST.

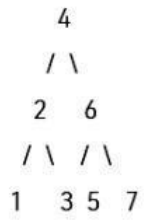
6.8: Given a sorted array. Write a function that creates a Balanced Binary Search Tree using array elements. Height balanced BST means a binary tree in which the depth of the left subtree and the right subtree of every node never differ by more than 1

Input: nums = {1,2,3,4,5,6,7}

Ouput: {4,2,1,3,6,5,7}

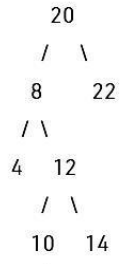
Explanation:

The preorder traversal of the following BST formed is {4,2,1,3,6,5,7} :



6.9: Given a BST, and a reference to a Node x in the BST. Find the Inorder Successor of the given node in the BST.

Input:



K(data of x) = 8

Output: 10

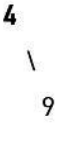
Explanation:

Inorder traversal: 4 8 10 12 14 20 22

Hence, successor of 8 is 10.

6.10: Given a Binary search tree, your task is to complete the function which will return the Kth largest element without doing any modification in the Binary Search Tree.

Input:



k = 2

Output: 4