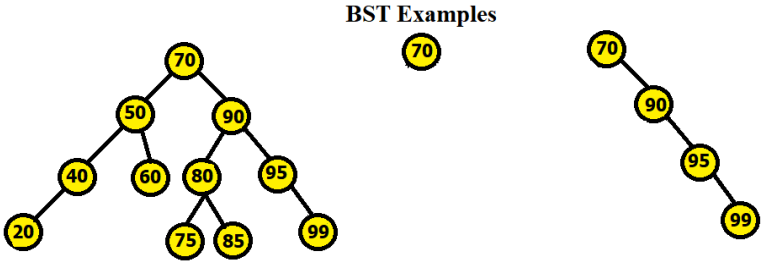
Chapter 6: Binary Search Tree (BST) and Heap

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.

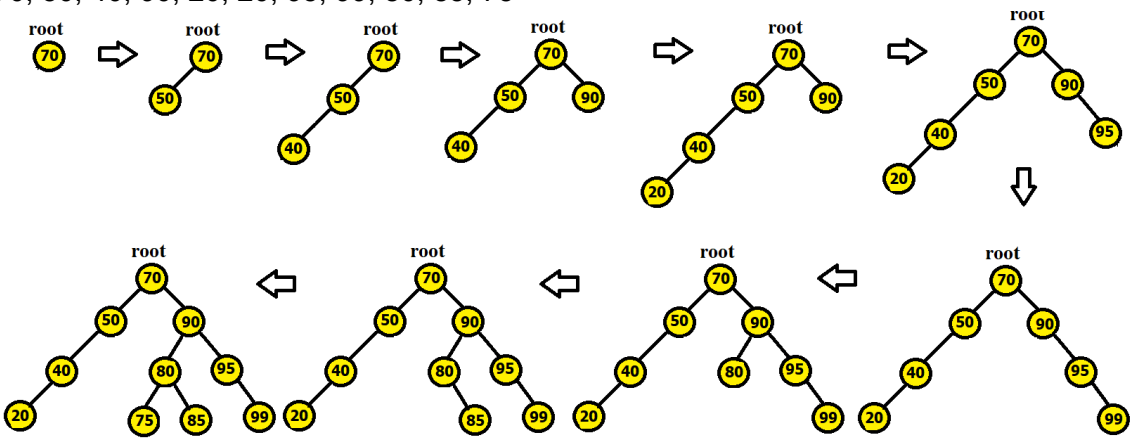


6.2 Basic Operations on a BST

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

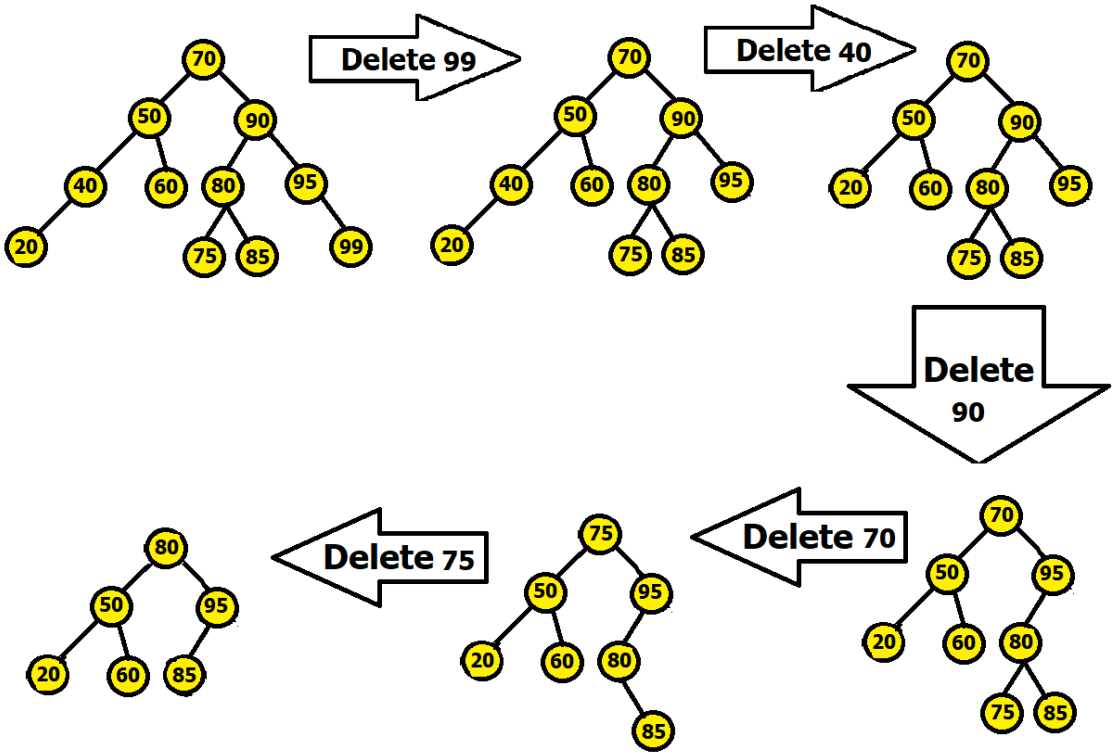
While inserting a node to an existing BST, the process is similar to that of creating a BST. We take the new node data which must not be repetitive or already present in the BST. We start comparing it with the root node, if the new node data is greater we go towards the right subtree of the root, if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



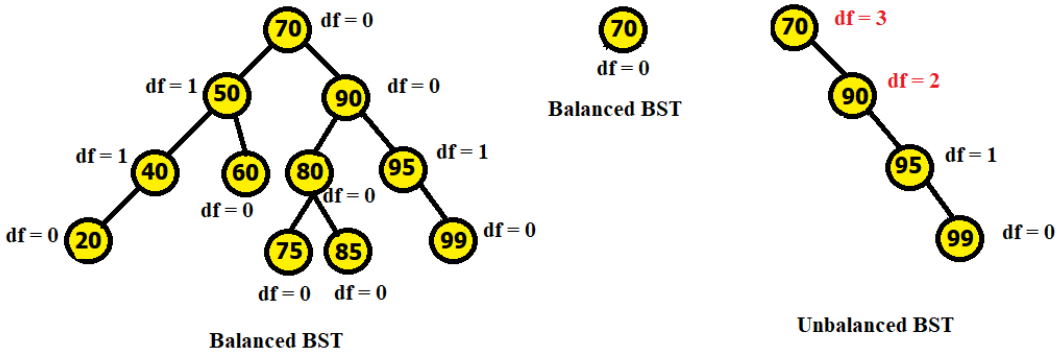
Here deletion of 99 falls under case 1, deletion of 40 falls under case 2, and deletion of 90, 70 and 75 fall under case 3. While deleting 75, we replaced 75 with its leftmost child from its right subtree, 80. After that 85 was put in the 80's previous place.

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 |

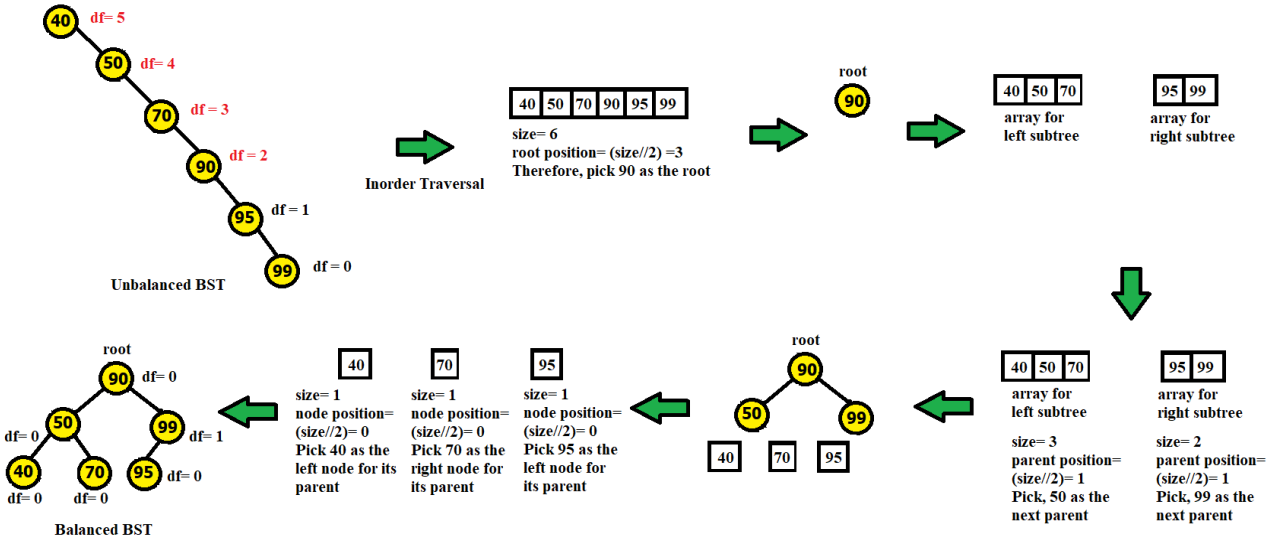


How to convert an unbalanced BST into a balanced BST?

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)

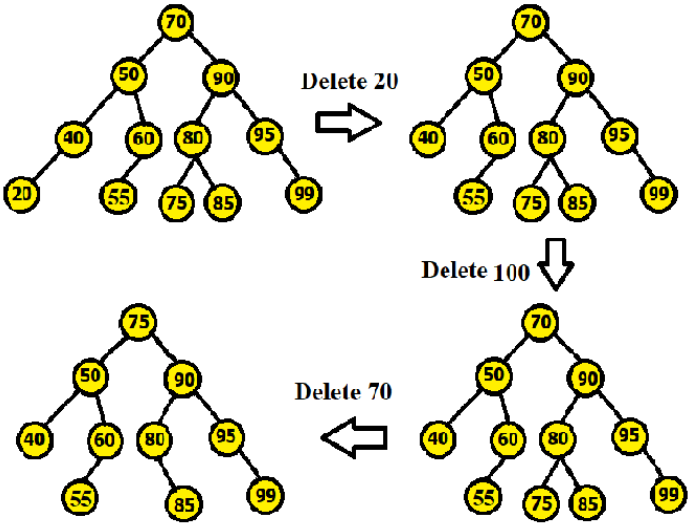
6.4.3 Searching for an element

```
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)
```

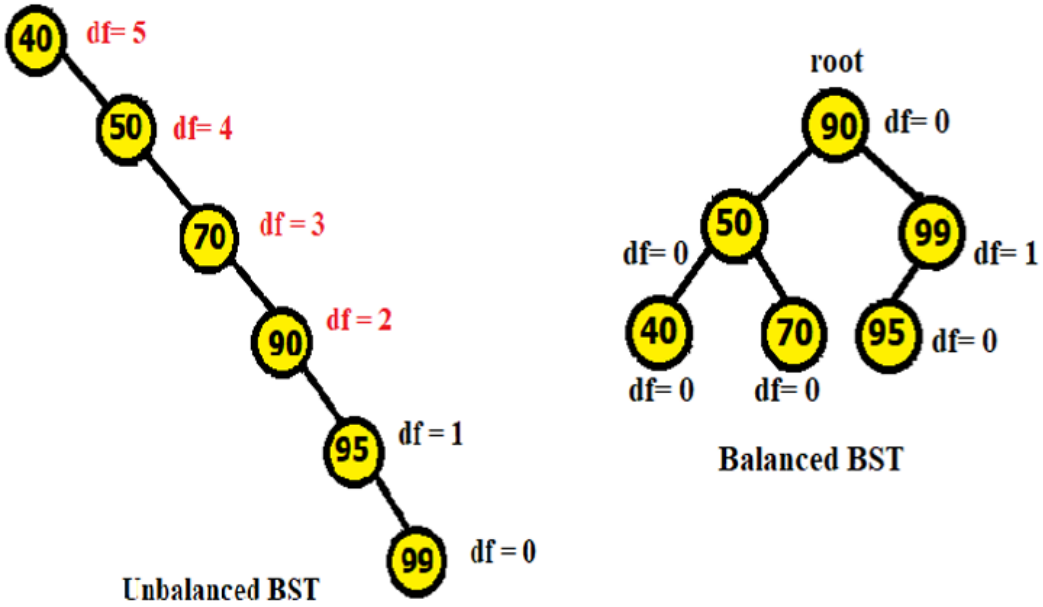
```
        predecessor = findMax(node.left)
        node.key = predecessor.key
        node.left = deleteUsingPredecessor(node.left, predecessor.key)

    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

A **heap** is an abstract data type (ADT) used for storing values. It is visually represented as a special **binary tree**, but its underlying data structure is an **array**. Arrays enable **random access** via indices, making parent and child lookups straightforward. The tree structure makes heaps intuitive to visualise and manipulate with pen and paper, which we will explore further.

Let's break down this "special tree":
A heap must be a **complete binary tree** and must satisfy the **heap property**.

In the previous chapter, we studied **queues**, which follow the **FIFO (First In, First Out)** principle. If we want to implement a **priority queue**—which maintains the FIFO order based on priority—a **heap** is the ideal underlying data structure.

6.5.1 Heap Property

A heap is a specialised binary tree that satisfies the heap property and can be of two types: max-heap or min-heap. Unless otherwise specified, a heap refers to a max-heap in this course.

- In a max-heap, for every node $i > 1$, the value of the parent node satisfies:
  A[Parent(i)] ≥ A[i].
  The maximum element is at the root.

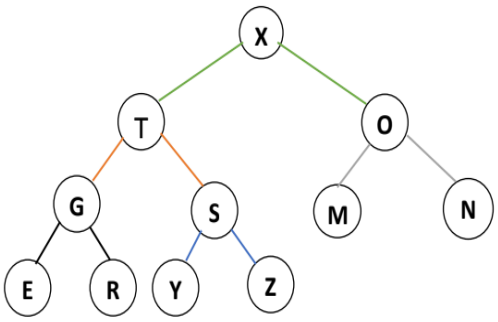
- In a min-heap, for every node $i > 1$, the value of the parent node satisfies:
  A[Parent(i)] ≤ A[i].
  The minimum element is at the root.

A heap can be either a max-heap or a min-heap, but not both simultaneously.

Heap as an array:
A heap is a complete binary tree that can be efficiently represented using an array, eliminating the need to store edges explicitly.

- For a node at index i:
  - Parent: $\lfloor i/2 \rfloor$
  - Left child: $2i$
  - Right child: $2i + 1$

Similar to binary search tree (BST) conventions, heap indexing starts at 1. While the tree structure aids in visualisation, the underlying data structure used in programming is a simple array. This array-based representation allows constant-time access to parent and child nodes.



The above tree is a heap. Below is the array representation of the above heap.

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

Benefits of Using an Array for Heap Instead of a Linked List

Arrays provide random access to elements via indices, allowing direct access to any element by its index. This makes finding a parent or its children straightforward.

In contrast, a linked list is sequential. To locate a specific element, you must traverse the list node by node, as it does not support random access like arrays do. Additionally, representing a tree using linked lists requires three references per node: one each for the parent, left child, and right child.

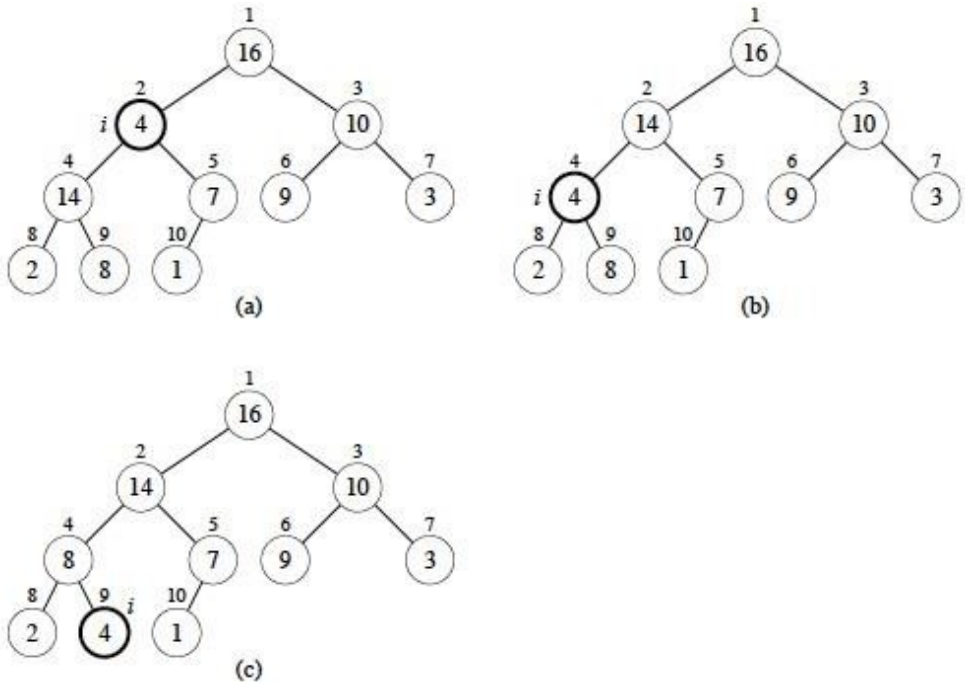
6.5.2 Operations on Heap

1. max-heapify(A, i): Ensures the max-heap property of the array A, starting at node i. Also known as the “sink” operation, as it sinks smaller elements down the tree.
2. heap-increase-key(A, i, key): Increases the value of the element at node i to key and restores the max-heap property of A by moving larger elements upward. Also known as the “swim” operation, as it moves larger elements up the tree.
3. max-heap-insert(A, key): Inserts key into the heap A, maintaining the max-heap property.
4. get-maximum(A): Returns the element with the largest key in the heap.
5. heap-extract-max(A): Extracts and returns the largest element from the heap A, maintaining the heap property.
6. build-max-heap(A): Builds a max-heap from an unordered array A.
7. Heapsort (A): Sorts the elements in array A using heap operations.

6.5.2.1 max-heapify() / Sink():

This operation ensures the max-heap property for the subtree rooted at node i. It is also known as the "sink" operation because it moves smaller elements down the tree. The precondition for this operation is that the subtrees rooted at the left and right children of i are already max-heaps. After executing max-heapify(A, i), the subtree rooted at i will satisfy the max-heap

property, where the value at i is greater than or equal to its children, and both subtrees remain max-heaps.



Note: Image from Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms*, 3rd edition, MIT Press, 2009.

In the figure above, (a) shows the result of calling max_heapify() on index 2, which contains the value 4. Since 4 is smaller than both of its children, we compare the values of the children. The largest child is 14, located at index 4. Therefore, in (b), the values at index 2 and index 4 are swapped.

Next, max_heapify() is called on index 4, where the value is now 4. Since 4 is smaller than its right child at index 9, which has the value 8, the values at index 4 and index 9 are swapped. This is shown in (c).

Time Complexity: The operation has a time complexity of O(log n), as it may require traversing the height of the tree in the worst case.

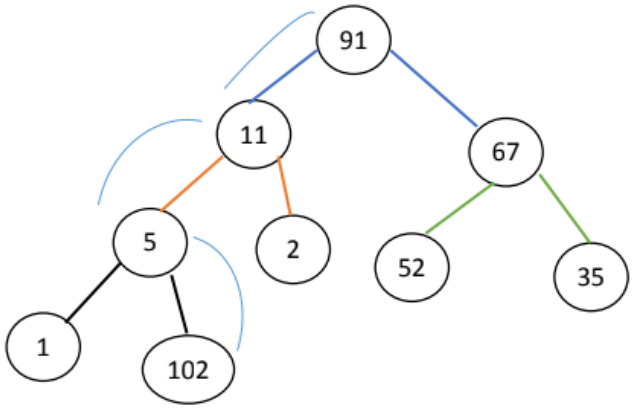
Pseudocode:

```
function max-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]:
```

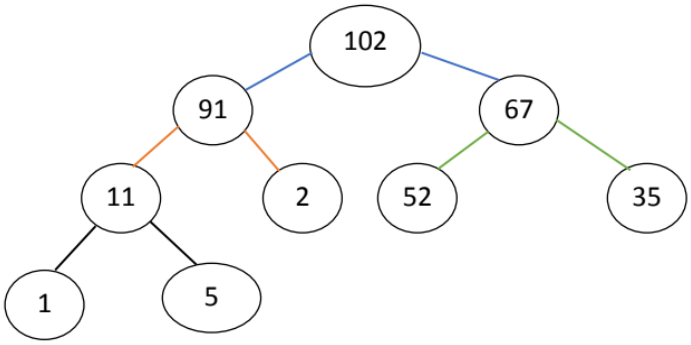
```
    largest = right

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

6.5.2.2 Heap_Increase_Key() / swim():

The heap-increase-key(A, i, key) operation increases the value of the element at index i to key and restores the max-heap property by moving the updated element upward through the heap. This process, known as the "swim" operation, repeatedly compares the updated node (n) with its parent and swaps them if the parent is smaller. This continues until n is no longer greater than its parent or it becomes the root. The operation ensures that the max-heap structure is maintained after increasing a key.



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



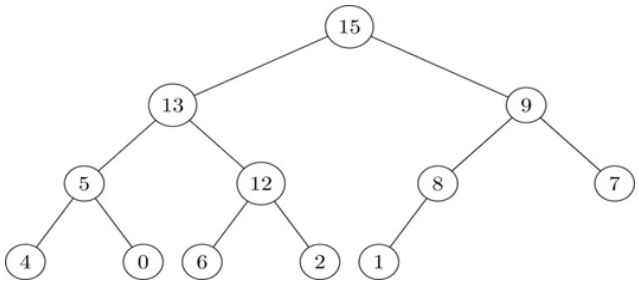
Time Complexity: The best-case time complexity of heap-increase-key is O(1) when the new key is already in the correct position. The worst case is O(log n), occurring when the updated node must "swim" up to the root. Thus, the overall time complexity is O(log n).

6.5.2.3 **max-heap-insert()**

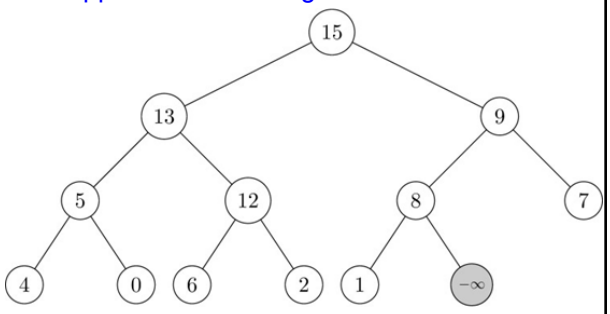
In a max-heap, insertion is performed by placing the new element at the next available position—i.e., at the end of the array (heap-size + 1), maintaining the complete binary tree structure. Initially, a placeholder value (e.g., -∞) is inserted, and then the actual key is set using heap-increase-key(), which adjusts the position of the new key by moving it upward (swim) until the max-heap property is restored. This ensures that the largest element remains at the root.

1. Original heap creation from the given array

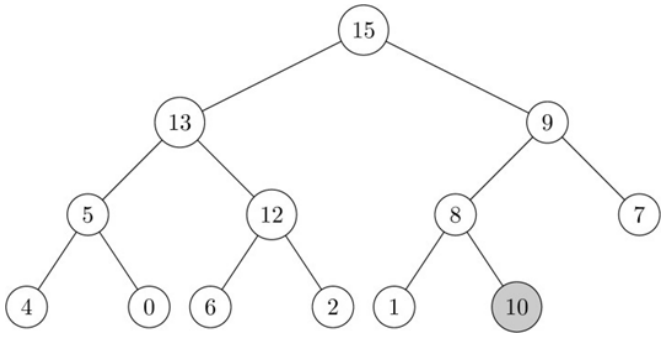
A=⟨ 15, 13 , 9, 5, 12, 8, 7, 4, 0, 6, 2, 1, NONE⟩



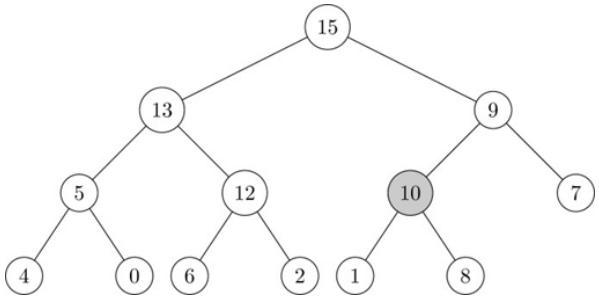
2. Since MAX-HEAP-INSERT(A,10) is called, we append a node assigned value −∞



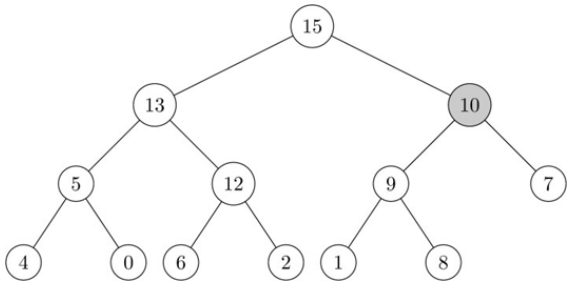
3. Update the key value of the new node.



4. Update the key value of the new node.



5. Since the parent key is smaller than 10, the nodes are swapped. (Swim up)



Pseudocode:

```
Function heap-increase-key(A, i, key)
    if key < A[i]
        error "new key is smaller than current key"
    A[i] = key
    while i > 1 and A[parent(i)] < A[i]
        exchange A[i] with A[parent(i)]
        i = parent(i)

Function parent(index):
   return index // 2

Function max-heap-insert(A, key):
    heap-size[A] = heap-size[A] + 1
    A[heap-size[A]] = -∞
    heap-increase-key(A, heap-size[A], key)
```

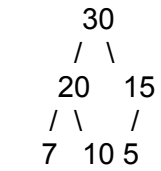
6.5.2.4 **get_maximum()**

Returns the element with the largest key from the max-heap A.
Time Complexity: O(1) (The maximum element is always at the root: A[1]).

6.5.2.5 **Heap-extract-max ()**

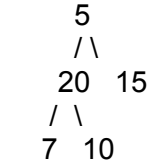
In a heap, you cannot delete an element randomly. Deletion is done by replacing the root with the last element in the heap, which may break the heap property. The heap will need to be restored by "sinking" the new root to its correct position.

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



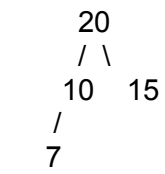
Step 1: Swap root (30) with last element (5), then delete the last element.

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



Step 2: Call max-heapify from index 1 (heapify down).

After heapify:



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

Time Complexity: heap-extract-max() has a time complexity of O(log n) due to the max_heapify() operation, which may traverse the height of the tree in the worst case.

Pseudocode:

```
function get_maximum(A):
    if heap is empty:
        return error
    return A[1]

function heap-extract-max(heap):
    if heap is empty:
        return error
    maxValue = heap[1]
    swap heap[1] with heap[last index]
    remove last element from heap
    dec heap size
    max-heapify(heap, 1)
    return maxValue
```

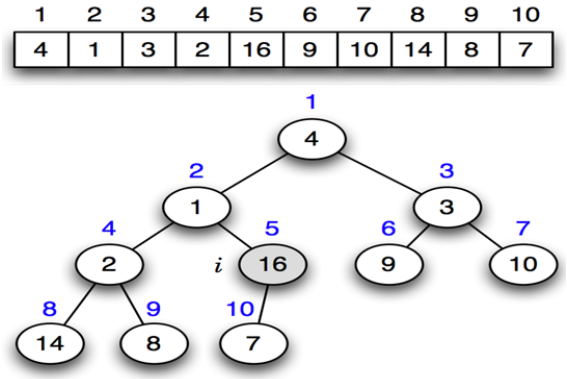
6.5.2.6 build-max-heap()

Build-Max-Heap(A) is used to convert an unsorted array into a valid max-heap, where each parent node is greater than or equal to its children. Starting with an array A[1..n], the algorithm applies the max-heapify operation from the last internal node up to the root. This ensures that each subtree satisfies the max-heap property.

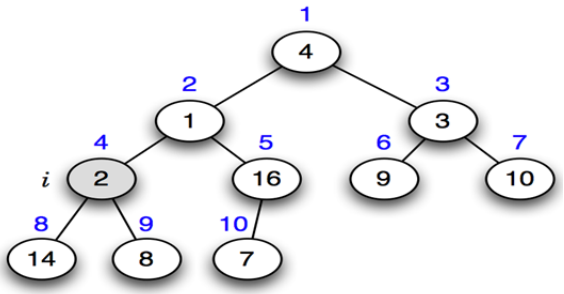
Pseudocode:

```
function buildMaxHeap(array):
    for i from ⌊size of array/2⌋ down to 1:
        heapify(array, i)
```

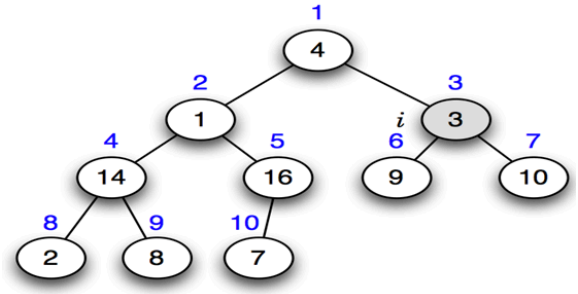
1. The heap has a length of 11, so the first internal node is at index $\lfloor 11/2 \rfloor = 5$.
At index 5, the value is 16, which is greater than its child at index 10 (value 7). Therefore, no swap is needed.



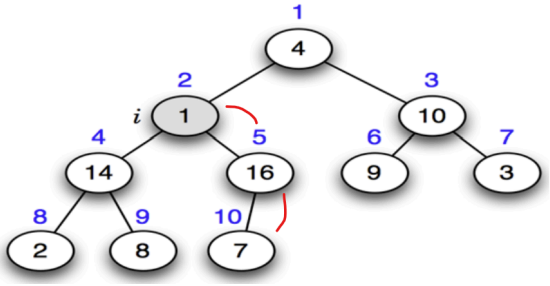
2. At index 4, the value is 2, which is smaller than both its children. The largest child, 14, is at index 8, so a swap occurs between 2 and 14.



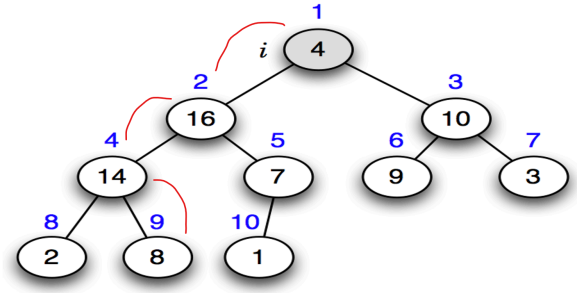
3. At index 3, the value is 3, which is smaller than both its children. The largest child, 10, is at index 7, so a swap occurs between 7 and 10.



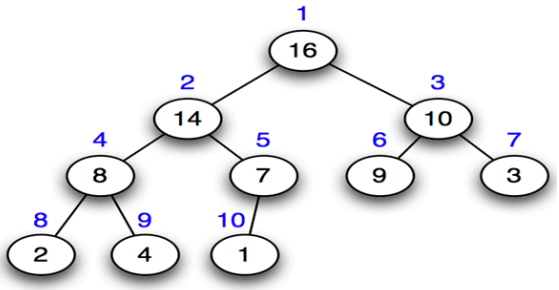
4. At index 2, the value is 1, smaller than both its children. The largest child, 16, is at index 5, so a swap occurs between 1 and 16. At index 5, the value is 1, which is smaller than its left child (7 at index 10), so a swap occurs between 1 and 7.



5. At index 1, the value is 4, smaller than both its children. The largest child, 16, is at index 2, so a swap occurs between 4 and 16. At index 2, the value is 4, smaller than both its children (14 and 7). The largest child, 14, is at index 4, so a swap occurs between 4 and 14. At index 4, the value is 4, smaller than its right child (8 at index 9), so a swap occurs between 4 and 8.



6. The final state of the tree is achieved after calling max-heapify on all internal nodes, from the last internal node to the root.



Time complexity: While max-heapify runs in O(log n) and is called up to n/2 times, the actual runtime of build-max-heap(A) is not O(n log n), but O(n). This is because most heapify operations occur on smaller subtrees near the bottom of the heap, which require less work. When analyzed more precisely using the aggregate method, the total cost sums to O(n).

6.5.2.7 Heap Sort

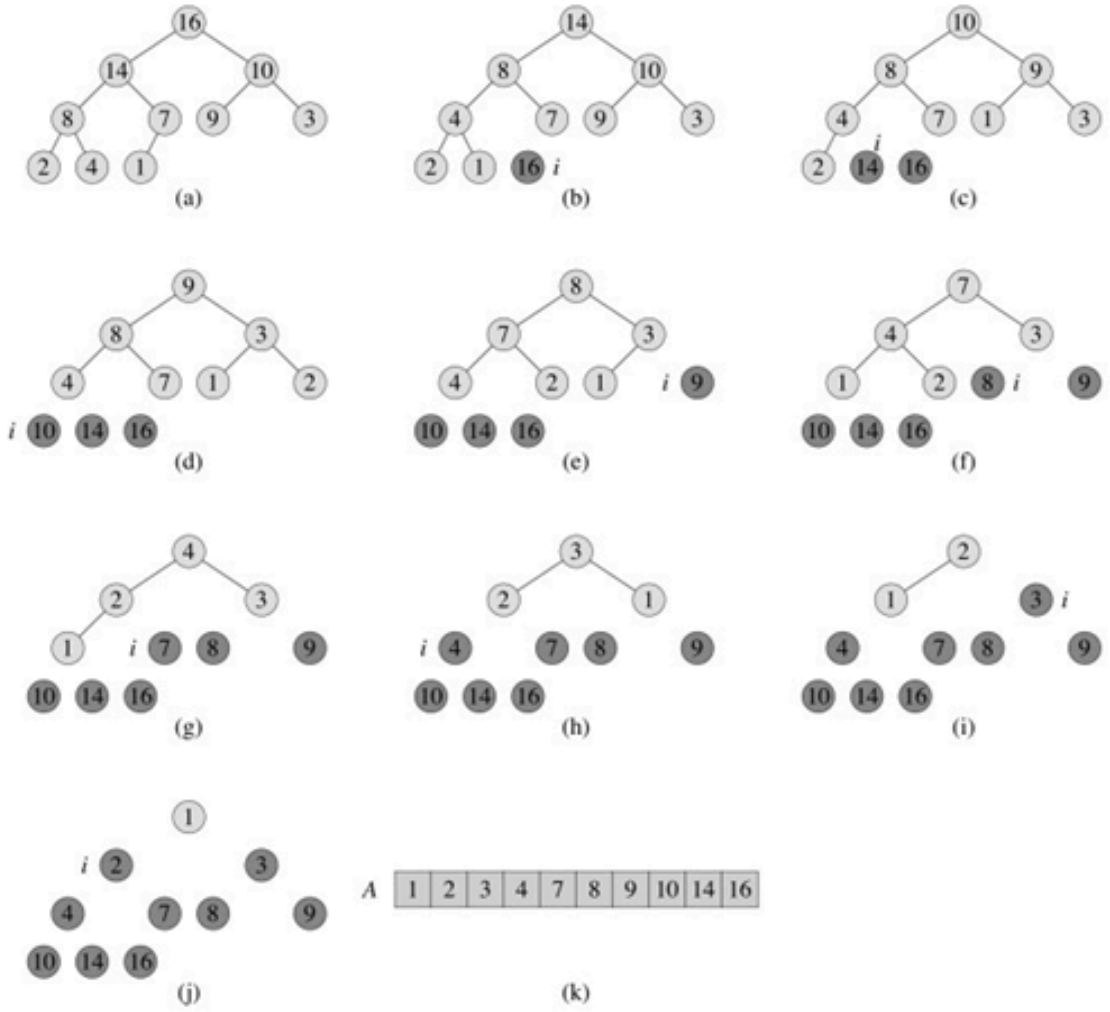
Heapsort is an in-place sorting algorithm that sorts an array A[1..n] in ascending order. It first builds a max-heap with the largest element at the root (A[1]). The root is then swapped with the last element (A[n]), and the heap size is reduced by 1. Max-heapify is called on the root to restore the heap property. This process repeats, progressively moving the largest elements to the end until the array is fully sorted.

Differences in Sorting Order
Max Heap: Sorts in ascending order by repeatedly moving the largest elements to the end.
Min Heap: Sorts in descending order by repeatedly moving the smallest elements to the end.

Heap Type	Sorting Order	Key Element Moved	Re-Heapify Operation
Max Heap	Ascending	Largest to End	Maintain Max Heap, calls max-heapify
Min Heap	Descending	Smallest to End	Maintain Min Heap, calls min-heapify

Time complexity: Heapsort operates in-place, meaning it sorts without using extra space for another array, modifying only the input array. Its worst-case time complexity is O(n log n).

Heapsort Simulation Example 01:

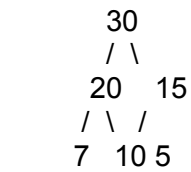


Note: Image from Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms*, 3rd edition, MIT Press, 2009.

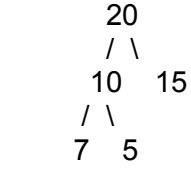
Heapsort Simulation Example 02:

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

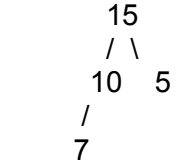
Initial Heap:



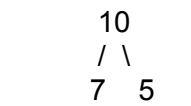
Step 1: Swap 30 and 5 → [5, 20, 15, 7, 10, 30], then heapify



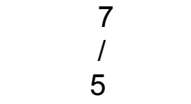
Step 2: Swap 20 and 5 → [5, 10, 15, 7, 20, 30], then heapify



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



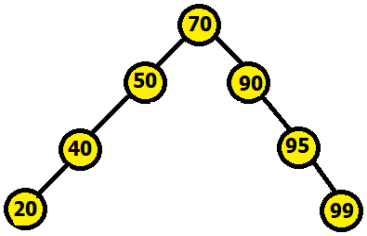
Step 5: Swap 7 and 5 → [5, 7, 10, 15, 20, 30]

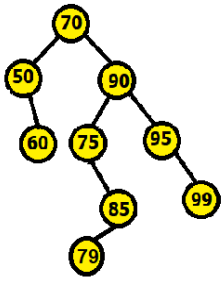
Pseudocode:

```
function heapsort(A):
    build-max-heap(A)
    for i = length[A] downto 2:
        exchange A[1] with A[i]
        heap-size[A] = heap-size[A] - 1
        max-heapify(A, 1)
```

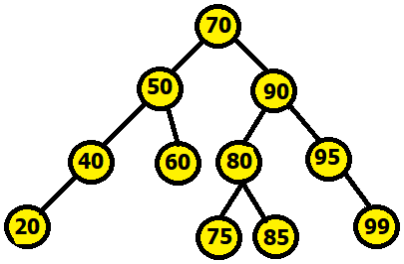
Exercises

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

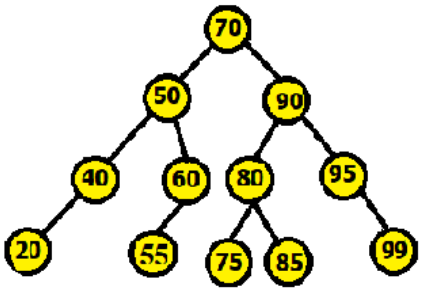




6.2: Insert keys 65, 105, 69 into the following BST and show the steps. Show simulation and code.

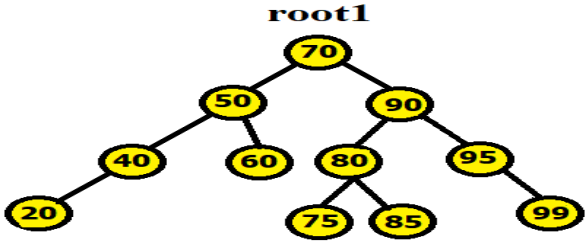


6.3: Delete keys 20, 95, 50, 70, 75 into the following BST and show the steps. Show simulation and code..



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.



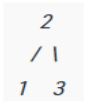
Output: In-order Successor: 75
In-order Predecessor: 60

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:

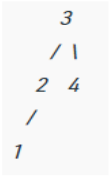
1. Set The middle element of the array as root.
2. Recursively do the same for the left half and right half.
3. Get the middle of the left half and make it the left child of the root created in step 1.
4. Get the middle of the right half and make it the right child of the root created in step 1.
5. Print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

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

BST#1



BST#2

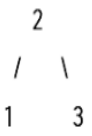


Given Array#2: [1, 2, 3,4]
Output: Pre-order of created BST: 3 2 1 4

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:



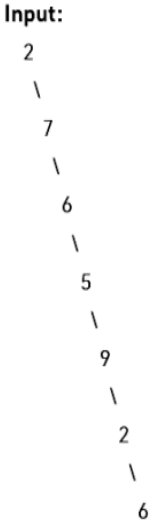
Output: 1

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.

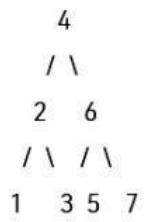
6.8: Given a sorted array. Write a function that creates a Balanced Binary Search Tree using array elements. Height balanced BST means a binary tree in which the depth of the left subtree and the right subtree of every node never differ by more than 1

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

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

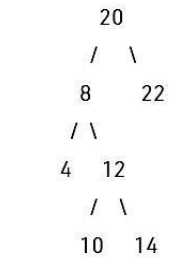
Explanation:

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

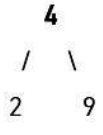
Explanation:

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