**4a**.Build step by step a min-heap for 27, 13, 56, 35, 48, 8, 18, 67, 5, 62, 7 by inserting those

elements (in order) into an initially empty heap. Show how the heap looks like (in a tree form

and in an array form) after each insertion of the first 10 elements, then show the insertion of

the last element (7) step by step in tree form and array form.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 27 |  |  |  |  |  |  |  |  |  |  |

**27**

**Insert 27**

**Insert 13**

**27**

**13**

**27**

27 is greater than 13.

**13**

Lets swap

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 27 | 13 |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 13 | 27 |  |  |  |  |  |  |  |  |  |

**13**

**27**

**Insert 56**

**56**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 13 | 27 | 56 |  |  |  |  |  |  |  |  |

**Insert 35**

**56**

**35**

**13**

**27**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 13 | 27 | 56 | 35 |  |  |  |  |  |  |  |

**35**

**13**

**27**

**56**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 13 | 27 | 56 | 35 | 48 |  |  |  |  |  |  |

**Insert 48**

**48**

**Insert 8**

13 is greater than 8.

Swapping both

**8**

**27**

**13**

**56**

**13**

56 is greater than 8.

Lets swap.

**27**

**56**

**35**

**48**

**35**

**48**

**8**

**27**

**8**

**48**

**56**

**35**

**13**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8 | 27 | 13 | 35 | 18 | 56 |  |  |  |  |  |

**Insert 18**

**18**

**27**

**8**

**48**

**56**

**35**

**13**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8 | 27 | 13 | 35 | 48 | 56 | 18 |  |  |  |  |

**Insert 67**

**18**

**35**

**56**

**48**

**8**

**27**

**13**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8 | 27 | 13 | 35 | 48 | 56 | 18 | 67 |  |  |  |

**67**

**Insert 5**

**3**

**67**

**18**

**35**

**56**

**48**

**8**

**27**

**13**

**2**

**1**

**5**

Since 35 is greater than 5, we swapped it. Yet 27 is now greater than 5, we swap it. Now, 8 is greater than 5. We swap it. This step takes 3 swaps, making 5 to be the root element.

**13**

**8**

**5**

**48**

**27**

**67**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 8 | 13 | 27 | 48 | 56 | 18 | 67 | 35 |  |  |

**18**

**56**

**35**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 8 | 13 | 27 | 48 | 56 | 18 | 67 | 35 | 62 |  |

**Insert 62**

**18**

**56**

**48**

**5**

**8**

**13**

**27**

**67**

**35**

**62**

**Insert 7**

**8**

**5**

**48**

**56**

**27**

**18**

**67**

**35**

**62**

**13**

2

1

**7**

Since 48 is greater than 7, we swap both of them. Now 8 is greater than 7, hence we swap them making the tree look like.

**48**

**62**

**35**

**67**

**18**

**27**

**56**

**8**

**5**

**7**

**13**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 7 | 13 | 27 | 8 | 56 | 18 | 67 | 35 | 62 | 48 |

Final Min heap and final array

**4.** b. Perform delete-min() on the min-heap you built in part a. Show how the heap looks like (in a tree form and array form) after each step.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 7 | 13 | 27 | 48 | 56 | 18 | 67 | 35 | 62 | 48 |

Our min heap looks like

**48**

**62**

**35**

**67**

**18**

**27**

**56**

**8**

**5**

**7**

**13**

Deleting 5 from our min heap tree and transferring the last node “48” to root.

**13**

**7**

**48**

**8**

**56**

**27**

**18**

**67**

**35**

**62**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 48 | 7 | 13 | 27 | 8 | 56 | 18 | 67 | 35 | 62 |  |

1

48 is greater than 7. So we swap them.

**13**

**48**

**7**

**8**

**56**

**27**

**18**

**67**

**35**

**62**

2

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 48 | 13 | 27 | 8 | 56 | 18 | 67 | 35 | 62 |  |

48 is now greater than 8. Now we swap them

**13**

**8**

**7**

**48**

**56**

**27**

**18**

**67**

**35**

**62**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 8 | 13 | 27 | 48 | 56 | 18 | 67 | 35 | 62 |  |

Our final heap and array looks like above figures

**4c.** Insert 27, 13, 56, 35, 48, 8, 18, 67, 5, 62, 3, 12, 14, 7, 10, 16, 17, 15 (as you read them) into an originally empty binary search tree. Show how the tree looks like after each insertion. Perform delete (13) on this tree step by step, showing how the tree looks like after each step

**Insert 27 Insert 13 Insert 56**

**27**

**27**

**27**

**56**

**13**

**13**

**Insert 35 Insert 48 Insert 8**

**27**

**27**

**13**

**56**

**35**

**48**

**56**

**27**

**13**

**13**

**56**

**35**

**8**

**35**

**48**

**Insert 18 Insert 67 Insert 5**

**27**

**27**

**67**

**18**

**8**

**48**

**35**

**56**

**13**

**18**

**8**

**35**

**56**

**13**

**27**

**56**

**13**

**67**

**18**

**35**

**8**

**48**

**48**

**5**

**Insert 62 Insert 3**

**3**

**62**

**5**

**67**

**18**

**8**

**48**

**35**

**56**

**13**

**27**

**62**

**48**

**5**

**67**

**18**

**8**

**35**

**56**

**13**

**27**

**Insert 12 Insert 14**

**27**

**13**

**56**

**48**

**18**

**62**

**27**

**56**

**35**

**48**

**8**

**18**

**67**

**5**

**62**

**3**

**13**

**67**

**35**

**8**

**12**

**5**

**14**

**12**

**3**

**Insert 7 Insert 10**

**13**

**56**

**27**

**12**

**27**

**12**

**3**

**62**

**5**

**67**

**8**

**48**

**35**

**56**

**13**

**67**

**10**

**7**

**3**

**5**

**14**

**18**

**48**

**62**

**35**

**8**

**14**

**18**

**7**

**Insert 16 Insert 17**

**27**

**27**

**27**

**17**

**16**

**10**

**7**

**14**

**12**

**3**

**62**

**5**

**67**

**18**

**8**

**48**

**35**

**56**

**13**

**16**

**10**

**7**

**14**

**12**

**3**

**62**

**5**

**67**

**18**

**8**

**48**

**35**

**56**

**13**

**Insert 15**

**16**

**10**

**7**

**14**

**12**

**3**

**62**

**5**

**67**

**18**

**8**

**48**

**35**

**56**

**13**

**15**

**17**

**17**

**17**

**Deleting 13**

14 is the minimum node in the right subtree of node 13. So we move 14 to empty node. And the children of 14 will become the children of 18 in the left subtree.

**27**

**1**

**56**

**48**

**8**

**18**

**5**

**3**

**12**

**14**

**7**

**10**

**16**

**17**

**17**

**15**

**17**

**2**

**67**

**35**

**62**

The BST after deleting the node 13

**10**

**7**

**12**

**3**

**62**

**5**

**67**

**18**

**8**

**48**

**35**

**56**

**14**

**27**

**3**

**16**

**15**

**17**

**17**

**17**

**3.a** Let T be a binary tree and assume that every node has one numerical data field (called“data”) of type double. Write a recursive algorithm that takes as input the tree T and computes the following values: (1) number of nodes in T that have positive values in the data field, (2) number of nodes in T that have negative values in the data field, and (3) number of Nodes in T that have value 0 in the data field. Note: You should give one single algorithm (not three separate algorithms) that computes all three numbers together.

Answer

// driver code start

//GetNodeDetails is the function written just to call the NodeDeterminer procedure.

//driver code end

**procedure GetNodeDetails(Tree T)**

**begin**

**nodePtr p;**

**p = T;**

**int positiveNodes = 0,negativeNodes = 0, zeroNodes = 0;**

**NodeDeterminer(T, positiveNodes,negativeNodes,zeroNodes)**

**end**

//Actual procedure starts

**procedure NodeDeterminer(Tree T, output positiveNodes, negativeNodes, zeroNodes)**

**begin**

**nodePtr p;**

**p = T;**

**while(p != null)**

**begin**

**if(p.data == 0)**

**zeroNodes++;**

**else if(p.data>= 1)**

**positiveNodes++;**

**else**

**negativeNodes++;**

**end if**

**NodeDeterminer(p.Right);**

**NodeDeterminer(p.Left);**

**end while**

**end**

The above algorithm computes the task in **O(n)** time. Because, we are using breadth first search and every node is visited exactly once.

**3b.** Write a recursive algorithm that takes as input a binary tree T and returns whether or not the

tree is full. Give the time complexity of your algorithm.

Answer

**func DetermineIfBinaryTreeIsComplete(Tree T)**

**begin**

**nodePtr p ;**

**p = T;**

**if(p==null)**

**return false;**

**Queue q = new Queue();**

**int popCounter = 0,height = 0;**

**q.Enqueue(p);**

**while(!q.IsNotEmpty())**

**begin**

**nodePtr temp = q.Dequeue();**

**int subTreeHeight =0;**

**popCounter++;**

**if(temp.Right!== null && temp.Left!==null)**

**q.Enqueue(temp.Right);**

**q.Enqueue(temp.Left);**

**else if(temp.Right == null && temp.Left == null)**

**if(height == 0)**

**height = GetHeightFromPoppedValues(popCounter);**

**subTreeHeight = height;**

**else if(height ! = subTreeHeight)**

**return false;**

**end if**

**end if**

**end while**

**if(height == GetHeightFromPoppedValues(popCounter))**

**return true;**

**end if**

**return false;**

**end**

**func GetHeightFromPoppedValues(popCounter)**

**begin**

**return 1+ max(popCounter/2, popCounter+1 /2);**

**end**

The above algorithm takes O(n) time to compute. Because we add n elements to the queue and scan one node exactly once to check if it has child nodes or not. We keep of track of number of elements popped from queue to know the height of the current node. We make note of the height of first leaf node and keep it as base to compare wit rest others. In a complete binary tree the leaf nodes are at the same level. So, we encounter any leaf node at other height, we can disregard that the tree is not a complete binary tree. If the leaf node height is consistent with all other leaf nodes and the height calculated from number of popped elements equals popcounter, we can be sure that the tree is a complete binary tree.

**2a.** Show all AVL trees of height 0 and height 1, and show 8 AVL trees of height 2.

AVL Trees Of Height 0



**A**

AVL Trees Of Height 1



**B**

**A**

**C**

**A**

**A**



**B**

**B**

AVL Trees Of Height 2

**E**

**A**

**C**

**D**

**B**

**E**

**B**

**D**

**C**

**A**

**C**

**A**

**B**

**D**

**D**

**C**

**A**

**B**



**A**

**A**

**C**

**E**

**B**

**D**

**F**

**A**

**G**

**F**

**D**

**B**

**C**



**B**

**C**



**D**



**G**

**F**

**E**



**A**



**C**

**E**

**B**

**F**



**D**



**2b**. A *left-heavy* AVL is an AVL where for every internal node, the height of its left subtree is one more than the height of its right subtree. Show all left-heavy AVL trees of height 0, height 1, height 2, and height 3.

Left Heavy AVL trees of height 0

**A**

Left Heavy AVL trees of height 1

**A**

**B**

Left Heavy AVL trees of height 2

**A**

**C**

**B**

**D**

Left Heavy AVL trees of height 3

**A**

**C**

**D**

**B**

**G**

**E**

**F**

**2 c.** Let 𝑁(ℎ) be the number of nodes of a left-heavy AVL tree of height *h*. Express 𝑁(ℎ) in

terms of 𝑁(ℎ - 1) and 𝑁(ℎ - 2).

Answer :

Given to represent N(h) in terms of N(h-1) and N(h-2). N(h) is defined as the minimum number of nodes at a given height “h” of an AVL tree.

In a left heavy AVL tree, each internal node is left heavy. That is left subtree is at most differ by 1 in height than right subtree.

Let h=0, then at a height h=0, number of nodes in a left heavy AVL tree = 1, i.e N(0) = 1, which is a root node.

For height h=1, number of nodes in a left heavy AVL tree = 2. Root node + 1 node on left side +0 nodes in right side.

N(1) = 1+1+0

For height h=2, number of nodes in a left heavy AVL tree = root node + nodes on left subtree + node in right subtree

Nodes in left subtree is N(t) where t is the height from the root node. If root node is at height h, then the height of left subtree is at height h-1 (since we exclude root node), ie. N(h-1)

Nodes in right subtree is N(t) where t is the height from the root node. If root node is at height h, then the height of right subtree is height h-2 (since we exclude root node and the height difference between it and left subtree is at most 1 and given it is a left heavy AVL tree), ie. N(h-2)

We can deduce that   
N(2) = 1 + N(2-1) + N(2-2)

This pattern follows for every h>=2 and base case is N(0) =1 and N(1) =2

Hence we can conclude that

**N(h) = 1+N(h-1)+N(h-2)**

**2d.** It can be shown that for some constants *a* and *b* (you do not have to prove this formula). Find the values of *a* and *b.*

Answer :

Given

N(h) is defined as the number of nodes at a given height “h” of an AVL tree.

We know that for N(0) = 1 and N(1) = 2

Substituting h=0 and h=1 in the given equation.

N(0) = 1 =

N(1) = 2 **=**

Solving both the equations. That is substituting a = 2-b

Substituting eq3 in eq1

Hence the values of a and b are and .

Problem 1

**1a.** Show by induction on *n* that

Answer

Base Step :

Let us check if substituting 1 satisfies the equation; for n=1

= 1(1+1)(2(1)+1)/2 => 1\*2\*3/6 => 6/6 => 1

Base case satisfied.

Inductive Hypothesis :

Let us assume that for any number m-1 where m >=1, it satisfies the equation. That is

=>

Inductive Step :

Prove the same for . That is

We know that

Substituting the above in eq 2

Hence, by the mathematical induction, we proved that the given equation holds true for all n>=1

**1b.** Let 𝑇(𝑛) be defined recursively as follows: 𝑇(1) = 𝑐, and 𝑇(𝑛) = 5𝑇 () + 𝑐 ∀ 𝑛 ≥ 5

where 𝑐 is an arbitrary constant, and 𝒏 **=**  for some non-negative integer 𝑘. Prove by induction on *k* that 𝑇(𝑛) = n -

Answer :

Base step :

Let us substitute , and

Let us check if the equation satisfies for the base case by substituting k=0

The base case is satisfied.

Inductive Hypothesis :

Let us assume that the holds true for any value k>=0. Then for some k=m the equation holds true.

Inductive Step :

Then prove the same for k = m+1 that

From the above recurrence relation,

Substituting eq 1 in eq 2.

Hence for k = m+1, the recurrence relation holds true. We can say that for ∀ 𝑛 ≥ 5, where 𝒏 **= ,** 𝑇(𝑛) = n - holds true and where 𝑇(𝑛) = 5𝑇 () + 𝑐

**1c.** Let 𝑇(𝑛) be defined recursively as follows: 𝑇(1) = 𝑐, and 𝑇(𝑛) = 2𝑇(⌊n/2⌋) + 𝑐𝑛 , for all integers 𝑛 ≥ 2, where 𝑐 is an arbitrary positive constant and ⌊𝑥⌋ for any number 𝑥 is the *floor* value of 𝑥. Prove by induction on 𝑛 that 𝑇(𝑛) ≤ 𝑐𝑛log 𝑛 + 𝑐𝑛 for all integers 𝑛 ≥ 1.

Answer :

Base Step :

Let us substitute n=1 in T(n), then

T(1) = c, hence satisfied.

Inductive Hypothesis :

Let us assume that for some n=k, the relation holds true. That is eq1 holds true for every n=k.

Inductive Step :

Then prove that for n=k+1 for some n >= 1, T(n) also holds true. That is

From the recurrence relation,

For the case where k+1 is even, then k+1 = 2q, and

Substituting eq 1 in eq2, then

For the odd case, where k+1 is odd, then k+1 wholly can be written as 2q+1(We know that if “t” is a number, then 2p is even and 2p+1 is odd). It means k=2q.

For . Substituting 2q+1 in k, then (Deduced from )

Substituting eq 1 in eq3, then

Hence, holds true for every n >=1

**1.d** Let 𝑇(𝑛) be defined recursively as follows: 𝑇(1) = 𝑐, and 𝑇(𝑛) . Use the Master Theorem to find the Θ value of 𝑇(𝑛),

Answer

Master Theorem states that for a recurrence relation of type , where a is number of subproblems and b is the size of each subproblem and f(n) is a function that is asymptotically positive for a>=1 and b>1,

Then if

Where for

In our case,

a = 4, b=2, f(n) =

clearly for some

Hence, case 3 applies for our problem.