FTP_Alg_Week 3: Exercises

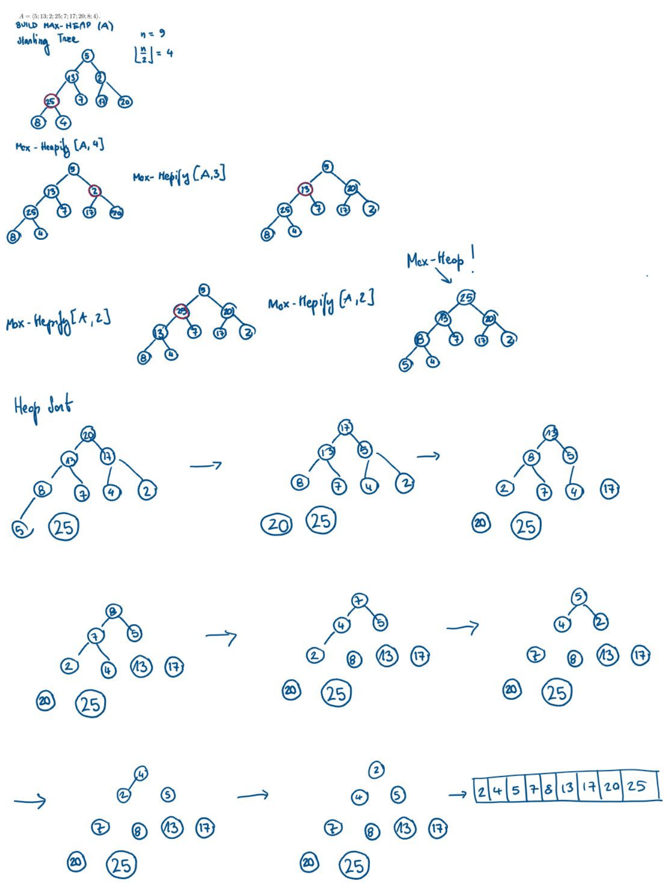
jungkyu.canci@hslu.ch

October 7, 2024

Exercise 1 *Using the figure in slide 25 of the slide of week 2 as a model, illustrate the operations of HEAPSORT on the array*

$$A = \langle 5; 13; 2; 25; 7; 17; 20; 8; 4 \rangle.$$

Solution:



Exercise 2 *Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. (Recall that every node is its own ancestor.)*

Solution: Since there is only the left subtree rooted in x, all descendants of x can not be the successor of x because their key is less then $x.key$. Thus y must be an ancestor of x. Now let us assume that the lowest ancestor of x whose left child is also an ancestor of x is not y (the successor of x) but another node z. This would mean that x is in the left subtree rooted in z, then $x.key < z.key$ as expected. But note that x has to be in the left subtree of y and because the above property verified by z, also z should be in the left subtree rooted in y, obtaining the contradiction $z.key < y.key$, since y is the successor of x.

Exercise 3 *Write the TREE-PREDECESSOR procedure.*

Solution: To obtain TREE–PREDECESSOR(x) procedure, replace in TREE–SUCCESSOR(x) “left” instead of “right” and “MAXIMUM”instead of “MINIMUM”.

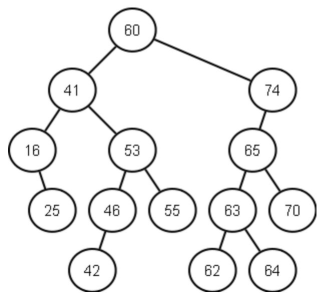
TREE-PREDECESSOR(x)

if $x.left \neq$ NIL
    return TREE-MAXIMUM($x.left$)
$y = x.p$
while $y \neq$ NIL and $x == y.left$
    $x = y$
    $y = y.p$
return y

Exercise 4 *Let T be a Binary Search Three. Prove that it always possible to insert a node z as a leaf of the three T with $z.key = r$.*

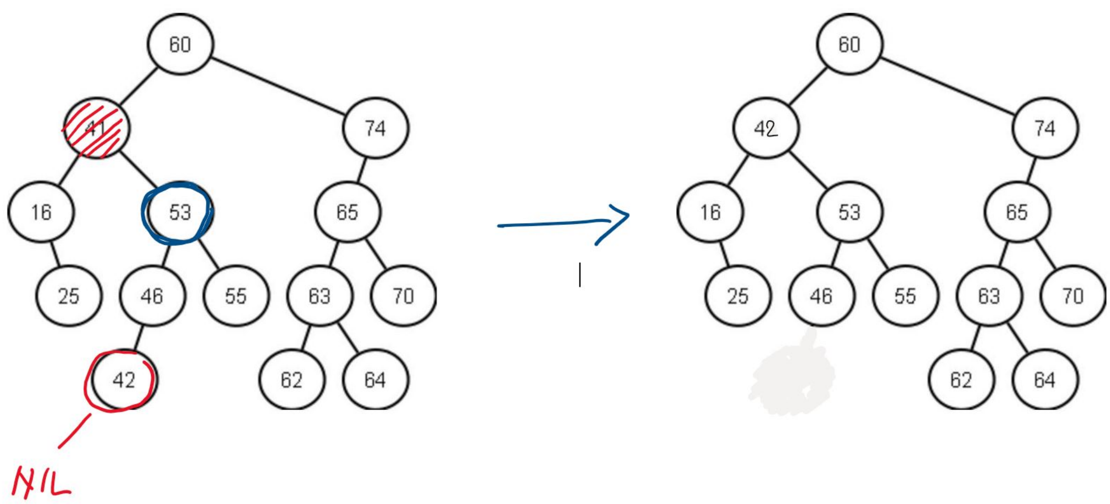
Solution. This is a straightforward property of Binary Search Three. We want to insert the leaf z with $z.key = r$ and maintain the Binary Search Property. We prove the above property by induction on the height of the tree T. If the height is zero, i.e. the tree consists only of the root that we denote by x. if $r \leq x.key$, then we put $z = lc(x)$ (left child of x) otherwise (else) we set $z = rc(x)$ (right child of x). Now we want to prove that the statement is true for a tree T of height h and we assume that the statement is true for a tree of height $h - 1$ (inductive assumption). Let us denote as before with x the root of the tree T. If $r \leq x.key$, then z must be inserted as leaf in the left sub tree with root $lc(x)$, that has height $h - 1$. Therefore by the inductive assumption we can place z as a leaf in the left sub tree. Similarly if $r \geq x.key$ but in that case z will be placed as leaf of the right sub tree.

Exercise 5 *Let T be a Binary Search Three given in the figure below*



Give the output tree after the call of TREE-DELETE(T, z) where z is the node with key 41.

Solution:



Exercise 6 (*) *What is the difference between the binary-search-tree property and the min-heap property? Can the min-heap property be used to print out the keys of an n-node tree in sorted order in $O(n)$ time? Show how, or explain why not.*

Solution: This exercise is the one with number 12.1-2 of the book "Introduction to Algorithms" by Cormen et al., whose solution is possible to find at the link given by the authors in the introduction of the book. Here I give a screen shot of their solution. Look out: the first inequality of the authors' solution is wrong, since in a min-heap a node has key $\leq$ the keys of its children.

Solution to Exercise 12.1-2

In a heap, a node's key is $\geq$ both of its children's keys. In a binary search tree, a node's key is $\geq$ its left child's key, but $\leq$ its right child's key.

The heap property, unlike the binary-searth-tree property, doesn't help print the nodes in sorted order because it doesn't tell which subtree of a node contains the element to print before that node. In a heap, the largest element smaller than the node could be in either subtree.

Note that if the heap property could be used to print the keys in sorted order in $O(n)$ time, we would have an $O(n)$-time algorithm for sorting, because building the heap takes only $O(n)$ time. But we know (Chapter 8) that a comparison sort must take $\Omega(n \lg n)$ time.