COMP 7276: Assignment #1

Due on Friday, February 15, 2013

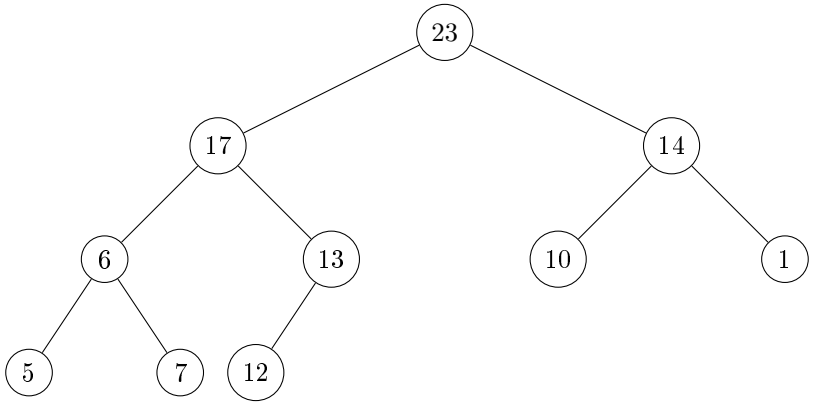
Narayanan 8:00am

Jonathan Hood

Contents

Problem 6.1-6

Is the array with values <23, 17, 14, 6, 13, 10, 1, 5, 7, 12> a max heap? (Answer yes or no then explain your answer in at most 5 sentences. No credit without explanation.)



No, this is not a max heap. Node 6 has a child greater than itself (7), violating the definition of a max heap.

Problem 6.2-2

Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MIN-HEAPIFY(A, i), which performs the corresponding manipulation on a min heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY? (Make only the minimum necessary changes to the MAX-HEAPIFY procedure on p.154 to obtain MIN-HEAPIFY. You do not need to answer the question about running time.)

Listing 1: MIN-HEAPIFY(A, i)

```
l = LEFT(i)
r = RIGHT(i)
if l <= A.heap-size and A[l] < A[i]
     smallest = l
else smallest = i
if r <= A.heap-size and A[r] < A[smallest]
     smallest = r
if smallest != i
     exchange A[i] with A[smallest]
     MIN-HEAPIFY(A, smallest)
```

Changes: change variable "largest" "smallest" for clarity (despite not being a "minimum necessary change"), and check for smaller child node.

Problem 6.3-2

Why do we want the loop index i in line 2 of BUILD-MAX-HEAP to decrease from [A.length/2] to 1 rather than increase from 1 to [A.length/2]? (Provide a clear explanation in fewer than 5 sentences.)

Problem 6.3-2 continued on next page...

The two keys here are efficiency and simplicity; the MAX-HEAPIFY function checks the local root node i and moves it downwards (if it is not already the maximum value). Then, it recursively calls itself again on the subheap where i was moved so that the i node can percolate down to the bottom. Going from [A.length/2] downto 1, BUILD-MAX-HEAP goes from the bottom up, allowing larger nodes to reach the top of the heap. If the algorithm went from 1 to [A.length/2], large values at the bottom of the heap tree would be checked at the end of the iteration, having an opportunity to only move up a single layer in the tree (and not percolate all the way to the top).

Problem 6.4-1

Using figure 6.4 as a model, illustrate the operation of HEAPSORT on the array A = <5, 13, 2, 25, 7, 17, 20, 8, 4>. (Instead of the trees as in Figure 6.4, show only the array after each call of MAX-HEAPIFY.)
Assumption: professor wants the state of the array before the recursive call back to MAX-HEAPIFY on line 10 of MAX-HEAPIFY (IE: after the swap with the greatest child but before running MAX-HEAPIFY on the swapped child node position). Call from BUILD-MAX-HEAP is on line 3. Recursive call from MAX-HEAPIFY is on line 10. Call from HEAPSORT is on line 5. step. A = <array> (next time MAX-HEAPIFY is called); 0 = initial state (first call)

0. A = <5, 13, 2, 25, 7, 17, 20, 8, 4> (next call from BUILD-MAX-HEAP; i = [9/2] = 4)
1. A = <5, 13, 2, 25, 7, 17, 20, 8, 4> (next call from BUILD-MAX-HEAP; i = 3)
2. A = <5, 13, 20, 25, 7, 17, 2, 8, 4> (next call recursively; largest = 7)
3. A = <5, 13, 20, 25, 7, 17, 2, 8, 4> (next call from BUILD-MAX-HEAP; i = 2)
4. A = <5, 25, 20, 13, 7, 17, 2, 8, 4> (next call recursively; largest = 4)
5. A = <5, 25, 20, 13, 7, 17, 2, 8, 4> (next call from BUILD-MAX-HEAP; i = 1)
6. A = <25, 5, 20, 13, 7, 17, 2, 8, 4> (next call recursively; largest = 2)
7. A = <25, 13, 20, 5, 7, 17, 2, 8, 4> (next call recursively; largest = 4)
8. A = <25, 13, 20, 8, 7, 17, 2, 5, 4> (next call recursively; largest = 8)
9. A = <25, 13, 20, 8, 7, 17, 2, 5, 4> (next from HEAPSORT; heap_size = 8; A = <4, 13, 20, 8, 7, 17, 2, 5, 25>)
10. A = <20, 13, 4, 8, 7, 17, 2, 5, 25> (next call recursively; largest = 3)
11. A = <20, 13, 17, 8, 7, 4, 2, 5, 25> (next call recursively; largest = 6)
12. A = <20, 13, 17, 8, 7, 4, 2, 5, 25> (next from HEAPSORT; heap_size = 7; A = <5, 13, 17, 8, 7, 4, 2, 20, 25>)
13. A = <17, 13, 5, 8, 7, 4, 2, 20, 25> (next call recursively; largest = 3)
14. A = <17, 13, 5, 8, 7, 4, 2, 20, 25> (next from HEAPSORT; heap_size = 6; A = <2, 13, 5, 8, 7, 4, 17, 20, 25>)
15. A = <13, 2, 5, 8, 7, 4, 17, 20, 25> (next call recursively; largest = 2)
16. A = <13, 8, 5, 2, 7, 4, 17, 20, 25> (next call recursively; largest = 4)

Problem 6.4-1 continued on next page...

17. A = <13, 8, 5, 2, 7, 4, 17, 20, 25> (next from HEAPSORT; heap_size = 5; A = <4, 8, 5, 2, 7, 13, 17, 20, 25>)
18. A = <8, 4, 5, 2, 7, 13, 17, 20, 25> (next call recursively; largest = 2)
19. A = <8, 7, 5, 2, 4, 13, 17, 20, 25> (next call recursively; largest = 5)
20. A = <8, 7, 5, 2, 4, 13, 17, 20, 25> (next from HEAPSORT; heap_size = 4; A = <4, 7, 5, 2, 8, 13, 17, 20, 25>)
21. A = <7, 4, 5, 2, 8, 13, 17, 20, 25> (next call recursively; largest = 2)
22. A = <7, 4, 5, 2, 8, 13, 17, 20, 25> (next from HEAPSORT; heap_size = 3; A = <2, 4, 5, 7, 8, 13, 17, 20, 25>)
23. A = <5, 4, 2, 7, 8, 13, 17, 20, 25> (next call recursively; largest = 3)
24. A = <5, 4, 2, 7, 8, 13, 17, 20, 25> (next from HEAPSORT; heap_size = 2; A = <2, 4, 5, 7, 8, 13, 17, 20, 25>)
25. A = <4, 2, 5, 7, 8, 13, 17, 20, 25> (next call recursively; largest = 2)
26. A = <4, 2, 5, 7, 8, 13, 17, 20, 25> (HEAPSORT finishes w/ swap; A = <2, 4, 5, 7, 8, 13, 17, 20, 25>)

Problem 6.5-4

Why do we bother setting the key of the inserted node to -∞ in line 2 of MAX-HEAP-INSERT when the next thing we do is increase its key to the desired value? (Provide a clear explanation in less than 5 sentences.)

MAX-HEAP-INSERT inserts a value at the end of the heap by calling HEAP-INCREASE-KEY on the newly inserted last element. Setting the value to -∞ allows the call to HEAP-INCREASE-KEY to pass its sanity check (if key < A[i]), since it is assumed that key must be a value greater than -∞.

Problem 7.1-2

What value of q does PARTITION return when all elements in the array $A[p..r]$ have the same value? Modify PARTITION so that q = $\lfloor (p+r)/2 \rfloor$ when all elements in the array $A[p..r]$ have the same value. (Answer only the first question. You do not need to modify PARTITION.)

r-1

Problem 7.3

(Explain in fewer than 5 sentences why randomizing Quicksort is useful.)

When the general quicksort algorithm is run on a sorted or partially sorted array, the pivot does not often equally divide up the partition. This causes the quicksort algorithm to have poor performance in such a scenario! Instead, randomizing which value is chosen for the pivot reduces the average runtime of the algorithm by assuring a greater probability that the partitions will not be as lopsided.

Problem 10.1-1

Using figure 10.1 as a model, illustrate the result of each operation in the sequence $PUSH(S, 4)$, $PUSH(S, 1)$, $PUSH(S, 3)$, $POP(S)$, $PUSH(S, 8)$, and $POP(S)$ on an initially empty stack S stored in array $S[1..6]$.

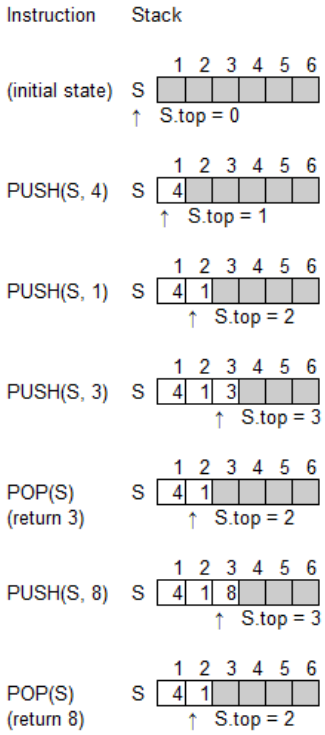


Image done vertically in excel.

Problem 10.2-4

As written, each loop iteration in LIST-SEARCH' procedure requires two tests: one for $x \neq L.nil$ and one for $x.key \neq k$. Show how to eliminate the test for $x \neq L.nil$ in each iteration.

Listing 2: LIST-SEARCH'(L, k)

```
x = L.nil.next
tempVal = L.nil.key
L.nil.key = k
while x.key != k
    x = x.next
L.nil.key = tempVal
return x
```

Problem 10.3-1

Draw a picture of the sequence $< 13, 4, 8, 19, 5, 11 >$ stored as a doubly linked list using the multiple-array representation. Do the same for the single-array representation.

Problem 10.3-1 continued on next page...

index	1	2	3	4	5	6	
next	2	3	4	5	6	/	
key	13	4	8	19	5	11	L=1
prev	/	1	2	3	4	5	

Multi-Array Representation is shown above.

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A	13	4	/	4	7	1	8	10	4	19	13	7	5	16	10	11	/	13
L=1																		

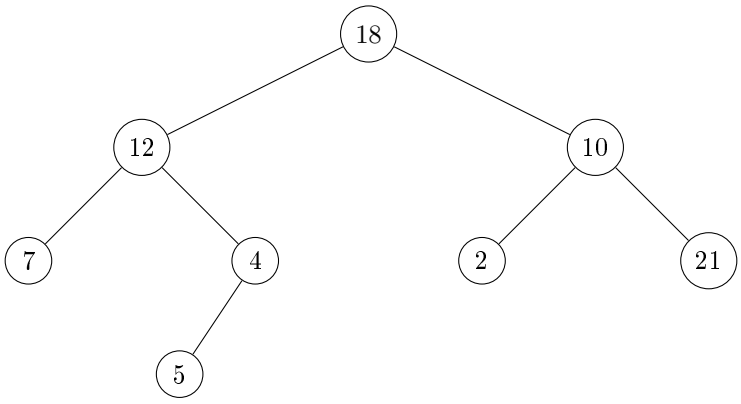
Single-Array Representation is shown above. Order of elements in each 3-element block: *key*, *next*, *prev*.

Problem 10.4-1

Draw the binary tree rooted at index 6 that is represented by the following attributes:

index	key	left	right
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

Table 1: 10.4-1 table



Binary tree representation of table beginning at node 18 (index 6)

Problem 12.1-2

What is the difference between the binary-search-tree property and the min-heap property (see page 153)? 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. (Answer only the first question.)

In a binary search tree, a top-to-bottom traversal of the tree guarantees that all nodes in the tree formed by the left child contain keys that are always less than the parent node, and the tree formed by the right child contain keys that are always greater than the parent node. In a min-heap, both child trees will always contain keys greater than the parent node.

Problem 12.2-1

Suppose that we have numbers between 1 and 1000 in a bst, and we want to search for the number 363. Which of the following sequences could not be the sequence of nodes examined? (Explain your answer in at most 5 sentences. No credit without explanation.)

a 2, 252, 401, 398, 330, 344, 397, 363.

b 924, 220, 911, 244, 898, 258, 362, 363.

c 925, 202, 911, 240, 912, 245, 363.

d 2, 399, 387, 219, 266, 382, 381, 278, 363.

e 935, 278, 347, 621, 299, 392, 358, 363.

C is impossible. After node value 911 is checked, the BST is traversed to the left of 911, indicating that no subvalues can ever be larger than 911; however, a node with value 912 exists in the tree formed by 911's left child. This violates the definition of a BST.
E is impossible. When node value 347 is reached, the tree traversal goes to the right, indicating that all values in the subtree will be greater than 347; however, a node of value 299 exists in node value 347's right child subtree, violating the definition of a BST.

Problem 12.3-2

Suppose that we construct a binary search tree by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is one plus the number of nodes examined when the value was first inserted into the tree. (Argue in at most 10 sentences.)

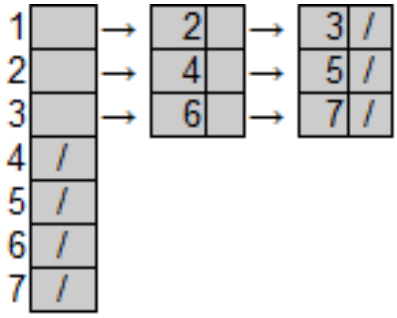
Problem 12.3-2 continued on next page...

Assumption 1: The tree does not have to be balanced (IE: no rebalancing was performed after any insertion tasks; only TREE-INSERT was ever performed on the tree T).
Assumption 2: The value being searched for exists in the tree T.
By observation, lines 3-7 of the TREE-INSERT algorithm are identical to lines 1-4 of the ITERATIVE-TREE-SEARCH algorithm. The difference is that ITERATIVE-TREE-SEARCH is configured to return a key that must exist in the tree; TREE-INSERT traverses to the NIL node that the inserting element should reside at. Let an element being inserted by TREE-INSERT be inserted to the fourth level of the BST. That means, the TREE-INSERT algorithm compared the first three levels and reached a NIL node at the fourth level (for a total of 3 node comparisons and 1 nil comparison). The ITERATIVE-TREE-SEARCH algorithm will traverse those same three nodes until it reaches the fourth level, where it will compare the key to the correct node (for a total of 4 node comparisons). Since the spanning sections of each of these algorithms are identical, they will always take the same path if the tree has not been balanced/restructured.

Problem 22.1-2

Give an adjacency-list representation for a completed binary tree on 7 vertices. Give an equivalent adjacency-matrix representation. Assume that vertices are numbered from 1 to 7 as a binary heap. (Draw the list and matrix representations using Fig. 22.1 as a model.)

Assumption: the tree does not contain backlinks to parent nodes.



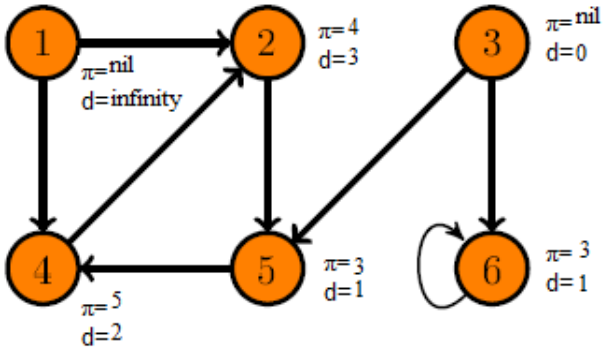
The image above is the adjacency-list representation.

	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	0	1	1
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0

The image above is the adjacency-matrix representation.

Problem 22.2-1

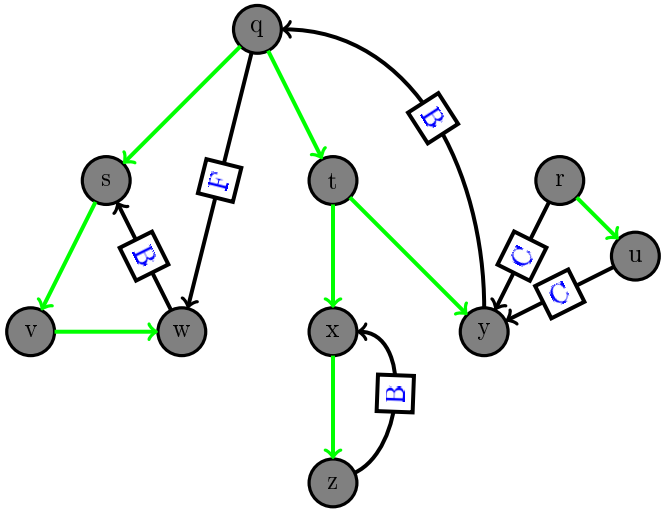
Show the d and π values that result from running breadth-first search on the directed graph of Figure 22.2(a), using vertex 3 as the source. (Draw the graph and clearly mark the d and π values by each node.)



Answer shown above.

Problem 22.3-2

Show how depth-first search works on the graph of Figure 22.6. Assume that the for loop of lines 5-7 of the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically. Show the discovery and finishing times of each vertex, and show the classification of each edge. (Draw the graph and clearly mark the d and f values and edge classification labels.)



Green arrows identify DFS tree Assumption: problem asks for something like the final graph of Figure 22.4. Nodes are shown individually below with their sort orders.
Assumption2: Since clarification from Dr. Narayanan's comment doesn't care about when the nodes are blackened, only the time steps are shown.
Assumption3: Each node's visitation schedule from DFS can be shown individually below.

Visit values (d & f): 1, 16

Visit values (d & f): 17, 20

s Visit values (d & f): 2, 7
t Visit values (d & f): 8, 15
u Visit values (d & f): 18, 19
v Visit values (d & f): 3, 6
w Visit values (d & f): 4, 5
x Visit values (d & f): 9, 12
y Visit values (d & f): 13, 14
z Visit values (d & f): 10, 11

Problem 22.4-1

Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the dag of Figure 22.8, under the assumption of Exercise 22.3-2.

Glad this question just asks for the order!

t q u z w x v y r m s o n p