Quiz/Homework Assignment #3
CSE326
Homework due date:
Wednesday, January 26th, beginning of class

February 1, 2000

This quiz is homework assignment #3. Any question you answered correctly in class were considered correct for the purposes of the assignment. Any question you did not answer or which you answered incorrectly was a short answer question for the homework assignment due in class on Wednesday.

In *italics* are instructions on how to answer each question for homework. The correct letter answer and an explanation of that answer is listed after each question.

Good luck!

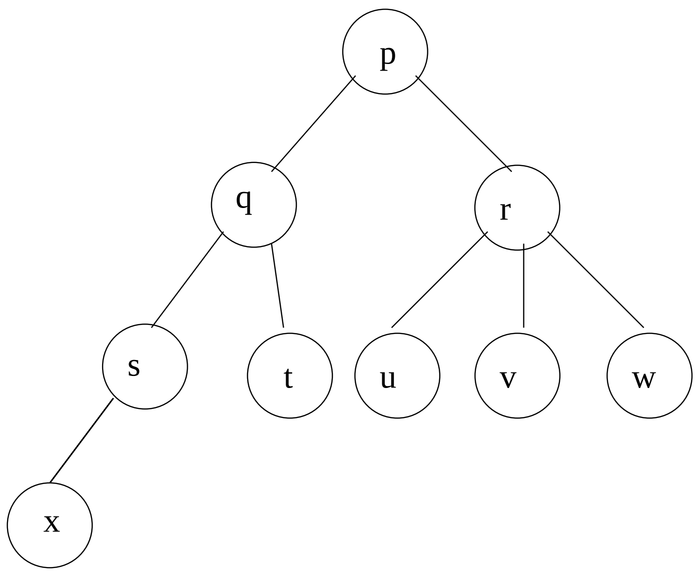


Figure 1: The tree from quiz #2. Use this tree to answer questions 1-2. Note that the priorities of the nodes are not necessarily equal to their letter labels!

1. *Explain why one of* **u**, **v**, *or* **w** *must be removed. Explain why* **x** *need not be removed. Describe the heap-order property, and explain why it may not be obeyed in this tree.*

   Which of the following must be done to ensure that this tree is a binary heap?

   (a) Remove **x**
   (b) Remove one of **u**, **v**, or **w**
   (c) Ensure that the heap-order property is obeyed
   (d) Both (b) and (c)

(e) All of the above

Correct answer(s): **(d)**

This question has *four* parts: (i) Explain why one of **u**, **v**, or **w** must be removed; (ii) Explain why **x** need not be removed; (iii) Describe the heap-order property; and (iv) Explain why it may not be obeyed in this tree.

(i) In a *binary* heap, each node may have at most *two* children. Therefore, one of **u**, **v**, or **w** must be removed so that **r** has only two children.

(ii) **x** can remain because it is as far up and left in the tree as it can go. More precisely, the first three levels of the tree are complete, and **x** is at the leftmost position in the fourth level, so the heap is indeed a complete tree.

(iii) The heap-order property states that each node in a heap must have a priority value lesser than or equal to that of its children.

(iv) The heap-order is not necessarily obeyed in this tree because, as the caption states, the priorities of the nodes are not necessarily equal to their letter labels.

2. *Explain what happens on a call to deleteMin and why the answer follows from this process. You* may *assume that all nodes have unique priorities.*

Assume that all necessary changes have been made to ensure this tree is a heap. After one call to deleteMin, which node will be at the root?

(a) **p**

(b) **q**

(c) the lesser of **q** and **r**

(d) **r**

(e) **x**

Correct answer(s): **(c)**

On a call to deleteMin, the root of the tree is removed (and returned). This leaves a hole at the top of the tree which is percolated downward until the last value in the heap (**x** in this case) can legally fill it. Percolating down works as follows: if the hole filling value is lower priority than the children of the hole (or if there are no children), stop; otherwise, fix the heap-order property at this level by moving up the lower-valued child to fill the hole (and moving the hole into the vacated space). Repeat. By the heap-order property (and the assumption that no two nodes have the same priority), **x** has a greater priority than its strict ancestors, including **q**. Therefore, the hole can't stop at the root and must percolate down at least one level. The lesser of **q** and **r** will fill that hole according to the algorithm.

3. *Explain why heap based priority queues* do *need to be able to compare the priority of data items. Explain why they do* not *need to know actual numerical values describing the priorities.*

Stacks and Queues require almost no knowledge about the data they store other than its size. What knowledge (other than size) do heap based priority queues require?

(a) None.

(b) They need to be able to compare the priority of the data items.

(c) They need to be able generate numerical values describing the priority of the data items.

(d) They need to know the exact structure of the data items.

(e) They need coffee, and lots of it.

Correct answer(s): **(b)**

In order to ensure that the heap-order property is obeyed, the heap must be able to compare a parent's priority against its childrens' priorities. However, all accesses to the priorities in the algorithm are performed with compares, so an exact value for any given node is unnecessary.

4. *Analyze the running time of the creation of the heap. Analyze the running time of the printout of the numbers. Give an overall runtime. You may assume that $O(n \log n)$ is a tight upper bound on $\log n!$. If your answer on the quiz was a loose bound, also explain why it is loose.*

Here's an algorithm to sort n numbers with a heap. Start with an empty heap and insert each number into the heap sequentially. Then, remove a number from the heap and output it, repeating until the heap is empty again. How long does this take? (Tight upper bound, please.)

(a) $O(\log^2 n)$
(b) $O(n^2)$
(c) $O(n)$
(d) $O(n \log n)$
(e) $O(n^2 \log n)$

Correct answer(s): **(d)**

(Note that this method of creating a heap—one insert at a time—is *not* the same as the `BuildHeap` method described in the text, which runs in $\Theta(n)$ time.)

Each insertion into a heap takes $O(\log n)$ time as shown in class (where n is the size of the heap). Therefore, the whole insertion process takes

$$\begin{aligned} \sum_{i=1}^{n} O(\log i) &=_d \sum_{i=1}^{n} \log i \\ &= \log\left(\prod_{i=1}^{n} i\right) \\ &= \log n! \\ &=_d O(n \log n) \qquad \text{as given above} \end{aligned}$$

So, the whole insertion takes $O(n \log n)$ time. By a similar argument, the deletion process takes $O(n \log n)$, and the sum of the two is $O(n \log n)$. Each of the looser answers is listed as strictly asymptotically greater on the Silicon Downs cheat sheet.

The following is a proof that $\log n! \in \Theta(n \log n)$. $n! < n^n$, so $\log n! < \log n^n = n \log n$. Thus, $O(n \log n)$ is an upper bound for $\log n!$. Similarly, $n! > \frac{n}{2}^{\frac{n}{2}}$, so $\log n! > \log \frac{n}{2}^{\frac{n}{2}} = \frac{n}{2} \log \frac{n}{2} =_d n \log n$. Therefore $\Omega(n \log n)$ is also a lower bound for $\log n!$, and so we must have $\log n! \in \Theta(n \log n)$.

5. *Explain why each of (a), (b), (c), and (d) may be concerns in choosing d in a d-heap.*

Which of the following is **not** a concern in choosing d in a d-heap?

(a) the size of cache lines
(b) making d a power of 2

(c) the size of elements in the heap
(d) the relative number of inserts and deleteMins
(e) the smallest key value in the heap

Correct answer(s): **(e)**

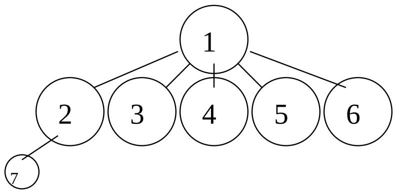
(a) The size of a cache line divided by the size of elements in the heap will be the number of elements which fit in a cache line. If we make d an integral multiple of that, we will not waste any fetches into the cache when we retrieve a set of children.
(b) Multiplying and dividing by a power of 2 is an efficient operation on a (binary) computer; therefore, making d a power of 2 makes finding a child or parent much faster.
(c) The size of elements in the heap is important for the same reason that the size of cache lines is important (see (a)).
(d) Larger d values make inserts take less time but deleteMins take more. Therefore, the relative number of inserts and deleteMins helps decide how much to optimize inserts by increasing d.

6. *Draw an example of the smallest and of the largest possible 5-heaps, and note their size. Draw the arrays that would contain those heaps.*

What is the smallest and largest number of entries in an array that a 5-heap of depth 2 can take (**not** including the element for size)?

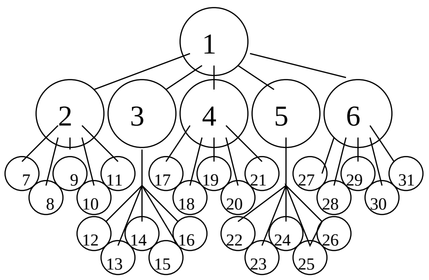
(a) 7 and 31
(b) 6 and 30
(c) 25 and 125
(d) 5 and 25
(e) None of these; a 5-heap cannot be stored in an array

Correct answer(s): **(a)**



7	1	2	3	4	5	6	7

It is clear from the diagrams above and below that the smallest 5-heap of depth 2 has seven elements, and the largest 5-heap of depth 2 has thirty-one elements.



31	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

7. *Describe how deleteMin and insert are actually implemented in leftist and skew heaps. Explain why these are efficient.*

   Which of the following is true about insert, deleteMin, and merge in leftist and skew heaps?

   (a) insert is a merge
   (b) merge is a sequence of inserts
   (c) merge is a sequence of deleteMins
   (d) deleteMin is a sequence of merges
   (e) those heaps don't implement merge

   Correct answer(s): **(a)**

   - In both leftist and skew heaps, deleteMin is implemented by removing the root and merging the left and right subtrees.
   - Insert is implemented by merging the existing heap with the single node to be inserted.
   - These operations are efficient because they both take only constant time other than their single call to merge, and merge is efficient.

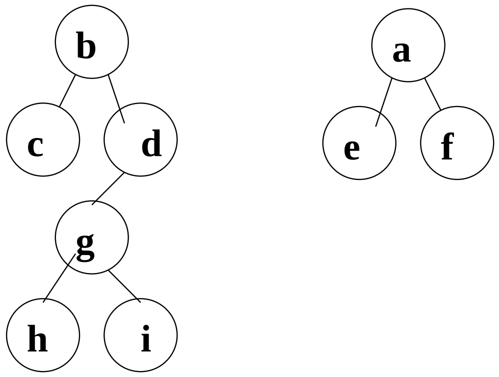


Figure 2: Two binary trees. Use this tree to answer questions 8-10. This time, assume that the priorities of the nodes **are** equal to their letter labels.

8. *Give the null path length of each node in each tree. Explain why these are therefore leftist heaps.*

   In order to make these into leftist heaps, what needs to be done?

   (a) Swap **b**'s right and left subtrees
   (b) Put **f** under **e**
   (c) Put **g** under **c**
   (d) (b) and (c)
   (e) Nothing; these are leftist heaps.

   Correct answer(s): **(e)**

   a, **b**, and **g** each have a null path length of 1; all other nodes have a null path length of 0. Hence, for each node in either tree, the NPL of its left child is greater than or equal to the NPL of its right child (with the convention that NPL($\emptyset$) = -1), so these are already leftist heaps. Oh, and also it is clear by examination that the heap-order property is obeyed.

9. *Draw out each step of the merge process in the same manner that we did in class (use recursive not iterative merge).*

If we merge these as leftist heaps, which of the following is true of the resulting tree?

(a) Every node has a null path length of 1.

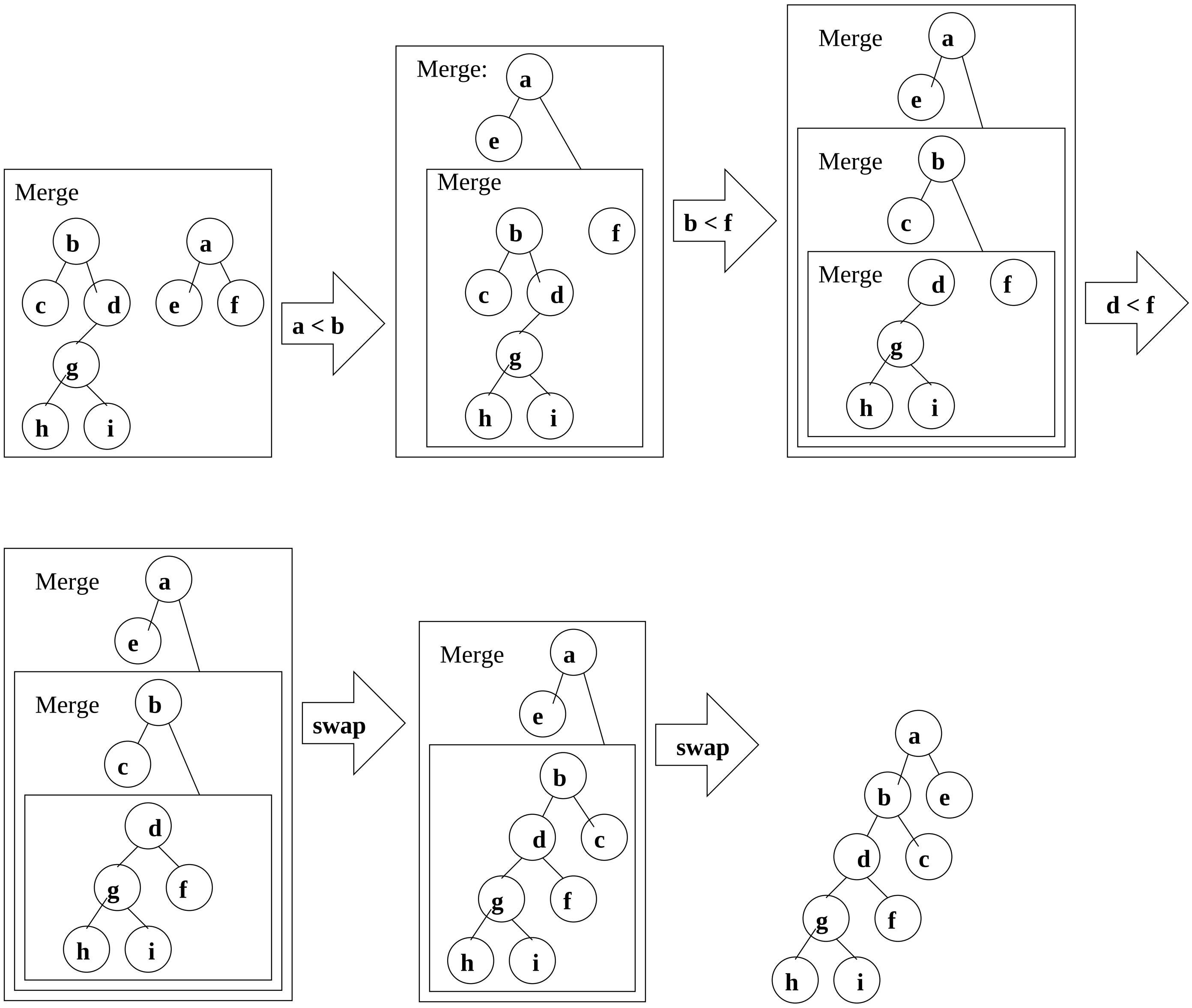
(b) Every node has zero or two children.

(c) The left path is: **a** ⇒ **b** ⇒ **d** ⇒ **f**.

(d) Every node has a null path length of 0.

(e) None of these.

Correct answer(s): **(b)**



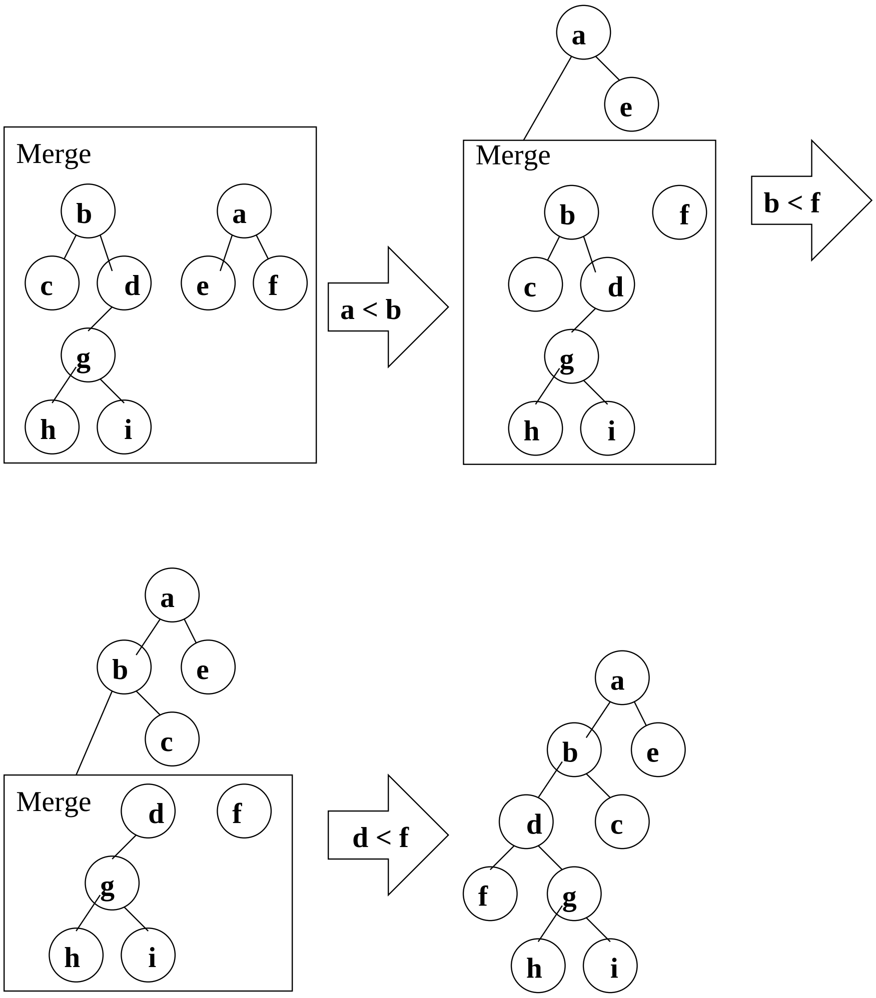
It is clear in the diagram above that every node in the resulting leftist heap has zero or two children.

10. *Draw out each step of the merge process in the same manner that we did in class (use recursive not iterative merge).*

If we merge these as skew heaps, which of the following is true of the resulting tree?

(a) Every node has a null path length of 1.
(b) It is a leftist tree.
(c) The left path is: **a** ⇒ **b** ⇒ **d** ⇒ **f**.
(d) One node has a null path length of 2.
(e) None of these.

Correct answer(s): **(c)**



It is clear in the diagram above that the left path in the resulting skew heap is: **a** ⇒ **b** ⇒ **d** ⇒ **f**.

11. *Note the clarification in this question (on part (a)). Explain why (c) and (d) are not conditions which favor skew heaps over d-heaps. Explain why (a), (b), and (e) together constitute a good situation to use skew heaps over d-heaps.*

    Which of the following together might constitute a good situation to use skew heaps over d-heaps? (Bubble in all that apply!)

    (a) Using minimal memory at any given point in time without sacrificing asymptotic speed performance is vital.

    (b) The size of the heap fluctuates wildly.

    (c) The size of the heap is stable and known.

    (d) Speed of merges is not essential.

    (e) Speed of each individual deleteMin is not essential.

    Correct answer(s): **(a, b, e)**

    (c) If the size of the heap is stable and known then we can easily declare an array just large enough to hold all the data as part of a d-heap. This will use less memory and fewer disk accesses than the pointer-based structure of a skew heap, so a d-heap would likely be more efficient in this case.

    (d) Quick merging is the *only* reason to use skew or leftist heaps. If this is not important then we're better off using d-heaps (either array-based or pointer-based, depending on the situation).

    (a) Asymptotically, skew heaps are just as good as d-heaps (in amortized time) for every operation other than merge, which they are better at. The pointer-based structure of skew heaps allows for memory usage proportional to the number of items in the heap, whereas an array-based d-heap may require extra memory, especially if d is very large or if the size of the heap fluctuates wildly. However, in a case such as (c), we're probably still better off using an array-based d-heap, and unless we care about the speed of merges we could always use a pointer-based d-heap.

    (b) If the size of the heap fluctuates wildly then we're probably better off using a skew heap (or a pointer-based d-heap); to use an array would require that we either make a gross overestimate of the size we'll need (which wastes memory), or that we dynamically resize the array as needed, which is slow (resizing takes linear time, compared to the logarithmic time of most heap operations).

    (e) The worst-case time for deleteMin in a skew heap is $O(n)$, but the amortized time is $O(\log n)$, so if the speed of each individual deleteMin is not essential, then skew heaps could be a valid choice.

    (a), (b), and (e) individually do not necessarily favor skew heaps over d-heaps, but the three of them together with the need for efficient merging certainly favors skew (or leftist) heaps over d-heaps. (Although if the speed of merging is not essential then we might still be better off with a pointer-based d-heap, depending of the specifics of our situation.)