Solutions -- Topic 9: Heaps

---

1. (1 point) Heap-Delete(A, i)

Procedure **Heap-Delete(*A*, *i*)** deletes the item at index *i* in heap *A* (represented as an array). **Give an implementation of Heap-Delete that runs in O(lg *n*) time** for a heap of size *n* = A.heapSize.

- Do not change the key of the item being deleted (in case it is an object referenced by another application).
- You may use instance variable A.heapSize.
- You do not need to include error checking.
- You do not need to return anything.
- *Hint: What other heap procedures do something similar? Use their code*

Comment: It is similar to Heap-Extract-Max, except that we are extracting max from a specified subtree. But since the replacement element may have come from a different subtree and therefore be larger, like Heap-Increase-Key we may need to propagate the replaced element up.

1 point. Half credit if there was some right idea but a logic error.

```
Heap-Delete(A,i)
// This part of solution comes from Heap-Extract-Max, and
// extracts the max from the subtree rooted at A[i]
1 A[i] = A[A.heapSize]
2 A.heapSize = A.heapSize - 1
3 Max-Heapify(A, i)
// Remainder of solution comes from Heap-Increase-Key, and
// ensures the new root of the subtree, which may have come
// from a different part of the heap, is smaller than its
// parents.
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchange A[i] with A[PARENT(i)]
6      i = PARENT(i)
```

Comment: If we allowed you to change the key of the item being deleted, an

alternative solution follows. Advantage is the use of existing code and error checking, at small cost of more instructions executed. Disadvantage in an object oriented setting where heap entries are objects with accessor methods for keys is that the client application may not expect the key of the object to change.

```
Heap-Delete(A,i)
1 Heap-Increase-Key(A, i, ∞) // that is, infinity
2 Heap-Extract-Max(A)
```

2. (2 points) Heapsort on Sorted Data

Previously we noted that Insertion-Sort runs faster on already sorted data, but Merge-Sort does not. What about Heapsort? **How is it affected by sorted data? Give your reasoning: justification is more important than getting the answer right**. Refer to line numbers in code (appended) when discussing your analyses. *Hints: Consider Build-Max-Heap and the for loop separately. You may want to work examples, but don't get bogged down in details: return to asymptotic reasoning as soon as you see what is going on.*

For each: Give 0.5 points for saying O(n lg n) and 0.5 points for justification. For full credit, each part should discuss the effect of the increasing or decreasing order (only 0.25 points for justifications that do not do this).

(a) What is the **asymptotic running time of Heapsort** using a Max-Heap on an array *A* of *n* elements that is already sorted in ***increasing*** order? Justify your claim.

> **Solution:** O(n lg n). Build-Max-Heap is O(n) in general, and indeed it has to do some work to convert the sorted list into a max heap (which generally has the larger items earlier in the array). But the second part of Heapsort has a loop with O(n) passes, and each pass has a call to Max-Heapify at cost O(lg n), so the loop dominates with O(n lg n). If you work out detailed examples, you will see that once Build-Max-Heap is done, the array looks similar to a reverse-sorted array, since larger elements must be higher up in the tree = more towards the left. However, this does not help because Max-Heapify is called after putting one of the leaves in the root position; the leaf keys are small; and the one promoted has to propagate down the tree, making each pass O(lg n).

(b) What is the **asymptotic running time of Heapsort** using a Max-Heap on an array *A* of *n* elements that is already sorted in ***decreasing*** order? Justify your claim.

Solution: Also O(n lg n). Build-Max-Heap has to do less work in this case, as a reverse sorted list is already a max heap, so it does not have to actually swap anything (each call to Max-Heapify is constant time for the two assignments and conditional test). However, this is a constant reduction: Build-Max-Heap is still O(n) on reverse sorted data as it still runs the loop through half the items. Furthermore, the time savings does not matter: the loop that follows is O(n lg n), as it includes an O(lg n) Max-Heapify on each of the n items processed. The sorting does not reduce the O(lg n) passes, again because Max-Heapify is called after putting one of the leaves in the root position; the leaf keys are small; and the one promoted has to propagate down the tree.

3. (2 points) Ternary Heaps

You have just studied binary max-heaps. Suppose we want to use ternary max-heaps (each node has three children with smaller keys). Prove that a complete ternary heap of height h has **f(h)=(3^(h+1) - 1)/2** nodes.

Half a point for base case and 1.5 for remainder.

(a) Base case: $f(h=0) = (3^1 - 1)/2 = 1$

(b) Induction step: Assume formula is true for h. Prove $f(h+1) = (3^{h+2} - 1)/2$.

$f(h+1) = f(h) + 3^{h+1}$ // 3^{h+1} leaf nodes at level h+1

$= (3^{h+1} - 1)/2 + 3^{h+1}$ // substitute inductive hypothesis

$= (3^{h+1} - 1)/2 + (2 \cdot 3^{h+1})/2$

$= (3 \cdot 3^{h+1} - 1)/2$ // combine fractions

$= (3^{h+2} - 1)/2$ // QED