Massachusetts Institute of Technology — Handout 15
6.046J/18.410J: Introduction to Algorithms — March 10, 2000
Professors Piotr Indyk and Madhu Sudan

# Problem Set 3 Solutions

## Problem 3-1. Heap Delete

The following code will perform HEAP-DELETE. The basic idea is to replace $A[i]$ with the last element in the heap and then float this element up or down as necessary.

HEAP-DELETE$(A, i)$
$A[i] \leftarrow A[\mathit{heap\text{-}size}(A)]$
$\mathit{heap\text{-}size}(A) \leftarrow \mathit{heap\text{-}size}(A) - 1$
**if** $i > 1$ and $A[i] > A[\lfloor i/2 \rfloor]$
  **then** FLOAT-UP-HEAP$(A, i)$
  **else** HEAPIFY$(A, i)$

FLOAT-UP-HEAP$(A, i)$
**while** $i > 1$ and $A[\lfloor i/2 \rfloor] < A[i]$
  **do** exchange $A[i] \leftrightarrow A[\lfloor i/2 \rfloor]$
    $i \leftarrow \lfloor i/2 \rfloor$

Since it is clear that this algorithm removes element $i$ from the heap, we need only show that the heap property is satisified upon termination to prove correctness. We consider two cases.

Case I: $(A[i] \leq A[\lfloor i/2 \rfloor])$
In this case the heap property above node $A[i]$ is satisifed because the value of $A[i]$ is less than its parent. Thus after the heap property is restored $A[i]$ will be somewhere in the subtree rooted at $i$. The trees rooted at $i$s children already maintain the heap property since $A$ was a heap before the delete operation and the only node changed so far is $i$. In this case we call HEAPIFY which has been previously proven to restore the heap property in the tree rooted at node $i$ under these conditions.

Case II: $(A[i] > A[\lfloor i/2 \rfloor])$
In this case the heap property is satisfied in the tree rooted at $i$; however, the value of $A[i]$ exceeds its parent and violates the heap property. We call FLOAT-UP-HEAP to fix this. Each iteration of the **while** loop swaps node $A[i]$ with its parent (and fixes the index $i$ to point at the new location). This always maintains the heap property in the tree rooted at $A[i]$. The algorithm terminates when $A[i]$ no longer exceeds its parent or reaches the root of the tree. In both of these cases the entire tree now satisifies the heap property.

Since the inserted element can, at most, float up or down the entire height of the tree, the procedure always terminates and runs in $O(\lg n)$ time.

## Problem 3-2. $k$-ary Heaps

**(a)** A $k$-ary heap can be represented in a 1-dimensional array as follows. The root is kept in $A[1]$, its $k$ children are kept in order in $A[2]$ through $A[k+1]$, their children are kept in order in $A[k+2]$ through $A[k^2+k+1]$, and so on. The two procedures that map a node with index $i$ to its parent and to its $j$th child (for $1 \leq j \leq k$), respectively, are;

K-ARY-PARENT$(i)$
    **return** $\lfloor (i-2)/k + 1 \rfloor$

K-ARY-CHILD$(i, j)$
    **return** $k(i-1) + j + 1$

To convince yourself that these procedures really work, verify that

$$\text{K-ARY-PARENT}(\text{K-ARY-CHILD}(i, j)) = i \ ,$$

for any $1 \leq j \leq k$. Notice that the binary heap procedures are a special case of the above procedures when $k = 2$.

**(b)** Ben is correct when he argues that for $k = 1$ the only operation required in the heapsort algorithm is BUILD-HEAP. As each node in the heap has only one child, the heap will be comprised of a single root-to-leaf chain containing the sorted elements. Since our array representation of a heap guarantees that a parent will always precede its children in the array, the elements in the array representing this heap will be correctly sorted.

Ben's argument fails when he claims that this BIULD-HEAP operation will run in $O(n)$ time. Look at the code for BUILD-HEAP.

BUILD-HEAP$(A)$
    *heap-size*$[A] \leftarrow$ *length*$[A]$
    **for** $i \leftarrow \lceil \mathit{length}[a]/k - 1 \rceil$ **downto** 1
        **do** HEAPIFY$(A, i)$

Since the elements in the subarray $A[(\lceil n/k - 1 \rceil)..n]$ are leaves in the tree, BUILD-HEAP goes through the remaining nodes and runs HEAPIFY on each one. Before each call to HEAPIFY, the subarray $A[i+1..n]$ is sorted. Since the unary heap has only one path from the root down to the leaf, the next call to HEAPIFY simply walks along the subarray $A[i+1..n]$ and inserts $A[i]$ so that the subarray $A[i..n]$ is now sorted. This is exactly the operation of INSERTION-SORT which has worst case running time $O(n^2)$.

**(c)** HEAPSORT and its subroutine BUILD-HEAP do not perfom any comparisons between elements themselves. Instead they rely on calls to HEAPIFY to perform the actual comparisons. Each execution of HEAPIFY performs $O(k)$ comparisons to determine the maximum element between $A[i]$ and its $k$ children. Thus if we count the worst case number of calls to HEAPIFY, including recursive calls, we can compute the total number of comparisons performed by HEAPSORT.

The first operation performed by HEAPSORT is BUILD-HEAP. BUILD-HEAP makes $O(n)$ calls to HEAPIFY, each of which can recurse at most $h$ times, where $h$ is the height of the heap. This yields an upper bound of $O(nhk)$ on the number of comparisons performed by BUILD-HEAP. A more detailed analysis of BUILD-HEAP shows that it actually performs $O(nk)$ comparisons, but $O(nhk)$ is a sufficient bound to complete this problem.

After building the heap, HEAPSORT repeatedly extracts the maximum element from the heap and places it at the end of the array. It moves the previous occupant of this position to the head of the heap in order to make room for the extracted element and ensure that no holes are created in the heap. A call to HEAPIFY then restores the heap property. Each of these HEAPIFY operations performs at most $O(hk)$ comparisons for a total worst case upper bound of $O(nhk)$ comparisons.

Thus the total number of comparisons performed during the execution of HEAPSORT in the worst case is $O(nhk)$. Since the heap is represented as a $k$-ary tree, $h = \Theta(log_k n)$. This gives a comparison upper bound in terms of $n$ and $k$ of $O(nklog_k n)$.

To find the value of $k$ that minimizes this expression, notice that $nklog_k n = nk \ln n / \ln k$. Minimize $\frac{k}{\ln k}$ by taking its derivitive with respect to $k$ and setting it equal to zero. This yields $k = e = 2.718...$ as the minimizing value. Since this problem requires that $k$ be an integer, compare $\frac{2}{ln 2}$ and $\frac{3}{ln 3}$ to determine that $k = 3$ is the actual integral value that will minimize the asymptotic number of comparisons performed by HEAPSORT.

**Problem 3-3. Pancake Sorting**

Let FLIP($i$) perform a flip operation with the spatula inserted beneath pancake $i$. Here is one algorithm that sorts the pancakes efficiently. To sort the entire stack, call PANCAKE-SORT$(A, n)$.

PANCAKE-SORT$(A, i)$
**if** $i < 1$
  **then** *return*
$j \leftarrow$index of largest pancake in $A[1..i]$
FLIP($j$)
FLIP($i$)
PANCAKE-SORT$(A, i-1)$

This procedure takes an unsorted stack and places the largest pancake in it at the bottom using two flips. It then recursively sorts the remaining pancakes on the top of the stack. This algorithm performs two flips per pancake for a total of $O(n)$ flips.

Since the number of comparisons performed does not affect the running time, we are free to use a simple algorithm to find the largest pancake in the stack. One such algorithm is to scan the entire stack while keeping track of the largest pancake seen so far. This requires $O(n)$ comparisons for a total of $O(n^2)$ comparisons to sort the entire stack.

**Problem 3-4. Sorting Strings and Things**

**(a)** Note that we can't just use radix sort as described in class, because that would cost the number of strings times the length of the longest. We could have $n/2$ one-character strings and one $n/2$-character string, causing $O(n^2)$ running time.

Instead we'll use the property that a string with a bigger first letter is necessarily bigger. First, we sort the strings on the first letter using counting sort. Note that empty strings should be taken as a special case and put first. Now looking only at the second letter on, we recursively sort each group of the strings with the same first letter. Observe that this is a effectively a most-significant-bit-first radix sort, as opposed to the least-significant-bit-first radix sort described in class.

It should be clear that this algorithm does the right thing. Now to analyze how long it takes, we ask how many times each string is involved in a counting sort. This is at most the number of characters it has plus one (the "plus one" is because it may have to be sorted as an empty string at some point; for example, "ab" and "a" end up in the same group in the first pass and are then ordered based on "b" and "" in the second pass, so the "a" is involved in its length plus one counting sorts.) Adding up over all strings, we get $O(n)$ time. Note that a counting sort takes time proprotional to at least the number of buckets, which is the range of each digit, but the number of counting sorts is also $O(n)$ by the above argument, so as long as the digits have a constant range (eg $a \dots z$), the time bound holds.

**(b)** The same algorithm doesn't work, because unlike strings, integers get bigger if you add characters to the end. However, we can exploit this property that longer integers are necessarily bigger. We can sort them *by length* first with counting sort, and then sort the numbers of the same length with radix sort.

It takes $O(n)$ time to compute the lengths, and $O(\text{number of integers}) = O(n)$ time to sort them using counting sort. Now we just need to sort each "pile" of integers of the same length. Radix sort takes $O((k + C)d)$ time, where $k$ is the number of elements being sorted, $d$ is the number of passes per element (number of digits), and $C$ is the number of values a digit can have. Thus, per "pile," radix sort simply takes time proportional to the number of digits in the pile. Therefore, the total time of all the radix sorts is $O(n)$. The total running time for this algorithm is $O(n)$.