# CS 457, Data Structures and Algorithms I
# Fourth Problem Set Solutions

November 8, 2018

1. You are given a red-black tree $T$ with 15 internal nodes (nodes that hold key values) that form a *full* binary tree of height 3 (i.e., a full binary tree of height 4 if you include the NIL leaves). Can you assign colors to the nodes so that a call to RB-INSERT$(T, z)$ for *any* new key value $z.key$ will cause RB-INSERT-FIXUP$(T, z)$ to change the color of the root to red before switching it back to black? The initial assignment of colors needs to obey the red-black properties. If such a color assignment exists, then provide a sequence of 15 numbers whose insertion (in that order) would lead to such a tree, along with a figure of the resulting tree. If not, then explain why such an assignment cannot exist, using the fact that the tree needs to satisfy the red-black properties. You can use the applet `https://www.cs.usfca.edu/~galles/visualization/RedBlack.html` to better understand how red-black trees work.

   **Solution:**

   Since we want to ensure that the root changes is colored red we need case 1 of the three possible RB-INSERT-FIXUP cases to propagate up the tree. This ensures that the 2 children of the root must be red. Further, since we know that we want case 1 to propagate up regardless of where we insert the new node and the new node is colored red for insertion, we must have all nodes of height 0 in our full tree be colored red. Thus, we know the root is black, its children must be red, its grandchildren must be black, and its great-grandchildren must be red. One such insertion which allows this to happen is if the following list is inserted in order: $[16, 8, 24, 4, 12, 20, 28, 2, 6, 10, 14, 18, 22, 26, 30]$.

2. A full $k$-ary tree is a (rooted) tree whose nodes either have exactly $k$ children (internal nodes) or have no children (leaves). *Using induction*, formally prove that every full $k$-ary tree that has $x$ internal nodes has exactly $kx + 1$ nodes in total. Note that for full *binary* trees, i.e., when $k = 2$, this would imply that the total number of nodes is $2x + 1$, which would verify what Cameron suggested in class (that the total number of nodes of full binary trees is always odd).

   **Solution:**

   We will use induction on the number of internal nodes ($x$ in the problem description). The base case is a full $k$-ary tree with 1 internal node. Since the tree must be full, there is only one possible tree in our base case – a root with $k$ children. Therefore, this tree has $k + 1 = k(1) + 1$ nodes, so the base case holds. Suppose the statement is true for all trees with $1 \leq x < n$ internal nodes, that is, any full $k$-ary tree with $x$ internal nodes (where $1 \leq x < n$) has $kx + 1$ nodes total. Consider any full $k$-ary tree $T$ with $n$ internal nodes. Since $T$ is a full tree, there must be at least one internal node $v$ whose $k$ children are sibling leaves. Consider the tree $T'$ obtained by removing all $k$ of these sibling leaves. In $T'$, $v$ has now become a leaf and since we only removed children of $v$ in $T$ to generate $T'$, all other leaves in $T$ are present in $T'$ and all other internal nodes in $T$ remained internal nodes in $T'$. Therefore, $T'$ has $n - 1$ internal nodes (one fewer internal node than $T$) and our inductive hypothesis applies. Thus, $T'$ has $k(n - 1) + 1$ total nodes. Since we removed exactly $k$ nodes from $T$ to generate $T'$, $T$ has $k(n - 1) + 1 + k = kn + 1$ internal nodes, and our induction holds.

3. (20 pts) Using proof by induction, show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height $h$ in any $n$-element heap. Make sure to clearly state your base case and your inductive step arguments. Also note that you need to prove the aforementioned upper bound for the number of nodes of height $h$ for *every* value of $n$ and *every* $n$-element heap.

**Solution:**

In an attempt to make this answer helpful, I will first provide an attempt that fails, and then provide the proof that works. Our *failed attempt* tries to prove the desired inequality using induction over the height of the tree starting from the root, which has the maximum height $\lfloor \log n \rfloor$, and then proving that the property remains true for every height from top to bottom, using the inductive step.

FAILED ATTEMPT...

**Base Case:** Since a heap is a binary tree, we know that the height of the root is $\lfloor \log n \rfloor$, and we also know that there is just a single node at that height, the root. Therefore, it suffices to show that $\lceil n/2^{h+1} \rceil \geq 1$ for $h \leq \lfloor \log n \rfloor$. But then, this implies that

$$\lceil n/2^{h+1} \rceil \;\geq\; \lceil n/2^{\lfloor \log n \rfloor + 1} \rceil \;\geq\; \lceil 1/2 \rceil \;=\; 1,$$

which concludes the proof for the base case.

**Inductive Step:** We start by assuming that the desired inequality is true for the number of nodes at some height $h$, and our goal is to prove that it remains true for the number of nodes at height $h-1$. Since a heap is a binary tree, we know that any node at height $h$ has at most two children at height $h-1$. Our assumption above says that the number of nodes at height $h$ is at most $\lceil n/2^{h+1} \rceil$. Therefore, the number of nodes at height $h-1$ can be at most two times that, i.e., at most $2\lceil n/2^{h+1} \rceil$. Now, it would be tempting to argue that $2\lceil n/2^{h+1} \rceil \leq \lceil 2n/2^{h+1} \rceil = \lceil n/2^{(h-1)+1} \rceil$, which would complete the proof, but unfortunately the first inequality is *not true*. To verify this fact note that if $n = 2$ and $h = 1$ then this inequality would imply that $2\lceil 2/2^2 \rceil \leq \lceil 4/2^2 \rceil$, i.e., that $2 \leq 1$.

SUCCESSFUL ATTEMPT...

Our successful attempt proves the induction bottom up using the fact that the number of leaves of a heap are $\lceil n/2 \rceil$. To verify this property of heaps, consider the function PARENT(i) on Page 152 of your textbook that computes that index of $i$'s parent. This implies that the last parent, i.e., the parent of the $n$-th node has index $\lfloor n/2 \rfloor$, which implies that the remaining $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$ are leaves.

**Base Case:** We consider the case where $h = 0$, and our goal is to show that for every value of $n$ and any $n$-element heap, the number of nodes at height 0 is at most $\lceil n/2^{0+1} \rceil = \lceil n/2^1 \rceil = \lceil n/2 \rceil$. But, the nodes at height 0 are the leaves of the heap and, as we showed above, the number of leaves is in fact at most $\lceil n/2 \rceil$, which proves our base case.

**Induction Step:** We now assume that for any value of $n$ and any $n$-element heap, the number of nodes at height $h$ is at most $\lceil n/2^{h+1} \rceil$, and we wish to prove that for any value of $n$, and any $n$-element heap, the number of nodes at height $h+1$ is at most $\lceil n/2^{h+2} \rceil$. Given any value of $n$ and any $n$-element heap, we observe that removing all of its $\lceil n/2 \rceil$ leaves leads to a new heap whose number of nodes is $n' = n - \lceil n/2 \rceil = \lfloor n-2 \rfloor$. In this new heap, the height of each node that was not removed has dropped by 1. As a result, the nodes of height $h+1$ in the original heap are now that nodes of height $h$ in the new heap. Using our assumption, we know that the number of nodes at height $h$ in the $n'$-element heap that we have created is at most $\lceil n'/2^{h+1} \rceil = \lceil \lfloor n/2 \rfloor / 2^{n+1} \rceil \leq \lceil n/2^{h+2} \rceil$, which completes the proof.

4. (25 pts) You start a new tech business and you reach the point where you have offices both in the east coast and in the west coast. There are different operating costs involved in running your business in each location, so you may need to move back and forth across the coasts in order to minimize the total costs. However, there are moving costs involved in traveling from one coast to the other which complicate your problem. More formally, for each month $i = 1, \ldots, n$ you have an operating cost $e_i$

associated with running your business in the east coast during that month and an operating cost $w_i$ for the west coast. You can begin working in either coast, but after any subsequent travel from one coast to another you suffer a fixed travel cost of $c$. Given input $\{e_1, \ldots, e_n\}$, $\{w_1, \ldots, w_n\}$, and $c$, provide a $O(n)$ time algorithm that outputs a plan defining which coast you will work from each month, and minimizes your total cost (operating costs plus travel costs).

For example, if the number of months was $n = 5$, a possible plan could be to work from the east coast during the first two months, then spend the next two months working from the west coast, and return to the east coast for the fifth and final month. The total cost in this case would be $e_1 + e_2 + c + w_3 + w_4 + c + e_5$, where the two costs of $c$ are due to the change of coast. Provide detailed pseudocode for your proposed algorithm, as well as an explanation regarding why it works and why its running time is indeed $O(n)$.

**Solution:**

There is already a notion of order in this problem (the order given by the months) so we can move directly towards thinking about this problem recursively. To determine what the optimal choice for the last month is, we would find the minimum cost plans ending on the east and west coasts for the second to last month. We could then find the minimizing plan overall by selecting the appropriate previous plan and current month combination. This observation suggests an algorithm for this problem. We will maintain the minimum cost of the plans ending on the east and west coast every month (to find the minimum cost plan overall). To do this, we define two cost variables $\hat{e}$ and $\hat{w}$ which originally equal 0. In month $i$, we compare $e_i + \hat{e}$ to $e_i + c + \hat{w}$ and store the minimum in $\hat{e}$. We similarly compare $w_i + \hat{w}$ to $w_i + c + \hat{e}$ and store the minimum in $\hat{w}$. After we have completed this $n$ times (one for each month), the minimum of $\hat{e}$ and $\hat{w}$ is the optimum cost. However, we also need to be able to reconstruct the plan which gave us this optimum cost. To do this, we can store two auxiliary arrays $\hat{E}$ and $\hat{W}$. For a month $i$, $\hat{E}[i]$ will either contain $e$ if $e_i + \hat{e}$ is less than (or equal to) $e_i + c + \hat{w}$ and $w$ otherwise. Similarly, $\hat{W}[i]$ will contain $w$ if $w_i + \hat{w}$ is less than (or equal to) $w_i + c + \hat{e}$ and $e$ otherwise. We can then recreate the plan in reverse by beginning at month $n$. If $\hat{e}$ is less than (or equal to) $\hat{w}$ at the end of our process our plan must end on the east coast and we examine $\hat{E}[n]$ (otherwise our plan must end on the west coast and we examine $\hat{W}[n]$). This entry tells us the coast location for the previous month, that is, if $\hat{E}[n] = e$ then we add $e$ as the location for month $n-1$ in our plan and examine $\hat{E}[n-1]$ and if $\hat{E}[n] = w$ then we add $w$ as the location for month $n-1$ and examine $\hat{W}[n-1]$, recursively. In effect, we are "remembering" the location our office occupied in the previous month for every month and both possibilities for each month. This allows us to reconstruct the minimum cost plan overall by retracing backward from the final month.

5. (25 pts) You are planning to attend a music and arts festival that hosts multiple events that you are interested in, but many of these events may (partially) overlap. Assume that you get some value $v_i$ from attending an event $i$, but you get this value only if you are there for the whole duration of the event; you get no value if you have to miss part of it. Given such a set of events, $E$, your goal is to choose a non-overlapping set of events, $E' \subseteq E$, to attend, aiming to maximize your total value. More formally, for each event $i \in E$ you have a start time $s_i$ and a finish time $f_i$ and you want to design an algorithm that runs in time $O(n \log n)$, where $n$ is the total number of events, and the output of the algorithm is a set $E' \subseteq E$ such that for any two events $i, j \in E'$ we have $f_i \leq s_j$ or $f_j \leq s_i$ (no overlap) and the total value $\sum_{i \in E'} v_i$ is maximized. Provide detailed pseudocode for your proposed algorithm (you can use sorting as a subroutine), as well as an explanation regarding why it works and why its running time is indeed $O(n \log n)$.

**Solution:**

We first sort the events by finish time to add structure to the problem. We may do so in $O(n \log n)$ using mergesort. We make the observation that the optimal solution either contains the last event to finish or it doesn't. Armed with this fact, we can think about the recursive solution to the problem with $n$ events as the maximum of these two options. At this point, this appears quite similar to the

longest increasing subsequence problem we discussed in recitation. We solve the remainder of this problem by creating an array $M$ of length $n$ where $M[i]$ is the value of the optimal solution using only events up to $i$ and passing left to right through the sorted (by finish time) array of events. For each event $i$, we add $v_i$ to $M[j]$ where $j$ is the event with the largest finish time smaller than the start time of $i$ which we find using binary search. We compare this value to $M[i-1]$ and store the larger of the two in $M[i]$. The optimal value we could achieve would then be stored in $M[n]$ at the end of this iterative process. To find the actual set of events, we may examine the array $M$ again. Beginning at index $n$, we compare $v_n + M[k]$ where $k$ is the event with largest finish time that ends before the start time of $n$ and $M[n-1]$. If $v_n + M[k]$ is greater (or equal) we add $n$ to our set of events and recursively perform this check on $k$. Otherwise, we do not add $n$ and we recursively perform the check on $n-1$. This reconstructs the set of events which led to the value seen at $M[n]$, the optimal value. Since we performed $O(n)$ constant time arithmetic operations were constant time and $O(n)$ binary searches, the total process takes $O(n \log n)$. To see that this works, observe that we are always maintaining the best possible solution up to job $i$ in $M[i]$ for each $i$ by comparing the best solution containing $i$ and the best solution not containing $i$. By doing this for every index all the way to $n$, we know that the optimal solution overall is stored at index $n$.