COSC 302: Analysis of Algorithms — Spring 2018
Prof. Darren Strash
Colgate University

Problem Set 5 — Heaps, Non-comparison sorts, Red-black trees, Hashing
Due by 4:30pm Friday, Mar. 9, 2018 as a single pdf via Moodle (either generated via LaTeX, or concatenated photos of your work). Late assignments are not accepted.

This is an *individual* assignment: collaboration (such as discussing problems and brainstorming ideas for solving them) on this assignment is highly encouraged, but the work you submit must be your own. Give information only as a tutor would: ask questions so that your classmate is able to figure out the answer for themselves. It is unacceptable to share any artifacts, such as code and/or write-ups for this assignment. If you work with someone in close collaboration, you must mention your collaborator on your assignment.

Suggested practice problems, from CLRS: Ch 11.1 (1 and 2); 11.2-3; 12.2 (3, 4, and 5); 12.3-5; 13.3 (1, 2, and 4)

1. In this problem, we will investigate d-ary max-heaps: A d-ary heap is one in which each node has at most d children, whereas, in a binary heap, each node has at most 2 children.

   (a) Describe how to store/represent a d-ary heap in an array.

   Solution: Just as with binary max-heaps, we will store elements with an implicit tree representation—however, this time as a d-ary (and not binary) tree. The root will be at $A[0]$, its d children will be stored in indices 1 to d. Thinking of the max-heap as a d-ary tree, the j-th child of node i will be stored in position $di + j$.

   (b) What is the height of a d-ary heap in terms of d and n?

   Solution: We noticed that each level has some power of d number of nodes on it. In particular, level 0 has $d^0 = 1$, nodes, level 1 has $d^1 = d$ nodes, and in general level i has d^i nodes. Summing the number of nodes on each level of the tree equals the total number of nodes n. Therefore, we want to find the height h such that.

$$\sum_{i=0}^{h-1} d^i \leq n \leq \sum_{i=0}^{h} d^i.$$

   Solving for h we get that

$$\begin{aligned}
\sum_{i=0}^{h-1} d^i \leq \quad & n \quad && \leq \sum_{i=0}^{h} d^i, \\
\frac{d^h - 1}{d-1} \leq \quad & n \quad && \leq \frac{d^{h+1} - 1}{d-1} && \text{when } d > 1, \\
d^h \leq \quad & n(d-1) + 1 \quad && \leq d^{h+1}, \\
h \leq \log_d & [n(d-1) + 1] \leq h + 1.
\end{aligned}$$

Therefore, asymptotically, the height is both upper and lower bounded by the function $\log_d [n(d-1)+1]$, and therefore the height is $\Theta(\log_d [n(d-1)+1]) = \Theta(\log_d n)$.

(c) Re-write function PARENT(i) for d-ary heaps, and give a new function CHILD(i,j) that gives the j-th child of node i (where $1 \leq j \leq d$).

$$\text{PARENT}(i) = \left\lfloor \frac{i-1}{d} \right\rfloor, \text{CHILD}(i,j) = di + j.$$

(d) Describe, and give pseudocode for, the algorithm MAX-HEAPIFY(A,i) for d-ary heaps and give a tight analysis for the worst-case running time of your algorithm.

Solution:

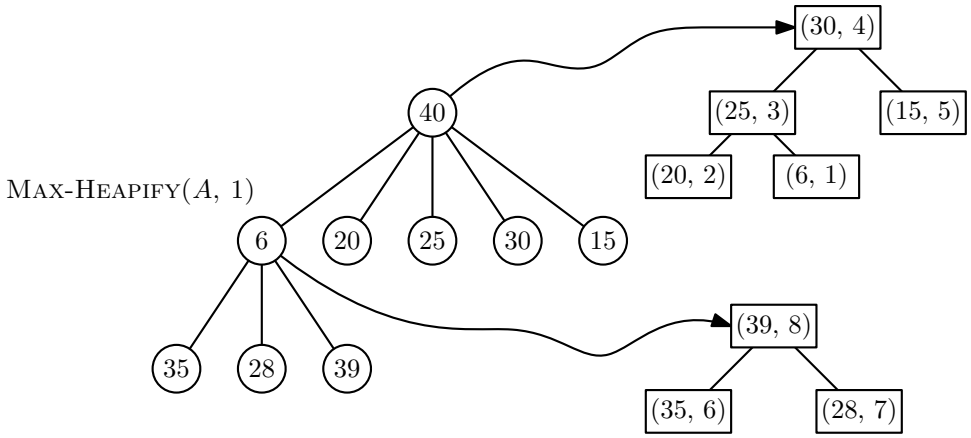
Algorithm 1 MAX-HEAPIFY for d-ary heaps.

```
proc MAX-HEAPIFY(A, i)
 1: largest ← i
 2: for j ← 1 to d
 3:     if CHILD(i, j) ≤ heap-size[A] and A[CHILD(i, j)] > A[largest] then
 4:         largest ← CHILD(i, j)
 5:     end if
 6: end for
 7: if largest ≠ i then
 8:     swap(A[i], A[largest])
 9:     MAX-HEAPIFY(A, largest)
10: end if
```

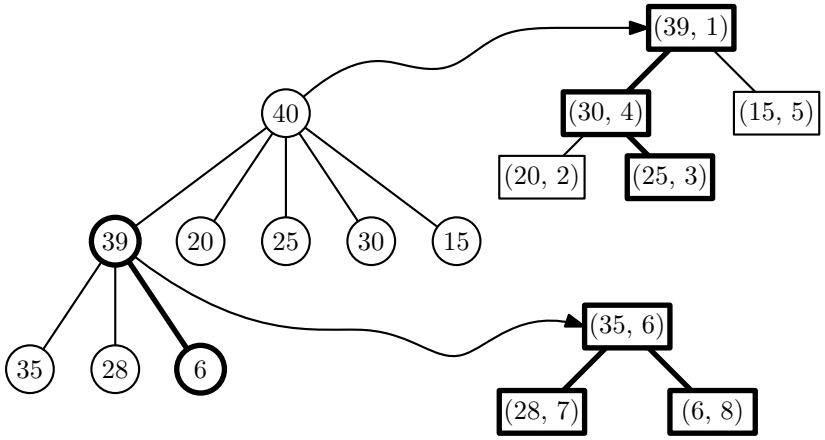
The running time of MAX-HEAPIFY is $\Theta(d \log_d n)$. First, we give an upper bound: First, it is $O(d \log_d n)$ since each call to MAX-HEAPIFY takes at most $O(d)$, and it can recursively call itself at most the height of the tree $O(\log_d n)$ times. However, it is also $\Omega(d \log_d n)$ since there is one instance that actually does this amount of work: suppose we are calling MAX-HEAPIFY on the root $A[1]$ and this element is the minimum element in the max-heap with exactly $\frac{d^k-1}{d-1}$ elements. Then this element *must* be at the bottom most level in order for the heap property to be maintained. Since the tree has height $\Omega(\log_d n)$, and each node has exactly d children, this will require $\Omega(d \log_d n)$ time to move this element to be a leaf.

(e) Describe (semi-formally) how to implement MAX-HEAPIFY(A,i) in $O((\log_d n) \lg d)$ time. *(Hint: you need auxiliary data structures; the heap itself is not sufficient.)*

Solution: My current implementation of MAX-HEAPIFY takes time $O(d \log_d n)$, since it takes $O(d)$ time to find the child with the largest element. To reduce this factor d to $\lg d$, we need an auxiliary data structure on the children of each node. This auxiliary data structure should make it possible to find the maximum value among all children of the given node in $O(\lg d)$ time. For each node, we keep a pointer to a *binary* max-heap of its children, which has size d. This allows us to quickly access the maximum-valued child in time $O(1)$. However, we also need auxiliary information to be able to



(a) Our max-heap before calling MAX-HEAPIFY(A, 1)



(b) After, with changes emboldened.

swap elements in line 8 of the above pseudocode. To do this, we store the index in A along with the elements in the auxiliary max-heaps, and when we swap elements $A[i]$ and A[largest], we also must swap elements between their max-heaps, and update their indices. This requires two EXTRACT-MAX and two INSERT calls to max-heaps of size d, which takes time $O(\lg d)$ overall. Thus, each call to MAX-HEAPIFY performs $O(\lg d)$ operations (excluding recursion), and will be called recursively $O(\log_d n)$ times, taking $O((\log_d n)(\lg d))$ time overall.

2. **(From homework 4, skip if already submitted)** Problem 8.2-4 from CLRS: Describe (semi-formally) an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a..b]$ in $O(1)$ time. Your algorithm should use $\Theta(n+k)$ preprocessing time.

   Solution: Execute the first half of counting sort: Create an array $C[0..k]$ with each $C[i]$ initialized to zero, and iterate over the input array (call it $A[1..n]$) in order from 1 to n in $O(n)$ time, incrementing $C[i]$ for each value i encountered. We then iterate over each index i of C in order from 0 to k in $O(k)$ and keep a cumulative count of all numbers of value i or less, which we store at $C[i]$. This takes time $O(n)+O(k)$ time, which is $O(n+k)$. The array C is our data structure.

   To answer a query, we execute the following method, which takes $O(1)$ time.

```
proc QUERY-COUNT(C, a, b)
1: if a > k or  b < 0 then// [a..b] does not overlap [0..k]
2:     return 0
3: else if a ≤ 0 then
4:     return C[min{b, k}]// count all elements less than b
5: else
6:     return C[min{b, k}] − C[a − 1]
7: end if
```

   This method computes the number of elements less than or equal to b and subtracts from it the number of elements less than a, giving the number of elements between a and b inclusive.

3. Problem 13.3-5 from CLRS. (Describe semi-formally.) *(Hint: Follow the structure for an invariant.)*

Solution:

Proof. (By Induction)

Base case: $n = 2$: Then two calls to RB-INSERT have been made. The first made a black root, and the second made a red node that is either the left or right child of the root.

Inductive step: Suppose that the red-black tree T_{i-1} with $i-1$ nodes has at least 1 red node; we show that by inserting another node, there is still at least 1 red node in T_i with i nodes.

Let z be the inserted node. And recall that a node is inserted as a red node. We show that either z or z's parent stays red after insertion, and therefore the new tree has at least one red node. Without loss of generality, we look at the case that z's parent is a left child, otherwise all cases are symmetric, exchanging right/left.

There there are 4 cases (of RB-INSERT-FIXUP):

Case 0 z's parent is not red. Then z remains red.

Case 1 z's uncle is red. Then z's parent, grand parent, and uncle are recolored, but z stays red.

Case 2 z's uncle is black and z is a left child. Then z's parent and grand parent are recolored, and a rotation occurs, but z remains red.

Case 3 z's uncle is black and z is a right child. Then a rotation occurs, and z's red parent takes the place of z in case 2 above. By case 2, this node stays red.

Thus, after each call to RB-INSERT, there is at least one red node in the red-black tree. □

4. **(Previous exam question)** Let $A[1..n]$ be an array of non-integers taken from some set K of size $k > 1$. *(Note: For this problem, you are not given the set K or k; this is only to illustrate that there are k distinct non-integer numbers. We only have access to elements through A. Further, note that k may be small or large: from constant to even larger than n.)*

   (a) Describe an algorithm that sorts A in expected time $O(n + k \lg k)$, and describe why it has this running time.

   (b) What is the worst-case running time of your algorithm? Justify your answer.

   (Question pushed to homework 6; solution to be given with homework 6.)