COSC 302: Analysis of Algorithms — Spring 2018
Adam Pettway
Colgate University
Problem Set 5 — Heaps, Non-comparison sorts, Red-black trees, Hashing

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) We can represent a d-heap in an array which the second element is the root. Then for any parent node x it's children are located $(x * d) + 1, (x * d) + 2, ...(x * d) + d$. An for any child can find it's parent by $(x - 1)/d$.

   (b) If the d-ary heap is completely filled then the ith level will be d^i. So the to find the nodes up to the last level of the heap at level l would be:

   $$\sum_{i=0}^{l} d^i = \frac{d^{l+1} - 1}{d - 1} \tag{1}$$

   This function describes the amount of nodes, or n, so we need to solve for l which would be the height.

   $$\begin{aligned} n &= \frac{d^{l+1} - 1}{d - 1} \\ n(d-1) &= d^{l+1} - 1 \\ log_d(n(d-1) + 1) - 1 &= l \end{aligned} \tag{2}$$

   This height is $\Theta(log_d(n(d-1)+1)-1)$.

   (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 $1 \leq j \leq d$).

1: **function** PARENT(i)
2: **return** $(i - 1)/d$
3: **function** CHILD(x, j)
4: **return** $(i * d) + j$

(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.

```
1: function MAX-HEAPIFY(A,i)
2:     largest ← i
3:     for x ← 1 to d do
4:         if Child(A, x) > i then
5:             largest ← Child(A,x)
6:     if largest ≠ i then
7:         exchange A[i] with A[largest]
8:         Max-heapify(A,largest)
```

The worst-case running time would be $\Theta(log_d(n(d-1)+1)-1)$ if the value floats to the bottom of the tree. If the tree is balanced.

(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.)*
One can make the function take $O((\log_d n)\lg d)$ if you store the children of each node using a binary search tree. Finding the largest between i and the children would take $O(lg(d))$ since the tree would have d nodes and in the worst case bubble down to the leaves which would be $log_d(n)$ recursive calls. During each call you would traverse the children in $O(lg(n))$ which is $log_d(n)lg(n)$.

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.

```
1: function PRE-PROCESS(A,k)
2:     C[0...k]
3:     for i = 0 to k do
4:         C[i] = 0
5:     for j = 1 to A.length do
6:         C[A[j]] = C[A[j]] + 1
7:     for i = 1 to k do
8:         C[i] = C[i] + C[i − 1]
9:     A ← C
10: function RANGE(A,k,a,b)
11:     Pre-Process(A,k)
12:     return A[b] − A[a]
```

3. Problem 13.3-5 from CLRS. (Describe semi-formally.) *(Hint: Follow the structure for an invariant.)* The base case is two nodes. When a node is inserted due to line 16 the code, the node always starts as red. RB-Insert-Fixup is called and no violations are found. There are now three cases:

   (a) The first case is the newest node being added to a black node in which there is no violation an the newest node is red.

   (b) The second case is another red node is attached to a red parent and the uncle is black. Then there is a rotation. In either case of being the right or left child, the parent node become the new grandparent and is colored black. Thus the previous grand parent becomes the right child and is colored red. And the newest node stays red and is placed as the left child. In this case there's 2 red nodes.

(c) the third case is when the grand parent is a black node with two red children. Another red node is added as the black's grandchild which is red.This causes a violation so if the uncle of the newest node is red, both the uncle and the parent of the newest node become black and the grandparent node becomes red. Thus, the newest node and the grandparent node leads to 2 red nodes.