CSCI-GA.1170-003/004 Fundamental Algorithms February 28, 2020

Solutions to Problem 1 of Homework 4 (27 points)

Name: Rahul Zalkikar (rz1567) *Due: 8 pm on Thursday, February 27*

Collaborators: ua388, jni215

Recall that binary heaps are those where each non-leaf node (but with one possible exception) will have 2 children. We can extend the same definition to where each non-leaf node (but with one possible exception) will have k children. This is called a k-ary heap. Assume that we have a *max-heap*. In this question you will be asked to write pseudocode for *efficient* implementations of the standard heap algorithms. Make calls to other algorithms you have defined in the previous parts when you can. You will lose points otherwise. Additionally, ensure that you check boundary cases and throw appropriate error messages.

(a) (2 points) Assume that you have a k-ary heap implemented as a 1-dimensional array similar to what is done for a binary heap with the *root at* $A[1]$. Write the pseudocode for k-Parent(i) to return the index of the parent of node i. Consider the following pseudocode. Further, write a pseudocode k-Child(i, j) to return the index of the j-th child of node i for $j = 1, \ldots, k$.

Solution:

```
1 k-Parent(i, k)
2     return floor((i+k-2)/k)
```

```
1 k-Child(i, j, k)
2     return (k(i − 1) + j + 1)
```

□

(b) (3 points) Verify that your algorithm is correct by showing that $\forall j = 1, 2, \ldots, k$,

$$\text{k-Parent}(\text{k-Child}(i, j)) = i$$

Solution:

k-Parent(k-Child$(i, j, k), k)$ =k-Parent$((k(i-1)+j+1), k)$ =k-Parent$(\frac{ki+j-1}{k}, k)$ =k-Parent$(i \times \frac{j-1}{k}, k) = i$. This is because j ranges from 1 to k and k-Parent returns the floor. □

(c) (3 point) Write a pseudocode for k-Max-Heapify(A, i, k) that takes as input the array A, an index i into the array and k. Your algorithm will make calls as needed to the algorithms defined above.

Solution:

```
1 k-Max-Heapify(A, i, k)
2     max = i
3     for j = 1...k:
4         child =k-Child(i, j, k)
5         if (child ≤ A.heapsize) and (A[child] > A[i]):
6             max = child
7     if (max ≠ i):
8         swap A[max] with A[i]
9         k-Max-Heapify(A, max, k)
```

□

(d) (6 points) Derive a Θ bound for the height of the heap in terms of n and k where n is the size of the max-heap. Use this to derive analyze the running time of k-Max-Heapify in terms of k and n and also solve for the optimal value of k.

(**Hint**: Let $T(h)$ be the running time of k-Max-Heapify on a heap of height h. Recall that a leaf node has height 0.)

Solution:

The total number of nodes (if the tree was perfect) is $\sum_{i=1}^{n} k^i = \frac{k^{h+1}-1}{k-1}$ (for a height h).

So,

$\frac{k^{h+1}-1}{k-1} \geq n > \frac{k^h-1}{k-1}$

$k^{h+1} \geq n(k-1)+1 > k^h$

$h+1 \geq log_k(n(k-1)+1) > h$

Now we have $h \geq log_k(n(k-1)+1)-1 = O(log_k(n))$ and $h < log_k(n(k-1)+1) = \omega(log_k(n))$ (In other words the number of nodes at a level h is, at most, k^n)

So it is clear that the height $h = \Theta(\log_k(n))$

We know that a node compares its values and the values of its k children to find the max ($O(k)$ runtime), we can see that $T(n) = T(k \log_k(n))$ and the runtime of k-Max-Heapify is $O(k \log_k(n))$

Now we set $\frac{d}{dk}(\frac{k \times ln(n)}{ln(k)}) = ln(n) \times \frac{d}{dk}(\frac{k}{ln(k)}) = ln(n) \times \frac{\frac{d}{dk}(k) + ln(k) - k \times \frac{d}{dk}(ln(k))}{ln^2(k)} = \frac{ln(n) \times ((1) ln(k) - \frac{k}{k})}{ln^2(k)} = \frac{ln(n) \times (ln(k)-1)}{ln^2(k)} = 0$

We can see that $ln(n) \times (ln(k) - 1) = 0$ leads to $k = e$ as the optimal k.

□

(e) (3 points) Write a pseudocode for k-Increase-Key(A, i, key) that increases the value of $A[i]$ to key. What is the runtime of the algorithm in terms of k and n?

Solution:

```
1 k-Increase-Key(A, i, key, k)
2     if A[i] < key:
3         A[i] = key
4         while i > 1 and A[k-Parent(i, k)] < A[i]:
5             swap A[i] with A[k-Parent(i, k)]
6             i = A[k-Parent(i, k)]
```

Runtime: $O(\log_k n)$ if $A[i] < key$. □

(f) (3 points) Use the previous parts to write a pseudocode for k-Insert(A, key) to insert key into the max-heap A. Analyze the running time in terms of k and n.

Solution:

```
1 k-Insert(A, key)
2     A.heapsize = A.heapsize + 1
3     A[A.heapsize − 1] = key
4     k-Max-Heapify(A, A.heapsize − 1, A.heapsize)
```

Runtime: $O(\log_k n)$ □

(g) (3 points) Write a pseudocode for k-Extract-Max(A). Analyze the running time in terms of k and n.

Solution:

```
1 k-Extract-Max(A)
2     max = A[i]
3     A[1] = A[A.heapsize − 1]
4     k-Max-Heapify(A, 1, A.heapsize)
5     return max
```

Runtime (the same as k-Max-Heapify): $O(k \times \log_k n)$ □

(h) (4 points) Fill up the following table with the running time of the algorithms for a binary heap and k-ary heap. Now, assume that you know that there are a calls to Insertion and b calls to Max Extraction. What is the relationship between a, b and k for which k-ary heap is better?

Algorithm	Binary Heap	k-ary heap
Heapify		
Max Extraction		
Increase Key		
Insert		

Solution:

Algorithm	Binary Heap	k-ary heap
Heapify	$2 \times \log_2(n)$	$k \times \log_k(k)$
Max Extraction	$2 \times \log_2(n)$	$k \times \log_k(k)$
Increase Key	$\log_2(n)$	$\log_k(k)$
Insert	$\log_2(n)$	$\log_k(k)$

To find this relationship we have the equation $2b\log_2(n)+a\log_2(n)-bk\log_k(n)+a\log_k(n) \geq 0$.
Using a similar process as part 1(d), we can see that
$\frac{d}{dk}(bk\log_k(n) + a\log_k(n)) = 0$ gives us $\frac{k \times ln^2(n) \times (ln(k)-1)}{ln^2(n)} = \frac{a}{b}$
So we arrive at $k \times ln(k) - k = \frac{a}{b}$ and $k \times (ln(k) - 1) = \frac{a}{b}$

□