Problem sheet 2

COL106 (Data Structures), Semester I, 2018-19, IIT Delhi

Amitabha Bagchi
Department of CS&E, IIT Delhi

September 15, 2018

Exercise 1
Assume that you have an `IntBinTree` ADT which is a binary tree ADT that has integers as data. You can assume all the usual operations associated with a binary tree. Write pseudocode to perform the following tasks:

1. Check if a `IntBinTree` is in *min-heap order*, i.e., check if the key at every node is less than or equal to the keys at its children. Note that the tree may not be a complete binary tree, we are just checking the heap order property, not the heap structure property.

2. Check if a `IntBinTree` has the heap structure property. (This is a little tricky and requires some thinking. You may want to use a queue or something like that to help you).

Exercise 2
In a min heap the smallest element is found in the root. It is not hard to see that the second smallest element is one of the children of the root. What can we say about the k-th smallest element? Suppose we wanted to find the k-th smallest element in a heap a simple thing to do is to run k delete-min operations and save all the elements removed. After we have noted the k-th smallest element, we can simply re-insert all the elements. Now, answer the following questions

1. If we do what was suggested above, i.e. run k delete-min operations followed by reinserting the deleted elements does the heap look exactly the same as it did earlier, i.e., is every key in the same position as it was earlier. Either prove this is the case or show by an example that it is possible to change the position of keys by doing this.

2. Suppose we have to find the k-th smallest element without changing the heap in any way (i.e. no delete-min is allowed), how will we do it? What is the most efficient algorithm for it? Note that the algorithm suggested above takes $\theta(k \log n)$ time. Can we do something that grows only with k but not with n?

Exercise 3
Herlihy, Shavitt and Tzafrir introduced an interesting twist on open addressing with linear probing in 2008. They called it *Hopscotch hashing*. It works like this: As usual we are given an array A of size n, a hash code f' and a hash function $f(x) = f'(x) \mod n$. We now fix some $h \geq 1$. Insertion of an item x proceeds as follows:

1. If $A[f(x)]$ is empty then insert x in $A[f(x)]$.

2. Otherwise probe linearly through the array till you find an empty an empty slot, let's say j.

3. If $(f(x)+h) \mod n > j$ then insert x in $A[j]$, otherwise find an i between $f(x)$ and j such that $(f(A[i])+h) \mod n > j$. Move $A[i]$ to location j and repeat this step with i in the role of j (i.e. i is now the empty location. If step 3 fails to find an appropriate i then we declare failuer and choose a larger array of size n and rehash with a new hash function $f'(x) \mod 2n$.

Note that linear probing is done by incrementing modulo n, i.e. the linear probe wraps around if we get to the end of the array. Assume that the size of the largest island in the array is ℓ and answer the following questions about hopscotch hashing:

1. What is the algorithm for finding x? And what is the time taken to find an element?
2. What is the running time for the insertion algorithm described above?
3. Will the simple solution that places a * in a location where we delete work for deletion here as well?
4. If we assume that ordinary linear probing also rehashes once no empty slots are available, can we show that hopscotch hashing has to rehash earlier than linear probing? My guess is yes. So fix n, let your insertion items be integers and choose an appropriate hash code and show an example where hospcotch hashing rehashes earlier than linear probing. Then show an example where they both rehash at the same time.

Consider coding up hopscotch hashing and linear probing. Then run sequences of insert, delete and find operations and see which is faster and under what conditions. You will have to decide on how to set h. What happens if h is very large? What happens if h is very small?

Exercise 4

Suppose we have a sorted array $A[]$ with n elements, we know that binary search takes $\theta(\log n)$ *in the worst case* as we discussed in class. However there may be times when simple linear search that starts from $A[0]$ and goes sequentially through the indices is better. Consider the case where x is present in $A[i]$: the time taken by linear search is $\theta(i)$. This is better than binary search for the case where $i \in o(\log n)$. The same running time holds for an x which is not in $A[]$ but for which $A[i-1] < x < A[i]$.

Consider the following algorithm that also starts from $A[0]$ like linear search:

- Set $j = 0$
- While $x > A[j]$
  - Set $j = 2 + 1$
- Binary search for x in A between indexes $(j-1)/2$ and j.

First prove that this algorithm is correct. If you need to change something to make it correct, go ahead and do that and then prove it is correct. Then prove that the running time of this algorithm is $\theta(\log i)$ when run on an x for which either $A[i] = x$ or $A[i-1] < x < A[i]$.