Problem Set 4

Assigned: June 19
Due: June 26

Problem 1

A programmer proposes to implement a binary search tree using an array implementation similar to the array implementation of a heap. As in the implementation of a heap, the children of \( a[i] \) are at \( a[2i+1] \) and \( a[2i+2] \). For instance, the tree below would be implemented as the array \([15, 3, 24, -, 8, 22, 29, -, -, 7, 11, -, -, 26, -]\). (Another way of viewing this is that this is breadth-first search if you fill in all the missing spaces with null.)

![Binary Search Tree Diagram]

A. Describe the algorithm for searching for an element \( x \) in this tree.

B. This implementation is rarely if ever used in practice. What is the disadvantage of this method, as compared to constructing a binary search tree from dynamic objects?

C. Some time ago, I gave parts (A) and (B) as problems on an exam. A few of the students came up with the following answer to (B):

With this implementation, delete can be inefficient in the following case: Suppose that you delete node \( N \) with parent \( P \) and a single child \( C \) by making \( C \) a child of \( P \). Then all of the subtree under \( C \) will have to be moved in the array.

This answer is actually only half right. (I gave it full credit though, since it was certainly a good enough answer for an exam.)

C.a What is the worst case running time of deleting \( N \) if you implement delete as described above?
C.b Find a more efficient method of deleting an internal node with one child with this implementation.

In both C.a and C.b, running time should be given as a function of \( n \), the total number of elements, and \( h \), the height of the tree.

**Problem 2**

Modify the definition of a 2-3 tree so that it supports the following operations with the specified running times. You may assume that the reader understands the standard definition of a 2-3 (the one given in class, with all the values in the leaves); all you have to describe are the modifications that need to be made.

Note that we want a single (compound) data structure that supports all these operations, not different data structures for each operation.

- **add(x)**: Add element \( x \) to the set. Time: \( O(\log(n)) \)
- **delete(x)**: Delete element \( x \) from the set. Time: \( O(\log(n)) \).
- **element?(x)**: Is \( x \) in the set? Time: \( O(\log(n)) \).
- **index(i)**: Find the \( i \)th smallest element in the set. Time: \( O(\log(n)) \).
- **indexOf(x)**: Find the index of \( x \) in the set. Time: \( O(\log(n)) \).
- **subrange(i,k)**: Return \( k \) elements in the set in sequence starting with the \( i \)th. For instance, subrange(100,5) should return a list of the 100th, 101st, 102nd, 103rd, and 104th smallest elements. Time: \( O(k + \log(n)) \).
- **min()**: Find the smallest element in the set. Time: \( O(1) \).
- **max()**: Find the largest element in the set. Time: \( O(1) \).
- **median()**: Find the median element in the set. For instance if there are 99 or 100 elements in the set, return the 50th. Time: \( O(1) \).