

COS 226

Algorithms and Data Structures

Fall 2011

## Midterm Solutions

### 1. Union find.

- (a) *Impossible*: has a cycle 0-1, 1-2, 2-3, and 3-0 in the parent-link representation.
- (b) *Impossible*: the nodes 1, 2, 3, 4, and 5 must link to 0 when 0 is a root; hence, 0 would not link to 9 because 0 is the root of the larger tree.
- (c) *Impossible*: tree rooted at 0 has height  $9 > \lg 10$ .
- (d) *Possible*: 8-6, 6-1, 7-1, 5-1, 9-2, 3-0, 4-0, 2-0, 1-0.
- (e) *Impossible*: tree rooted at 0 has height  $4 > \lg 10$ .
- (f) *Impossible*: tree rooted at 0 has height  $3 > \lg 7$ .

### 2. Analysis of algorithms.

$$T(N) = \frac{1}{100,000} N^{5/3}.$$

When  $N$  increases by a factor of 8, the running time increases by a factor of 32. Thus,  $T(N) = aN^b$ , where  $b = \log_8 32 = \lg 32 / \lg 8 = 5/3$ . Since  $T(1000) = 1.00$ , we have  $1.00 = a \times 1000^{5/3}$ , which implies  $a = \frac{1}{100000}$ .

### 3. Data structures.

- (a)  $40 + 48N$  bytes.
- A `Node` uses 48 bytes of memory (16 bytes object overhead + 8 bytes inner class overhead + 8 bytes for `Item` reference 16 bytes for two `Node` references).
  - A `LinkedList` with  $N$  items uses 40 bytes (16 bytes object overhead + 16 bytes for two `Node` references + 4 bytes for an integer + 4 bytes of padding) plus the memory for the  $N$  nodes.

(b)

<code>addFirst(item)</code>	<i>prepend the item to the beginning of the list</i>	1
<code>get(i)</code>	<i>return the item at position <math>i</math> in the list</i>	$N$
<code>set(i, item)</code>	<i>replace position <math>i</math> in the list with the item</i>	$N$
<code>removeLast()</code>	<i>delete and return the item at the end of the list</i>	1
<code>contains(item)</code>	<i>is the item in the list?</i>	$N$

### 4. 8 sorting and shuffling algorithms.

0 5 6 9 4 3 8 2 7 1

## 5. Binary heaps.

(a)

0	1	2	3	4	5	6	7	8	9	10	11	12
-	Y	X	H	G	T	C	A	F	B	Q	R	-

(b)

0	1	2	3	4	5	6	7	8	9	10	11	12
-	Y	X	*P	G	T	*H	A	F	B	Q	R	*C

(c) H I J K L M N

Key is  $\leq N$  because it is a child of N in original heap and  $\geq H$  because it is a parent of H in final heap.

## 6. Red-black BSTs.

(a) T U V

Key is  $< W$  because it is in left subtree of W and  $> S$  because it is in right subtree of S.

(b)

B link between W and S

A. red

A link between ? and W

B. black

A link between S and Y

C. either red or black

B link between Q and S

(c)

	H	D	B	J
rotateLeft()	1	0	1	2
rotateRight()	0	0	0	3
flipColors()	0	0	0	3

## 7. Comparing two arrays of points.

(a) *Sort* the two arrays  $a[]$  and  $b[]$ , using the point's natural order (say, compare by  $y$ -coordinate, breaking ties by  $x$ -coordinate). Scan through the two sorted arrays and check that  $a[i]$  equals  $b[i]$  for each index  $i$  (using the point's natural order). We can achieve the performance requirements by using heapsort to sort.

(b) For each point in  $a[]$ , add  $a[i]$  to a *set*. For each point in  $b[]$  check that  $b[i]$  is in the set. We can achieve the performance requirements by using a hash table (either linear probing or separate chaining) to implement the set data type and by making the uniform hashing assumption.

## 8. Stabbing count queries.

The key observation is that the number of intervals containing  $x$  is equal to the number of intervals with a left endpoint less than  $x$  (number of intervals that start before  $x$ ) minus the number of intervals with a right endpoint less than  $x$  (number of intervals that end before  $x$ ). To keep track of these quantities, we build two BSTs, one containing the left endpoints as keys and one containing the right endpoints as keys. Recall that the `rank()` method returns the number of keys in a BST less than a given quantity. We can achieve the performance requirements by using a red-black BST for the BST.

For reference, here is a complete Java implementation:

```
public class IntervalStab {
    private RedBlackSET<Double> left, right;

    public IntervalStab() {
        left = new RedBlackSET<Double>();
        right = new RedBlackSET<Double>();
    }
    public void insert(double xmin, double xmax) {
        left.add(xmin);
        right.add(xmax);
    }
    public int count(double x) {
        return left.rank(x) - right.rank(x);
    }
}
```