COS 226 **Algorithms and Data Structures** **Fall 2010**

Midterm Solutions

1. **Analysis of algorithms.**

(a) For each expression in the left column, give the best matching description from the right column.

B. $1 + 2 + 4 + 8 + \ldots + N \sim 2N$

C. $1 + 2 + 3 + 4 + \ldots + N \sim \frac{1}{2}N^2$

B. $1 + 3 + 5 + 7 + \ldots + N \sim \frac{1}{4}N^2$

C. $\frac{1}{2}N^2$

C. $\frac{1}{2}N^2 + 100N \lg N$

B. N^2

D. N^3

A. $\sim \frac{1}{2}N^2$.

B. $O(N^2)$.

C. Both A and B.

D. Neither A nor B.

(b) For each quantity in the left column, give the best matching description from the right column.

B. Height of a weighted quick union data structure with N items.

C. Height of a binary heap with N keys.

A. Height of a left-leaning red-black BST with N keys.

C. Maximum function-call stack size when (top-down) mergesorting N keys.

A. Maximum function-call stack size when quicksorting N keys.

B. Number of compares to binary search in a sorted array of size N.

A. $\sim \lg N$ in the best case.

B. $\sim \lg N$ in the worst case.

C. Both A and B.

D. Neither A nor B.

(c) 2 minutes.
The order of growth of the running time is $N^2 \log N$ from the $\sim \frac{1}{2}N^2$ calls to binary search. Thus, if the problem size increases by a factor of 10, the running time will increase by a bit more than a factor of 10^2.

(d) 40 bytes (*using 32-bit cost model from Intro to Programming*).
(8 bytes of object overhead, 4 bytes for the `int`, and 4 bytes for each of the 7 references)

88 bytes (*using 64-bit cost model from Algorithms, 4th edition*).
(16 bytes of object overhead, 8 bytes of inner class overhead, 4 bytes for the `int`, and 8 bytes for each of the 7 references, 4 bytes of padding)

2. **8 sorting algorithms.**

0 5 7 6 4 3 9 2 8 1

3. **Binary heaps.**

(a)

0	1	2	3	4	5	6	7	8	9	10	11	12
-	Y	W	M	G	U	K	C	A	F	H	P	-

(b)

0	1	2	3	4	5	6	7	8	9	10	11	12
-	W	U	M	G	P	K	C	A	F	H	-	-

(c)

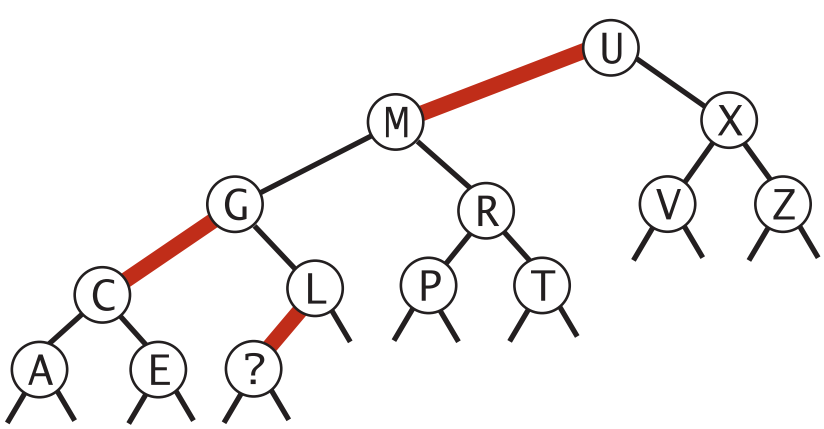
0	1	2	3	4	5	6	7	8	9	10	11	12
-	Y	W	Q	G	U	M	C	A	F	H	P	K

4. **Red-black BSTs.**

(a) H, I, J, and K

(b) RED

(c)



5. **Hashing.**

0	1	2	3	4	5	6
G	B	D	A	C	E	F

6. **Bitonic max.**

(a) We use a binary-search like algorithm, where we compare the middle key to the neighboring key to its right. Depending on the result of the comparison, we recur in the left subarray or the right subarray (which is also bitonic).

```
// find the max in the bitonic subarray a[lo] to a[hi]
public static int max(int[] a, int lo, int hi) {
   if (hi == lo) return a[hi];
   int mid = lo + (hi - lo) / 2;
   if      (a[mid] < a[mid+1]) return max(a, mid+1, hi);
   else if (a[mid] > a[mid+1]) return max(a, lo, mid);
   else                        return a[mid];
}
```

We maintain the invariant that the subarray contains the maximum key. At each step, the size of the subarray decreases by a factor of 2, so the number of compares is logarithmic in N.

(b) Here are the first four compares when finding the maximum of the following bitonic array:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a[i]	10	34	56	76	87	80	70	66	56	30	28	25	20	15	11

1. compare `a[7]` to `a[8]` (66 to 56)
2. compare `a[3]` to `a[4]` (76 to 87)
3. compare `a[5]` to `a[6]` (80 to 70)
4. compare `a[4]` to `a[5]` (87 to 80)

7. **Stable priority queue.**

   Binary heap solution. The basic idea is to associate the integer timestamp i with the i^{th} key that is inserted into the priority queue. To compare two keys, compare their keys as usual; if they are equal, break ties according to the associated timestamp. This ensures that if there are two equal keys, the one that was inserted first is considered smaller than the one that was inserted last. Here are two ways to implement this approach in Java.

   - *Implementation 1.* We modify our standard `MinPQ` implementation as follows:
     - Associate the integer timestamp i with the i^{th} key that is inserted by creating a nested class `StableKey`.

```
private class StableKey {
    private Key key;
    private long timestamp;

    public int compareTo(StableKey that) {
        int cmp = this.key.compareTo(that.key);
        if (cmp < 0) return -1;
        if (cmp > 0) return +1;
        return this.timestamp - that.timestamp;
    }
}
```

     - When comparing two keys in `less()`, break ties according to the `timestamp` field using the `compareTo()` method above.

   - *Implementation 2.* We modify our standard `MinPQ` implementation as follows:
     - Associate the integer timestamp i with the i^{th} key that is inserted by adding a parallel array `long[] timestamp` as an instance variable of `StableMinPQ`.
     - Modify `exch()` so that whenever it exchanges `pq[i]` with `pq[j]`, it also exchanges `timestamp[i]` with `timestamp[j]`.
     - Modify `less()` so that it breaks ties according to `timestamp[]`.

   Binary search tree solution. An alternate solution is to use a red-black BST. Some care is needed because our implementation of `RedBlackBST` does not support duplicate keys without modification. To handle duplicate keys, we declare a `RedBlackBST<Key, Queue<Key>>`, where the value is a queue of all the keys equal to the key.

   - To insert a key, we check if there is a key equal to it already in the BST. If there is, we add the key to the corresponding queue; if there is not, we add the key to the BST first.
   - To delete the minimum key in the stable priority queue, we call `min()` to find the minimum, and return the first element of corresponding queue. We don't invoke the `delMin()` method of `RedBlackBST` unless the queue becomes empty.

Wrong solution. A tempting idea is to swap equal keys during `sink()` but not to swap equal keys during `swim()`. However, the following sequence of operations discredits this approach:

- *insert* C_1
- *insert* B_1
- *insert* A
- *insert* B_2
- *insert* C_2
- *delmin* (returns C_1)
- *delmin* (returns C_2)
- *min* (returns B_2 instead of B_1)