COS 226 Algorithms and Data Structures Spring 2008

Midterm Solutions

1. **8 sorting algorithms.**

   8 7 4 9 5 6 3 2

2. **More sorting.**

   (a) Heapsort.

   (b) Quicksort partitioning (but instead of using the rightmost element, use the value 1 million, swapping all keys strictly greater than the value to the left).

3. **Hard problem identification.**

   E You don't need any compares. Make the median element (`a[N/2]`) the root, and apply this idea recursively.

   I If you could do this, then an inorder traversal (which requires no extra compares) would yield the elements in sorted order. This would violate the sorting $N \lg N$ lower bound.

   E This is achieved in the first phase of heapsort. See Sedgewick Property 9.4.

   I Any binary tree on N nodes has height at least $\lg N$.

   I If you had such a priority queue, you could insert N keys and delete-max them to get the keys in sorted order in $O(N \lg \lg N)$ time. This would violate the sorting $N \lg N$ lower bound.

4. **Priority queues.**

   (a) Prints the k smallest values in descending order.

   (b) $N \log k$. The maximum size of the heap is k.

   (c) Prints the k largest values in descending order.

   (d) $N \log N$. The maximum size of the heap is N.

5. **Binary heaps.**

(a)

0	1	2	3	4	5	6	7	8	9	10	11	12	13
-	*Y*	W	*V*	T	G	K	*A*	R	S	F	*-*	-	-

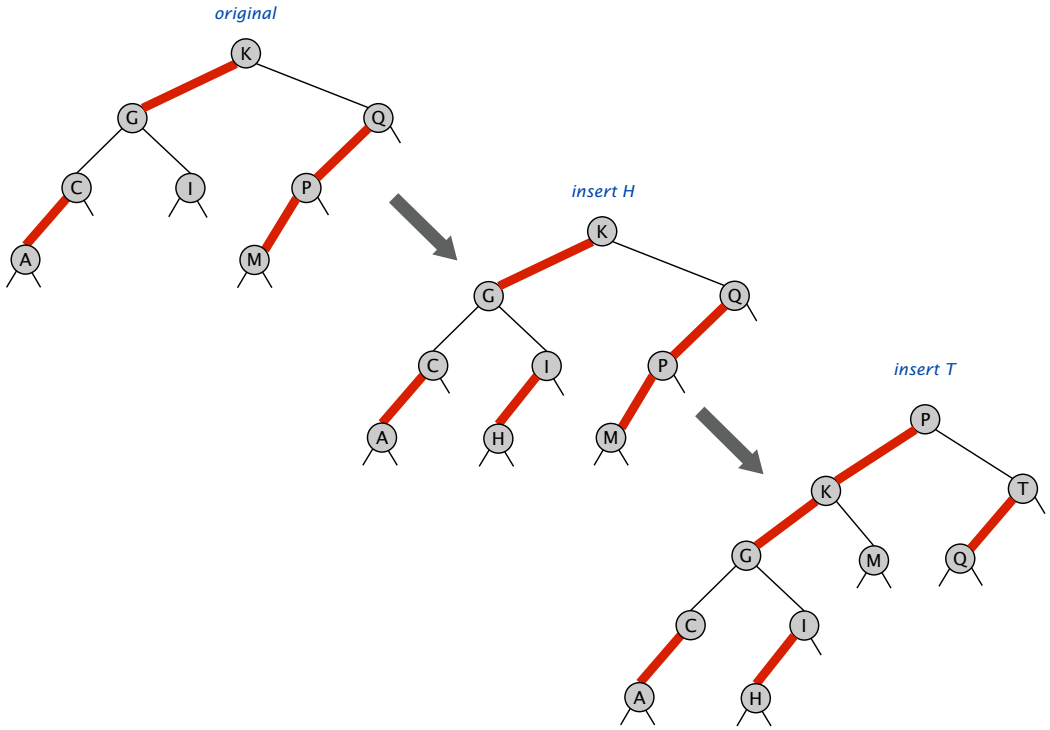
(b)

0	1	2	3	4	5	6	7	8	9	10	11	12	13
-	Z	W	Y	T	G	*X*	V	R	S	F	A	*K*	-

6. **Left-leaning red-black trees.**

(a) Only (i) is legal. The following are illegal: (ii) because the path from the root to the leftmost leaf has fewer blank links than the path from the root to the rightmost leaf, (iii) because the keys are not in symmetric order, (iv) because there is a right-leaning red link, (v) because there are 3 red links in a row, and (vi) because the path from the root to the leftmost leaf has fewer black links than the path from the root to the leaves hanging off 7.

(b)



7. **Binary search trees.**

   (a) Ceiling.

   (b) $\log N$. As usual, the number of compares is bounded by the height of the tree.

8. **Randomized queue.**

   When you add an item to a queue with N items, choose a random integer r between 0 and N and associate the new item with index r (and associate the item previously at index r with index N). To delete, always return the item associated with the index $N-1$ so that the keys are contiguous integers.

```
public class RandomizedQueue<Item> {
   private RedBlackBST<Integer, Item> st = new RedBlackBST<Integer, Item>();

   // add the item to the randomized queue
   public void enqueue(Item item) {
      int N = st.size();
      int r = StdRandom.uniform(N+1);
      st.put(N, st.get(r));
      st.put(r, item);
   }

   // delete and return a random item from the queue
   public Item dequeue() {
      int N = st.size();
      if (N == 0) throw new RuntimeException("Randomized queue underflow");
      Item item = st.get(N-1);
      st.delete(N-1);
      return item;

   }

}
```

Here is an alternate approach. When you add an item to a queue with N items, associate the new item with index N. To delete, pick a random index r between 0 and $N-1$ and return the associated item. Change the association of the item with index $N-1$ to r.

```
public class RandomizedQueue<Item> {
   private RedBlackBST<Integer, Item> st = new RedBlackBST<Integer, Item>();

   // add the item to the randomized queue
   public void enqueue(Item item) {
      int N = st.size();
      st.put(N, item);
   }

   // delete and return a random item from the queue
   public Item dequeue() {
      int N = st.size();
      if (N == 0) throw new RuntimeException("Randomized queue underflow");
      int r = StdRandom.uniform(N);
      Item item = st.get(r);
      st.put(r, st.get(N-1));
      st.delete(N-1);
      return item;

   }

}
```