

Each team can solve as many problems as possible. We are using flip to collect solutions. Just highlight the question (eg: Dynamic Median) and then write the solution as a comment. Team members cannot edit each other's comments in FLIP (that would have been nice) and so agree on a solution as a team and then enter into FLIP by one member of the team. We will rank the teams based on the correct solutions.

1. **Dynamic median.** Design a data type that supports insert in logarithmic time, find-the-median in constant time, and remove-the-median in logarithmic time. You may use any ADTs we've discussed in class. This problem is harder than any you'll find on a 226 exam.
2. **Generalized Queue.** Design a data structure that supports the following API for a generalized queue in logarithmic worst-case time. You may assume that we have access to a PQ, ST, and SET that take worst-case logarithmic time for all operations.

```
public class GQ<Item item> {
    public GQ();
    public Item get(int i);
    public void addFirst(Item item);
    public void addLast(Item item);
    public Item remove(int i);
}
```

3. **Logarithmic Randomized Queue.** For Assignment 2, you implemented a randomized queue that supported enqueue and dequeue (delete and return random) in amortized constant time (using space proportional to the number of items on the queue).

Next week, we'll discuss binary search trees that use clever tricks to maintain logarithmic height even in the worst case. Use such a balanced search tree to implement the two operations in logarithmic time per operation in the worst-case (using space proportional to the number of items on the queue).

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

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

        // YOUR CODE HERE
    }
}
```

```

// 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");
    // YOUR CODE HERE

}
}

```

4. **Heapification.** What is the run-time required to perform a bottom-up sink-based heapification? What is the run-time required for a top-down swim-based heapification? Provide a mathematical justification for each.

5. **Balanced binary trees.** Given an array of integers, provide an algorithm for building a balanced binary tree of those integers. **Extra:** Prove that it is impossible to do this in linear time.

6. **BST Sort.** Suppose we use a version of partitioning that is stable, e.g. if we start with the array [6 4 8 3 7 1 2 5], then partitioning on the leftmost item yields [4 3 1 2 5 6 8 7]. Write out the list of compares performed to fully sort this array. Write out the list of compares used to build a BST if keys are inserted in the order [6 4 8 3 7 1 2 5]. What do you observe?