PRINCETON UNIVERSITY

COS 226–Algorithms and Data Structures
Midterm Design Questions Practice

Version: March 6, 2015

1 Part I: Creating a Meta-Type

Exercise 1 – Extrinsic Max Priority Queue

An `ExtrinsicMaxPQ` is a priority queue that allows the programmer to specify the priority of an object independent of the intrinsic properties of that object. This is unlike the `MaxPQ` of class, which assumed the items are comparable.

```
public class MaxPQ<ItemType extends Comparable<ItemType>> {
    void insert(ItemType x)
    ItemType delMax()
}
```

The `ExtrinsicMaxPQ` is defined as follows:

```
public class ExtrinsicMaxPQ<ItemType> {
    ExtrinsicMaxPQ()
    void insert(ItemType x, int priority)
    ItemType delMax()
}
```

Here is a sample execution:

```
// Suppose pq is a properly initialized instance of ExtrinsicMaxPQ
pq.insert("last", 10);
pq.insert("first", 134);
pq.insert("middle", 14);
pq.insert("middle", 92);
pq.insert("other␣middle", 90);

pq.delMax();     // returns "first" (which largest priority, 134)
pq.delMax();     // returns "middle" (priority: 92)
pq.delMax();     // returns "other middle" (priority: 90)
pq.delMax();     // returns "middle"
pq.delMax();     // returns "last"
```

We consider that the `ExtrinsicMaxPQ` may contain duplicate elements.

A. Define the `ExtrinsicMaxPQ` type, and the order of growth of the run time of the `insert` and `delMax` methods.

B. Suppose we decide that `insert(item, priority)` changes the priority of `item` if it is already contained in the extrinsic priority queue (instead of adding a new duplicate element). What would you need to make this change?

Exercise 2 – Rank Queries in Symbol Tables

In lectures, we introduced the BST data type (and later its improvements as balanced BST data types), of which here is part of API:

```
public class BST<KeyType extends Comparable<KeyType>, ValueType> {
    private Node root;                 // root of BST

    private class Node {
        private KeyType key;           // sorted by key
        private ValueType val;         // associated data
        private Node left, right;      // left and right subtrees

        public Node(KeyType key, ValueType val) { this.key = key; this.val = val; }
    }

    public ValueType get(KeyType key) { }
    public void put(KeyType key, ValueType val) { }
    public void delete(KeyType key) { }

    public int rank(KeyType key) { }   // new operation
}
```

We are concerned with implemented the method `rank(key)` which returns the *rank* of `key` (its position in sorted order). For instance, if the BST contains the keys "a", "b", "e", "g", then the rank of "e" is 3.

A. Suppose run time does need to be optimized, how would you implement the `rank` function?

B. Is it possible to make adjustments to the data structure to obtain a constant run time, while not degrading the *order of growth of* the run time of the other operations by more? If so, explain how; if not, explain why.

C. What other data structures (besides a BST) would be suitable for this problem (insert/delete key-value pairs, and retrieve rank of keys)?

Exercise 3 – Stable Priority Queue (Midterm Fall 2010)

A min-based priority queue is *stable* if `min()` and `delMin()` return the minimum key that was least-recently inserted. Describe how to implement a `StableMinPQ` data type such that every operation takes at most (amortized) logarithmic time.

```
public class StableMinPQ<Item exends <Comparable<Item>>
{
  StableMinPQ()           // creates an empty priority queue

  void insert(Item item)  // add an item to the priority queue
  Item min()              // return the least recently inserted minimum
  Item delMin()           // delete the minimum (that was least recently
                          //   inserted) from the PQ and return it

  int size()              // number of elements in the PQ
  boolean isEmpty()       // returns true if the size is zero
}
```

2 Part II: General Practice

Exercise 4 – Min-Max Priority Queue

Standard priority queues come in two flavors: either a minimum priority queue (in which elements can be removed in descending order) or a maximum priority queue (in ascending order); and generally all operations can be implemented in logarithmic time.

Here, we would like to design a priority queue that allows for deletion, at any given time, *either* of the current minimum or the current maximum. In other words, the API should support the following operations:

```
public class MinMaxQueue<ItemType extends Comparable<ItemType>>
{
  MinMaxQueue()               // creates an empty priority queue

  void insert(ItemType item)  // add an item to the priority queue
  Item delMin()               // delete the minimum from the PQ and return it
  Item delMax()               // delete the maximum from the PQ and return it

  int size()                  // number of elements in the PQ
  boolean isEmpty()           // returns true if the size is zero
}
```

Design an implementation in which all operations are supported in logarithmic worst-case time.

Exercise 5 – Bitonic Max (Midterm Fall 2010)

An array is *bitonic* if it consists of a strictly increasing sequence of keys immediately followed by a strictly decreasing sequence of keys. Design an algorithm that determines the maximum key in a bitonic array of size N in time proportional to $\log N$.

Exercise 6 – Comparing Two Arrays of Points (Midterm Fall 2011)

Given two arrays `a[]` and `b[]`, each containing N distinct points in the plane, design two algorithms (with the performance requirements specified below) to determine whether the two arrays contains precisely the same set of points (but possibly in a different order).

A. Design an algorithm for the problem whose running time is linearithmic in the *worst case* and uses at most constant extra space.

B. Design an algorithm for the problem whose running time is linear under reasonable assumptions and uses at most linear extra space. Be sure to state any assumptions that you make.

Exercise 7 – Stabbing Count Queries (Midterm Fall 2011)

Given a collection of x-intervals and a real value x, a *stabbing count query* is the number of intervals that contain x. Design a data structure that supports interval insertions intermixed with stabbing count queries by implementing the following API:

```
public class IntervalStab
{
  IntervalStab()                          // create empty data structure

  void insert(double xmin, double xmax) // insert the interval (xmin, xmax)
                                          //    into the data structure

  int count(double x)                     // number of intervals that contain x
}
```

For example, after inserting the five intervals $(3, 10)$, $(4, 5)$, $(6, 12)$, $(8, 15)$, and $(19, 30)$ into the data structure, `count(9.1)` is 3 and `count(17.2)` is 0.

If there are N intervals in the data structure, you should support *insert* and *count* in time proportional to $\log N$ in the worst case (even if `count()` returns N). For simplicity, you may assume that no two intervals contain a left or right endpoint in common and that the argument to the stabbing count query is not equal to a left or right endpoint.

Exercise 8 – Duplicate Detection in Multiple Arrays (Midterm Fall 2012)

We have already seen in precept ways to detect duplicates in a single array. Here we are given k sorted arrays containing N keys in total, design an algorithm that determine whether there is any key that appears more than once.

Your algorithm should use extra space at most proportional to k. The best solution should run in time at most proportional to $N \log k$ in the worst case; a partial solution exists that runs in time proportional to Nk.

Exercise 9 – Randomized Priority Queue (Midterm Spring 2012)

In one of the assignments, you designed a randomized queue. Here, we are looking to describe how to extend the binary heap implementation of the `MinPQ` API, with the methods `sample()` and `delRandom()`. The two methods return a key that is chosen uniformly at random among the remaining keys, with the latter method also removing that key.

```
public class MinPQ<Item>
{
  MinMaxQueue()           // creates an empty priority queue

  void insert(Item item)  // add an item to the priority queue
  Item min()              // return the minimum from the PQ
  Item delMin()           // delete the minimum from the PQ and return it

  Item sample()           // pick a uniformly random key from the PQ
  Item delRandom()        // return and remove a uniformly random key from the PQ

  int size()              // number of elements in the PQ
  boolean isEmpty()       // returns true if the size is zero
}
```

You should implement the `sample()` method in constant time and the `delRandom()` method in time proportional to $\log N$, where N is the number of keys in the data structure. For simplicity, do not worry about resizing the underlying array.

Exercise 10 – LRU Cache (Midterm Fall 2012)

An *LRU cache* is a data structures that stores up to *N distinct* keys. If the data structure is full when a key not already in the cache is added, the LRU cache first removes the key that was *least recently cached*. Design a data structure with the following API:

```
public class LRU<Item> {

  public LRU(int N)           // create an empty LRU cache with capacity N

  void cache(Item item)       // if there are N keys in the cache and the given
                              //   key is not already in the cache, (i) remove
                              //   the key that was least recently used as an
                              //   argument to cache and (ii) add the given key
                              //   to the LRU cache

  boolean inCache(Item item)  //
}
```

The operations `cache()` and `inCache()` should take constant time on average under the uniform hashing assumption.

For example,

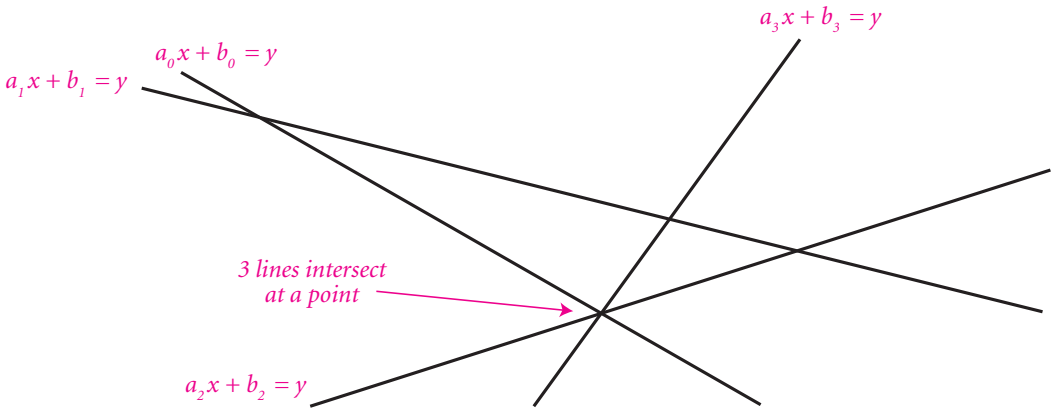
```
LRU<String> lru = new LRU<String>(5);

                                 // LRU cache   (in order of when last cached)
lru.cache("A");                  // A           (add A to front)
lru.cache("B");                  // B A         (add B to front)
lru.cache("C");                  // C B A       (add C to front)
lru.cache("D");                  // D C B A     (add D to front)
lru.cache("E");                  // E D C B A   (add E to front)
lru.cache("F");                  // F E D C B   (remove A from back; add F to front)
boolean b1 = lru.inCache("C");   // F E D C B   (true)
boolean b2 = lru.inCache("A");   // F E D C B   (false)
lru.cache("D");                  // D F E C B   (move D to front)
lru.cache("C");                  // C D F E B   (move C to front)
lru.cache("G");                  // G C D F E   (remove B from back; add G to front)
lru.cache("H");                  // H G C D F   (remove E from back; add H to front)
boolean b3 = lru.inCache("D");   // H G C D F   (true)
```

Describe the data structure(s) that you use. For example, if you use a linear probing hash table, specify what are the hash table key-value pairs.

Exercise 11 – Line Intersection (Midterm Fall 2008)

Given N lines in the plane, design an algorithm to determine if any 3 (or more) of them intersect at a single point.



For simplicity, assume that all of the lines are distinct and that there are no vertical lines.

A. We specify a line by two numbers a and b such that a point (x, y) is on the line if and only if $ax + b = y$. Suppose that you are given two lines $a_0x + b_0 = y$ and $a_1x + b_1 = y$. Design a constant-time algorithm that determines if the two lines intersect and, if so, finds the point of intersection. (*Assume that you can perform arithmeti operations with arbitrary precision in constant time. That is, don't worry about floating-point precision.*)

B. Given an array of N lines, design an $O(N^2)$ algorithm to determine if any 3 (or more) of them intersect at single point. A partial solution would yield an $O(N^2 \log N)$ algorithm.

C. How would you modify your previous algorithm if you needed to output all sets of 3 (or more) lines that intersect at a single point, with the condition that each set be output exactly once?