COS 226–Algorithms and Data Structures
Practice Design Questions

Version: December 13, 2016

Exercise 1 – MST Edge (Fall 2011)

Given an edge-weighted graph G and an edge e, design a linear-time algorithm to determine whether e appears in an MST of G. For simplicity, assume that G is connected and that all edge weights are distinct. Note that since your algorithm must take linear time, in the worst case, you cannot afford to compute the MST itself.

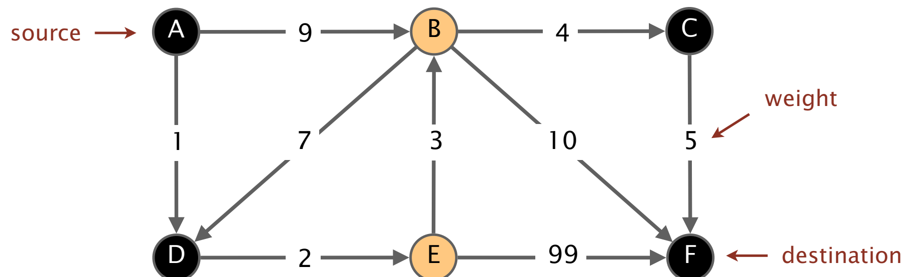
Exercise 2 – Shortest-Princeton-Path (Final Spring 2011)

Consider the following two graph-processing problems:

- Shortest-Path. Given an edge-weighted digraph G with nonnegative edge weights, a source vertex s, and a destination vertex t, find a shortest path from s to t.

- Shortest-Princeton-Path. Given an edge-weighted digraph G with nonnegative edge weights, a source vertex s, a destination vertex t, and *with each vertex colored black or orange*, find a shortest path from s to t that uses *at most one orange vertex*. Assume that the source vertex is not orange.

In the edge-weighted digraph below, the shortest path from A to F is $A \to D \to E \to B \to C \to F$ (weight 15) but the the shortest Princeton path is $A \to B \to C \to F$ (weight 18).



1. Give a linear-time reduction from Shortest-Path to Shortest-Princeton-Path. To demonstrate your reduction, draw the edge-weighted digraph (labeling the source and destination vertices and coloring each vertex black or orange) that you would construct to solve the Shortest-Path instance above.

©Fall 2016 by Princeton/Dept. of Comp. Sci.

2. Give a linear-time reduction from Shortest-Princeton-Path to Shortest-Path. To demonstrate your reduction, draw the edge-weighted digraph (labeling the source and destination vertices) that you would construct to solve the Shortest-Princeton-Path instance on the facing page.

Exercise 3 – 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 4 – 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 5 – 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
}
```

Exercise 6 – 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 7 – 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 8 – 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 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.