CS 466/666, Spring 2009
Timothy Chan

Assignment 2 (due June 10 Wednesday 5pm)

Please read `http://www.student.cs.uwaterloo.ca/~cs466/policies.html` first for general instructions.

1. [*15 marks*] In this question, we investigate a special case of the problem of maintaining the minimum of a set S of numbers. (This special case arises, for example, in running Dijkstra's shortest path algorithm on a graph with small integer weights.) Specifically, we want a data structure to support the following operations:

   - insert-special(x): insert an element x to S where x is an integer in the range $\{1, \ldots, U\}$ and x is greater than the current minimum;
   - decrease-key-special(x, k): decrease x's value to k where k is an integer in the range $\{1, \ldots, U\}$ and k is greater than the current minimum;
   - delete-min(): return the minimum from S and remove this element.

   Note that because of the assumptions in insert-special() and decrease-key-special(), we know that the current minimum can only increase over time.

   (a) [*4 marks*] Give a data structure that supports insert-special() in $O(U/n)$ amortized time, decrease-key-special() in $O(1)$ amortized time, and delete-min() in $O(1)$ amortized time, where n denotes the number of insert operations.
   [Note: this method is thus superior to the methods from class when U is linear in n. Hint: just use an array of size U... You may assume that n and U are known in advance, $U \geq n$, and that all keys are distinct.]

   (b) [*11 marks*] Give a still better data structure that supports insert-special() in $O(\log(U/n))$ amortized time, decrease-key-special() in $O(1)$ amortized time, and delete-min() in $O(1)$ amortized time.
   [Hint: let $d = \lceil U/n \rceil$. Use an array of n doubly linked, unsorted lists $L_1, \ldots, L_n$, where each list L_i stores elements in the range $\{id + 1, \ldots, (i+1)d\}$. Suppose the current minimum lies in L_{i^*}. Store the elements in L_{i^*} in a Fibonacci heap H.]

2. [*15 marks*] Give a data structure to support the following operations on a collection of disjoint point sets in 2D:

   - initialize(P): create a new set P containing n points.
   - count(P): return the number of points in P.
   - x-partition(P, x_0): create two new sets $P_L = \{(x, y) \in P \mid x \leq x_0\}$ and $P_R = \{(x, y) \in P \mid x > x_0\}$ and return (the labels of) P_L and P_R; the old set P is no longer in the collection.

- y-partition(P, y_0): create two new sets $P_B = \{(x, y) \in P \mid y \leq y_0\}$ and $P_T = \{(x, y) \in P \mid y > y_0\}$ and return (the labels of) P_B and P_T; the old set P is no longer in the collection.

Your solution should take $O(n \log n)$ amortized time for initialize(), $O(1)$ time for count(), and $O(\log n)$ amortized time for x-partition() and y-partition().

[Hint: For each point set, simply maintain the points in two doubly linked lists, one sorted in x-coordinates, the other sorted in y-coordinates. In partition, remove points from the smaller side... For the analysis, don't use potentials; use an argument similar to the one from class for the weighted union heuristic.]

3. [*10 marks*] Give a data structure to support the following operations on a set S of intervals in one dimension:

   - insert(a, b): given two real numbers a and b with $a < b$, insert the interval $[a, b]$ to S.
   - query(x): given a real number x, return yes iff x lies in the union of the intervals in S.

   (For example, if S contains the intervals $[1, 4]$, $[6, 10]$, $[8, 13]$ and $x = 9$, the union of S is $[1, 4] \cup [6, 13]$ and query(x) should return yes.)

   Your solution should have $O(\log n)$ amortized insertion time and query time.

   [Hint: Recall that standard balanced search trees can support insertions and deletions to a set T of numbers in $O(\log n)$ time and can find the predecessor of any value x (not necessarily in the set T) in $O(\log n)$ time. The *predecessor* of x is the largest value in T smaller than x.]

4. [*10 marks*] In a popular form of logic puzzles, you land on an imaginary island where inhabitants are classified into three types: "knights", who always tell the truth; "knaves", who always lie; and "spies", who sometimes lie and sometimes tell the truth.

   Suppose there are n inhabitants, where 60% are known to be decent folks, i.e., knights. The remaining 40% are bad, i.e., knaves or spies. You want to know who the good/bad guys are, i.e., you want to determine the types of all n inhabitants. You are allowed to ask only questions of the form, "is person A a knight/knave/spy?", to another person B. (All the inhabitants know each other.) Obviously, if you can find a person who you know is a knight, the problem is solved after asking n additional questions.

   (a) [*2 marks*] Give a (very) efficient Monte-Carlo algorithm that finds a knight. State the probability of error. [Hint: this is supposed to be easy!]

   (b) [*8 marks*] Give a Las-Vegas algorithm that finds a knight by asking $O(n)$ expected number of questions. Analyze the constant factor in the big-Oh and make it smaller than 1.5. [Hint: use (a). How can you confirm whether a specific person is a knight by asking $O(n)$ questions?]

   [Note: there is also a deterministic algorithm that requires $O(n)$ questions, but it's more complicated and has a larger constant.]