Problem Set 5

MIT students: This problem set is due in lecture on **Monday, October 31, 2005.** The homework lab for this problem set will be held 2–4 P.M. on Sunday, October 30, 2005.

Reading: Chapter 14 and Skip List Handout.

Both exercises and problems should be solved, but *only the problems* should be turned in. Exercises are intended to help you master the course material. Even though you should not turn in the exercise solutions, you are responsible for material covered in the exercises.

Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date and the names of any students with whom you collaborated. **Please staple and turn in your solutions on 3-hole punched paper**.

You will often be called upon to "give an algorithm" to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of the essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudo-code.
2. At least one worked example or diagram to show more precisely how your algorithm works.
3. A proof (or indication) of the correctness of the algorithm.
4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Full credit will be given only to correct solutions *which are described clearly*. Convoluted and obtuse descriptions will receive low marks.

Exercise 5-1. Do Exercise 14.1-5 on page 307 of CLRS.

Exercise 5-2. Do Exercise 14.2-1 on page 310 of CLRS.

Exercise 5-3. Do Exercise 14.3-4 on page 317 of CLRS.

Exercise 5-4. Do Problem 14.2 on page 318 of CLRS.

Problem 5-1. Skip Lists and B-trees

Intuitively, it is easier to find an element that is nearby an element you've already seen. In a dynamic-set data structure, a ***finger search from x to y*** is the following query: given the node in the data structure that stores the element x, and given another element y, find the node in the data structure that stores y. Skip lists support fast finger searches in the following sense.

(a) Give an algorithm for finger searching from x to y in a skip list. Your algorithm should run in $O(\lg(2+|\mathit{rank}(x)-\mathit{rank}(y)|))$ time with high probability, where $\mathit{rank}(x)$ denotes the current rank of element x in the sorted order of the dynamic set.

When we say "with high probability" we mean high probability with respect to $m = 2 + |\mathit{rank}(x) - \mathit{rank}(y)|$. That is, your algorithm should run in $O(\lg m)$ time with probability $1 - 1/m^\alpha$, for any $\alpha \geq 1$.

Assume that the finger-search operation is given the node in the bottommost list of the skip list that stores the element x.

To support fast finger searches in B-trees, we need two ideas: B$^+$-trees and level linking. Throughout this problem, assume that $B = O(1)$.

A ***B$^+$-tree*** is a B-tree in which all the keys are stored in the leaves, and internal nodes store copies of these keys. More precisely, an internal node p with $k+1$ children $c_1, c_2, \ldots, c_{k+1}$ stores k keys: the maximum key in c_1's subtree, the maximum key in c_2's subtree, ..., the maximum key in c_k's subtree.

(b) Describe how to modify the B-tree SEARCH algorithm in order to find the leaf containing a given key x in a B$^+$-tree in $O(\lg n)$ time.

(c) Describe how to modify the B-tree INSERT and DELETE algorithms to work for B$^+$-trees in $O(\lg n)$ time.

A ***level-linked B$^+$-tree*** is a B$^+$-tree in which each node has an additional pointer to the node immediately to its left among nodes at the same depth, as well as an additional pointer to the node immediately to its right among nodes at the same depth.

(d) Describe how your B$^+$-tree INSERT and DELETE algorithms from part (c) can be modified to maintain level links in $O(\log n)$ time per operation.

(e) Give an algorithm for finger searching from x to y in a level-linked B$^+$-tree. Your algorithm should run in $O(\lg(2 + |\mathit{rank}(x) - \mathit{rank}(y)|))$ time.

These ideas suggest a connection between skip lists and level-linked 2-3-4 trees. In fact, a skip list is essentially a randomized version of level-linked B$^+$-tree.

(f) Describe how to implement a ***deterministic skip list***. That is, your data structure should have the same general pointer structure as a skip list: a sequence of one or

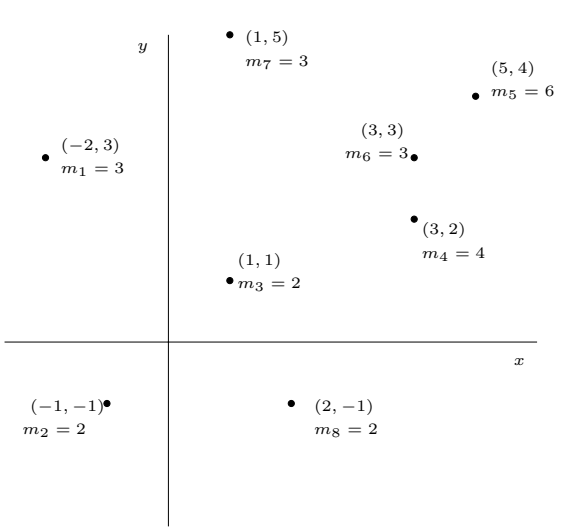

Figure 1: In this example of eight points, if $f(p_i) = m_i$, then $F(S) = 25$.

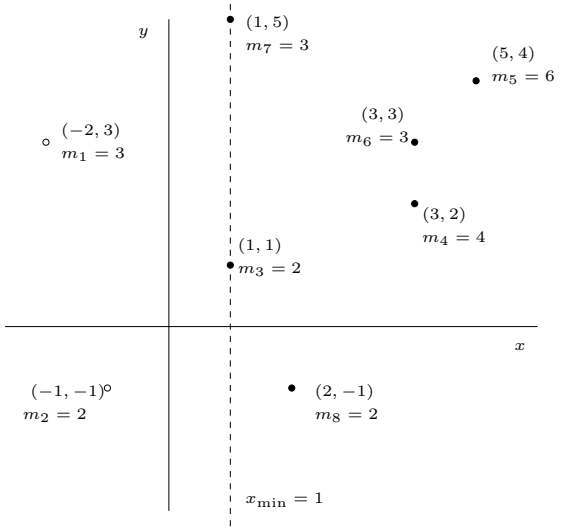

Figure 2: When $x_{\min} = 1$, $T(1) = \{p_3, p_4, p_5, p_6, p_7, p_8\}$ and $F(T(1)) = 20$.

more linked lists with pointers between nodes in adjacent lists that store the same key. The SEARCH algorithm should be identical to that of a skip list. You will need to modify the INSERT operation to avoid the use of randomization to determine whether a key should be promoted. You may ignore DELETE for this problem part.

Problem 5-2. Fun with Points in the Plane

It is 3 a.m. and you are attempting to watch 6.046 lectures on video, looking for hints for Problem Set 5. For some odd reason, possibly because you are fading in and out of consciousness, you start to notice a strange cloud of black dots on an otherwise white wall in your room. Thus, instead of watching the lecture, your subconscious mind starts trying to solve the following problem.

Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of n points in the xy plane. Each point p_i has coordinates (x_i, y_i) and has a ***weight*** m_i (a real number representing the size of the dot). Let $f(p) = f(x, y, m)$ be an arbitrary function mapping a point p with coordinates (x, y) and weight m to a real number, computable in $O(1)$ time. For a subset T of S, define the function $F(T)$ to be the sum of $f(p_i)$ over all points in T, i.e.,

$$F(T) = \sum_{p \in T} f(p).$$

For example, if $f(p_i) = m_i$, then $F(S)$ is the sum of the weights of all n points. This case is depicted in Figure 1.

Our goal is to compute the function F for certain subsets of the points. We call each subset a *query*, and for each query T, we want to calculate $F(T)$. Because there may be a large number of queries, we want to design a data structure that will allow us to efficiently answer each query.

First we consider queries that restrict the x coordinate. In particular, consider the set of points whose x coordinates are at least $x_{\min}$. Formally, let $T(x_{\min})$ be the set of points

$$T(x_{\min}) = \{p_i \in S \mid x_i \geq x_{\min}\}.$$

We want to answer queries of the following form: given any value $x_{\min}$ as input, calculate the value $F(T(x_{\min}))$. Figure 2 is an example of such a query. In this case, $x_{\min} = 1$, and the points of interest are those with x coordinate at least 1.

(a) Show how to modify a balanced binary search tree to support such a query in $O(\lg n)$ time. More specifically, the computation of $F(T(x_{\min}))$ can be performed using only a single walk down the tree. You do not need to support updates (insertions and deletions) for this problem part.

(b) Consider the static problem, where all n points are known ahead of time. How long does it take to build your data structure from part (a)?

(c) In total, given n points, how long does it take to build your data structure and answer k different queries? On the other hand, how long would it take to answer k different queries without using any data structure and using the naïve algorithm of computing $F(T(x_{\min}))$ from scratch for every query? For what values of k is it asymptotically more efficient to use the data structure?

(d) We can make this data structure dynamic by using a red-black tree. Argue that the augmentation for your solution in part (a) can be efficiently supported in a red-black tree, i.e., that points can be inserted or deleted in $O(\lg n)$ time.

Next we consider queries that take an interval $X = [x_{\min}, x_{\max}]$ (with $x_{\min} \leq x_{\max}$) as input instead of a single number $x_{\min}$. Let $T(X)$ be the set of points whose x coordinates fall in that interval, i.e.,

$$T(X) = \{p_i \mid x_i \in [x_{\min}, x_{\max}]\}.$$

See Figure 3 for an example of this sort of query.

We claim that we can use the same dynamic data structure from part (d) to compute $F(T(X))$.

(e) Show how to modify your algorithm from part (a) to compute $F(T(X))$ in $O(\lg n)$ time. *Hint:* Find the shallowest node in the tree whose x coordinate lies between $x_{\min}$ and $x_{\max}$.

Finally, we generalize the static problem to two dimensions. Suppose that we are given two intervals, $X = [x_{\min}, x_{\max}]$ and $Y = [y_{\min}, y_{\max}]$. Let $T(X, Y)$ be the set of all points in this rectangle, i.e.,

$$T(X, Y) = \{p_i \mid x_i \in X \text{ and } y_i \in Y\}.$$

See Figure 4 for an example of a two-dimensional query.

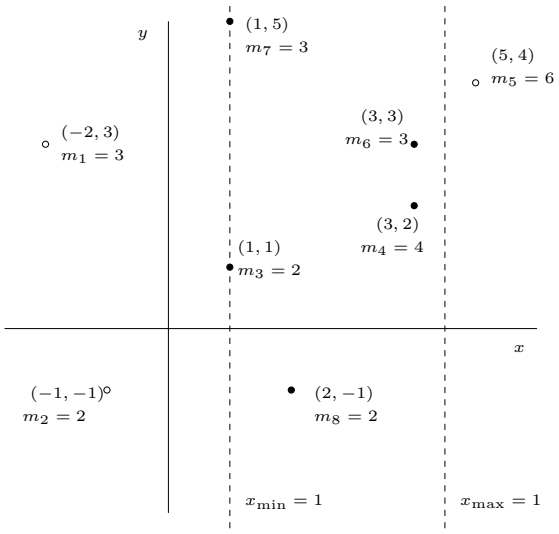

Figure 3: When $X = [1, 3.5]$, $T(X) = \{p_3, p_4, p_6, p_7, p_8\}$ and $F(T(X)) = 14$.

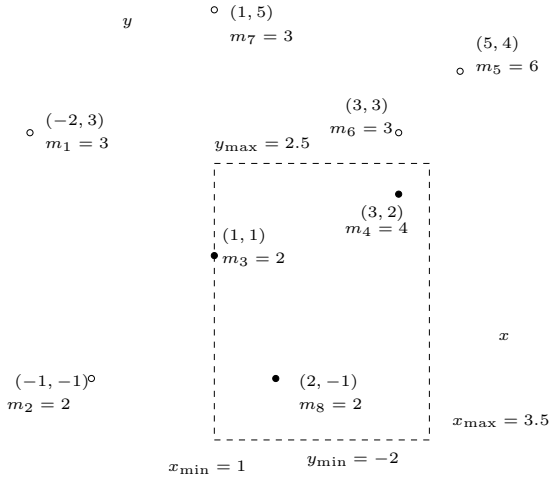

Figure 4: For $X = [1, 3.5]$ and $Y = [-2, 2.5]$, $T(X, Y) = \{p_3, p_4, p_8\}$ and $F(T(X, Y)) = 8$.

(f) Describe a data structure that efficiently supports a query to compute $F(T(X, Y))$ for arbitrary intervals X and Y. A query should run in $O(\lg^2 n)$ time. *Hint:* Augment a range tree.

(g) How long does it take to build your data structure? How much space does it use?

Unfortunately, there are problems with making this data structure dynamic.

(h) Explain whether your argument in part (d) can be generalized to the two-dimensional case. What is the worst-case time required to insert a new point into the data structure in part (f)?

(i) Suppose that, once we construct the data structure with n initial points, we will perform at most $O(\lg n)$ updates. How can we modify the data structure to support both queries and updates efficiently in this case?

Completely Optional Parts

The remainder of this problem presents an example of a function F that is useful in an actual application and that can be computed efficiently using the data structures you described in the previous parts. Parts (j) through (l) outline the derivation of the corresponding function $f(p_i)$.

The remainder of this problem is completely optional. Please do not turn these parts in!

As before, consider a set $S = \{p_1, p_2, \ldots, p_n\}$ of n points in the plane, with each point p_i having coordinates (x_i, y_i) and a weight m_i. We want to compute the axis that minimizes the moment of

inertia of the points in the set. Formally, we want to compute a line L in the plane that minimizes the quantity

$$\sum_{i=1}^{n} m_i \Big(d(L, p_i) \Big)^2,$$

where $d(L, p_i)$ is the distance from point p_i to the line L. If $m_i = 1$ for all i, we can think of this axis as the "orientation" of the set.

(j) One parameterization of a line in the xy plane is to describe it using a pair (ρ, θ), where ρ is the distance from the origin to the line and θ is the angle the line makes with the x axis. It can be shown that the distance between a point (x, y) and a line L parameterized by (ρ, θ) is

$$|x \sin\theta - y \cos\theta + \rho|.$$

We defined the orientation of the set of points S as the line $L = (\rho, \theta)$ that minimizes the function

$$f(\rho, \theta) = \sum_{i=1}^{n} m_i \left(x_i \sin\theta - y_i \cos\theta + \rho \right)^2.$$

Show that setting $\frac{\partial f}{\partial \rho} = 0$ gives us the constraint

$$M_{x1} \sin\theta - M_{y1} \cos\theta + M_0 \rho = 0,$$

where

$$M_0 = \sum_{i=1}^{n} m_i, \qquad M_{x1} = \sum_{i=1}^{n} m_i x_i, \qquad M_{y1} = \sum_{i=1}^{n} m_i y_i \quad .$$

(k) Show that setting $\frac{\partial f}{\partial \theta} = 0$ and using the constraint from part (j) leads to the equation

$$\tan 2\theta = \frac{2 \left(M_0 M_{xy} - M_{x1} M_{y1} \right)}{M_0 \left(M_{x2} - M_{y2} \right) + M_{y1}^2 - M_{x1}^2},$$

where

$$M_{xy} = \sum_{i=1}^{n} m_i x_i y_i, \qquad M_{x2} = \sum_{i=1}^{n} m_i x_i^2, \qquad M_{y2} = \sum_{i=1}^{n} m_i y_i^2 \quad .$$

(l) Give the function $f(p_i)$ that makes the orientation problem a special case of the problem we just solved. ***Hint:*** The function $f(p_i)$ is a vector-valued function.