Introduction to Algorithms November 4, 2005
Massachusetts Institute of Technology 6.046J/18.410J
Professors Erik D. Demaine and Charles E. Leiserson Handout 21

Problem Set 5 Solutions

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.

Solution:

For the purposes of this problem, we assume that each node x in the skip list has the following fields:

$\mathit{key}[x]$	the key associated with node x
$\mathit{next}[x]$	the next element in the linked list containing x
$\mathit{level}[x]$	the level of the linked list containing x
$\mathit{up}[x]$	the element in the linked list of level $\mathit{level}[x]+1$ containing the same key as x
$\mathit{down}[x]$	the element in the linked list of level $\mathit{level}[x]-1$ containing the same key as x

We present code for the case where $\mathit{key}[x] \leq k$. The case where $k \leq \mathit{key}[x]$ is symmetric. We also assume that $\mathit{level}[x] = 0$, i.e., the search starts at the lowest level of the skip list. (If the search starts at level $\lg n$, then the finger search may take $O(\lg n)$ time.)

The algorithm proceeds in two phases. In the first phase, we ascend levels as rapidly as possible, until we find a level that does not allow forward motion.

```
FINGER-SEARCH(x, k)                          ▷ Search for k in skip list containing node x.
 1  while key[next[x]] < k
 2        do  while up[x] ≠ NIL
 3                  do  x ← up[x]
 4            x ← next[x]
 5  while level[x] ≥ 0
 6        do  while key[next[x]] ≤ k
 7                  do  x ← next[x]
 8            if level[x] > 0
 9              then x ← down[x]
10  return x
```

(Notice that the search climbs higher than it needs to. A better solution is to examine the next point at each level, and only continue up if the next key is smaller than the target key. As we prove, though, the simpler algorithm above succeeds as well.)

We first establish the highest level reached during the finger search. The proof is exactly the same as the proof that a skip list has at most $O(\lg n)$ levels.

Lemma 1 *While executing* FINGER-SEARCH, *level*$[x] \leq O(\lg m)$, *with high probability, where* $m = 2 + rank(key[x]) - rank(k)$.

Proof. Notice that there are at most $m - 1$ elements in the skip list in the (closed) interval $M = [key[x], k]$. We calculate the probability that any of these $m-1$ elements exceeds height $c \lg(m)$.

$$\begin{aligned} \Pr[\text{any element in M is in more than } c\lg(m+1) \text{ levels}] &\leq (m-1) \cdot \frac{1}{2^{c\lg(m)}} \\ &\leq \frac{m}{(m)^c} \\ &\leq \frac{1}{(m)^{c-1}} \end{aligned}$$

Since none of the elements in the interval M are promoted past level $c \lg(m)$, with high probability, and *key*$[x]$ is always in the interval M, the result follows. Notice that in the case of this proof, the high probability analysis is with respect to m, not n.

We now consider the cost of the two phases of the algorithm: first the cost of ascending, and then the cost of descending. Recall the lemma proved in class:

Lemma 2 *The number of coin flips required to see* $c \lg m$ *heads is* $O(\lg m)$ *with high probability.*

Using this lemma, we can see that the first phase, ascending levels, completes within $O(\lg m)$ steps as there are only $c \lg m$ levels to ascend and each heads results in moving up a level. This lemma also shows that the second phase completes in $O(\lg m)$ steps: as in the analysis of a SEARCH, we calculate the cost in reverse, ascending

from the node containing k to the level reached during the first phase; since there are only $c \lg m$ levels to ascend, and each heads results in moving up a level, the cost of the second phase is also $O(\lg m)$. Hence the total running time is $O(\lg(m+1))$ as desired.

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.

Solution:

The only modification necessary is to always search to the leaves, rather than to return if the key is found in an internal node.

```
B+-TREE-SEARCH(x, k)
1  i ← 1
2  while i ≤ n[x] and k > key_i[x]
3        do  i ← i + 1
4  if leaf[x]
5     then if i ≤ n[x] and k = key_i[x]
6               then return (c_i[x], k)
7               else  return NIL
8     else  return B+-TREE-SEARCH(x, k)
```

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

Solution: Normally, during an insert, B-TREE-SPLIT-CHILD cuts a node into three pieces: the elements less than the median, the median, and the elements greater than the median. It puts the median in the parent, and the other parts in new nodes.

In B$^+$-trees, the split is unchanged at internal nodes, but not at leaves. The median has to be kept in one of the two leaf nodes—the left one—as well as being inserted into a parent. The following pseudocode implements the new split routine.

```
B+-TREE-SPLIT-CHILD(x, i, y)
 1  if leaf[y]
 2     then z ← ALLOCATE-NODE()
 3          leaf[z] ← FALSE
 4          n[z] ← B − 1
 5          for j ← 1 to B − 1
 6              do key_j[z] ← key_{j+B}[y]
 7                 c_j[z] ← c_{j+B}[y]
 8          c_t[z] ← c_{2B}[y]
 9          n[y] ← B                    ▷ Unlike a B-tree, the old node keeps B keys.
10          for j ← n[x] + 1 downto i + 1
11              do c_{j+1}[x] ← c_j[x]
12                 key_j[x] ← key_{j−1}[x]
13          c_{i+1}[x] ← z
14          key_i[x] ← key_B[y]
15          n[x] ← n[x] + 1
16     else return B-TREE-SPLIT-CHILD(x, i, y)
```

The remaining insert routines must be modified to use the new split routine.

A B$^+$-tree delete consists of two parts: deleting the key from the leaf, which may entail fixing up the tree, and ensuring that a copy of the key no longer appears in an internal node. Each part takes $O(\lg n)$ time.

The delete begins by descending from the root to the leaf containing the key. If the key being deleted is discovered in an internal node, it is ignored. (Remember these nodes; we will come back to these nodes later, in the second part of the delete.) As in the case of a B-tree, we want to ensure that each node along the descent path has at least B keys. Recall that a B-tree delete consisted of three cases. In particular, case (3) ensured that while descending the tree to perform a delete, a child contained at least B keys. Apply case (3) to ensure that each node along the path has at least B keys. The only modification is that when merging leaves, it is unnecessary to move a key down into the new merged node; the key can simply be removed from the parent. On finding the key in the leaf, remove it from the leaf.

In the second part of the delete, we want to ensure that a copy of the deleted key does not appear in an internal node. First, search for the predecessor to the deleted key. Recall the nodes discovered during the first part that contained the key being deleted. Wherever the deleted key appeared along the path, replace it with the predecessor. (Copies of the deleted key can only appear on the root-to-leaf path traversed during the deletion.)

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.

Solution: Each node x now contains a pointer $\mathit{link}[x]$ pointing to its right sibling. The split routine is modified to include the following two lines:

```
1  link[z] ← link[y]
2  link[y] ← z
```

This copies the link of the node being split, y to the new node z, and updates the link of y to point to z. Otherwise, the insert operation remains unchanged.

During a delete operation, when two nodes are merged, the link pointers must be appropriately updated: the link from the left node (being deleted) must be copied to the link point of the remaining merged node. Otherwise, the delete operation remains unchanged.

(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.

Solution: The algorithm is quite similar to the solution to part (a). The search proceeds up the tree until the next key is greater than the key being searched for. At this point, the search can continue down the tree in the usual fashion, using links when necessary.

As before, the code below assumes that initially $k \geq \mathit{key}_i[x]$, where (x, i) represents the pointer to a key in the tree. The opposite case is symmetric. In order to perform the finger-search efficiently, we assume that each node x has a parent pointer $p[x]$ and a parent index $pi[x]$.

```
B+-TREE-FINGER-SEARCH(x, i, k)
 1  while k > key_i[x] and link[x] ≠ NIL and k > key_1[link[x]]
 2        do while i ≤ n[x] and k > key_i[x]
 3                 do i ← i + 1
 4           if i > n[x] and k > key_1[link[x]]
 5              then x ← p[x]
 6                   i ← pi[x]
 7  ▷ Begin downwards search.
 8  while not leaf[x]
 9        do if k ≥ key_1[link[x]]
10              then x ← link[x]
11                   i ← 1
12              else x ← c_i[x]
13                   i ← 1
14           while i ≤ n[x] and k > key_i[x]
15                 do i ← i + 1
16  if k = key_i[x]
17     then return (x, i)
18     else return NIL
```

The key to showing the search efficient is proving that the first phase ascends no more than $O(\lg m)$ levels. In order to prove this fact, notice that x is only set to $p[x]$ in line 5 when $k > \mathit{key}_1[\mathit{link}[x]]$. This implies that k is larger than every leaf in the subtree rooted at $c_1[\mathit{link}[x]]$. Moreover, every leaf in the subtree rooted at $c_1[\mathit{link}[x]]$ must be larger than the initial $\mathit{key}_i[x]$ when the search began (since the search moves only in the direction of increasing keys). Hence, if x ascends from level ℓ to level $\ell+1$ (assuming that the leaves are at level 0), then $m \geq 2^{\ell-2}$, implying that $\ell = O(\lg m)$. That is, the search never ascends more than $O(\lg m)$ levels.

On the other hand, while searching down the tree, x only traverses at most one lateral link. Therefore we can conclude that the total search cost is $O(\lg m)$.

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 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.

Solution:

The resulting deterministic skip-list is exactly the level-linked B$^+$-tree that was just developed in the prior parts. We repeat the same data structure here in a slightly different form, primarily to help see the connection between the two data structures.

The data structure consists of a set of linked lists, just as a typical skip list. The lists all begin with $-\infty$. The SEARCH algorithm remains exactly as in the regular skip-list data structure. Each node in a linked list is augmented to include a count of consecutive non-promoted nodes that follow it in the list. For example, if the next node in the linked list is promoted, the count is 0; if the next node is not promoted, but the following node is promoted, the count is 1. The goal is to ensure that the count of a promoted node is at least $B-1$ and no more than $2B-1$. (The slightly different values as compared to a B-tree result from the slightly different definition of the count.)

We now describe how the INSERT proceeds. The INSERT operation begins at the top level of the skip list at $-\infty$, and proceeds down the skip list following the usual skip list search. On reaching the lowest level, the new element is inserted into the linked list, and the counts are updated. (Notice that no more than $2B-1$ counts must be updated.) If updating the counts causes the previous promoted node to have a count larger than $2B-1$, then the node with count B is promoted. This continues recursively up the tree, adding a new level at the highest level if needed. In this way, we ensure that the skip list is always close to a perfect skip list, without using any randomization.

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

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

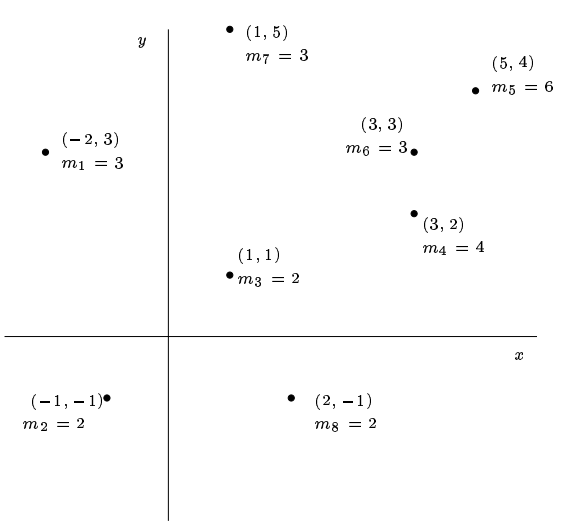

Figure 1: In this example of eight points, if $f(p_i) = m_i$, then $F(S) = 25$.

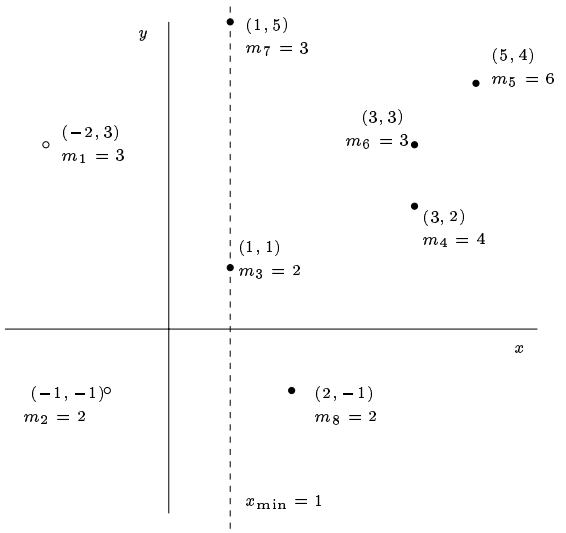

Figure 2: When $x_{\min} = 1$, $T(1) = \{p_3, p_4, p_5, p_6, p_7, p_8\}$ and $F(T(1)) = 20$.

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.

Solution:

For this part of the problem, we present two different solutions using binary tree. In the first version, we store keys only at the leaves of the binary tree, while in the second version we store keys at every node. The solutions are nearly identical.

Let $g(z)$ be the sum of $f(p_i)$ for all points p_i represented by the subtree rooted at z. If z is a leaf, then $g(z)$ is simply the value of $f(p_i)$ for the point stored in z. For both solutions, we augment each node z in the tree to store $g(z)$. The idea behind a query is to search for $x_{\min}$ in the binary tree, computing the desired sum along the way.

The first solution applies when the keys are stored only at the leaves. If we reach a node z in our search and $\mathit{key}[z] < x_{\min}$, then recurse on the right subtree. Otherwise, we add the value $g(\mathit{right}[z])$ to the answer from recursion on the left subtree.

```
FWITHXMIN-V1(z, x_min)          ▷ Keys stored at leaves.
1  if z = NIL
2     then return 0
3  if leaf[z]
4     then if key[z] ≥ x_min return g(z)
5     else return 0
6  if key[z] < x_min
7     then return FWITHXMIN-V1(right[T], x_min)
8     else return g(right[z])+ FWITHXMIN-V1(left[T], x_min)
```

Note that in the last step, we could also avoid visiting z's right child by noticing that $g(\textit{right}[z]) = g(z) - g(\textit{left}[z])$.

When the keys can be stored at internal nodes in the tree, the only modification to the query is that we have to look at the value of $f(p_i)$ for the node z we are currently visiting (in a slight abuse of notation, call this $f(z)$).

```
FWITHXMIN-V2(z, x_min)          ▷ Keys stored at every node.
1  if z = NIL
2     then return 0
3  if key[z] < x_min
4     then return FWITHXMIN-V2(right[T], x_min)
5     else return f(z) + g(right[z])+ FWITHXMIN-V2(left[T], x_min)
```

As before, in the last recursive call, we can avoid visiting z's right child by noticing that $f(z) + g(\textit{right}[z]) = g(z) - g(\textit{left}[z])$.

Finally, one could prove the correctness of FWITHXMIN by induction on the height of the tree.

(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)?

Solution: Constructing the data structure requires $O(n \lg n)$ because we are effectively sorting the points. One solution is to sort the points by x coordinate, and then construct a balanced tree by picking a median element as the root and recursively constructing the left and right subtrees. This construction takes $O(n \lg n)$ time to sort the points and $O(n)$ to construct the tree.

(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?

Solution: Using parts (a) and (b), we require $\Theta(n \lg n + k \lg n)$ time to build the data structure and then answer k queries. The naïve algorithm to compute $F(T(x_{\min}))$ from scratch is for each query, to scan through all n points and sum those with x-coordinate at least $x_{\min}$. This second algorithm requires $\Theta(kn)$ time. Intuitively, from these expressions, we can conclude that if $k = \omega(\lg n)$, then building the data structure will be more efficient than computing F from scratch.

We were not looking for a formal proof for this part, but one way to give such an argument is as follows. Let $T_1(n, k)$ and $T_2(n, k)$ be the times required to answer k queries on n points, for the first and second algorithms, respectively. Then we want to give sufficient conditions on k such that $T_1(n, k) \leq T_2(n, k)$ asymptotically.

Lemma 3 *If $k = \omega(\lg n)$, then there exists a constant n_0 such that $T_1(n, k) < T_2(n, k)$ for all $n \geq n_0$.*

Proof. Let c_1 and n_1 be the constants for the $O(n \lg n + k \lg n)$, for the first algorithm, and let c_2 and n_2 be the constants for $\Omega(kn)$ for the second algorithm.

Then we want to give sufficient conditions on k and n that give us

$$T_1(n, k) \leq c_1(n \lg n + k \lg n) < c_2 kn < T_2(k, n).$$

If $n \geq \max\{n_1, n_2\}$, then we know

$$k > \frac{c_1 n \lg n}{c_2 n - c_1 \lg n}.$$

Let n_3 be a constant such that for all $n \geq n_3$, $c_1 \lg n \leq c_2 n/2$. Then, for all $n \geq \max\{n_1, n_2, n_3\}$, we know that

$$k > \left(\frac{2c_1}{c_2}\right) \lg n.$$

If $k = \omega(n)$, then by definition, choosing the constant c to be $2c_1/c_2$, we can find a value n_4 such that for all $n \geq n_4$, k is large enough to satisfy the above condition. Therefore, if $n_0 = \max\{n_1, n_2, n_3, n_4\}$, $T_1(n, k) < T_2(n, k)$ for all $n > n_0$.

(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.

Solution:

When keys are stored at every node, the augmentation of maintaining $g(z)$ at every node in a red-black tree is almost exactly the same as the augmentation to maintain the sizes of subtrees for the dynamic order-statistic tree.

When we insert a point p to the tree, we increment $g(z)$ by $f(p)$ for the appropriate nodes z as we walk down the tree. For a delete, when a point p is spliced out of the tree, we walk up the path from p to the root and decrement $g(z)$ by $f(p)$. In the cases where we delete a node p by replacing it with its predecessor or successor p', then we walk up the path from p and increment $g(z)$ by $f(p') - f(p)$, and then recursively delete p' from the tree rooted at p.

Maintaining $g(z)$ during rotations is also similar. For example, after performing a left-rotation on a tree that originally had x as the root and y as its right child (see Figure 14.2 on p.306), we update $g(x)$ and $g(y)$ by $g(y) \leftarrow g(x)$ and $g(x) \leftarrow g(\textit{left}[x]) + g(\textit{right}[x]) + f(x)$.

Note that for the first type of solution, when keys are stored only at the leaves in a red-black, we also need to maintain the maximum value of the left subtree of z. We should also argue that this property can be maintained. Similar arguments for normal tree inserts and deletes and for rotations will work.

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 $x_{\max}$.

Solution:

First, we find a split node w for $x_{\min}$ and $x_{\max}$. The split node w satisfies the property that it is the node of greatest depth whose subtree contains all the nodes whose keys are in the interval $[x_{\min}, x_{\max}]$. We find a split node by following the normal search algorithm for $x_{\min}$ and $x_{\max}$ until their paths diverge.

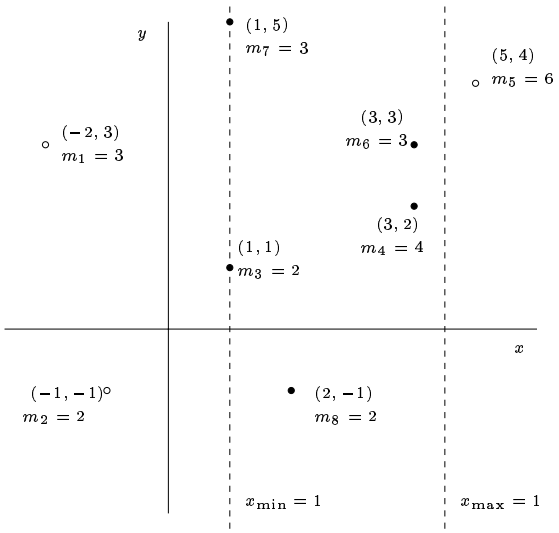

Figure 3: When $X = [1, 3.5]$, $T(X) = \{p_3, p_4, p_6, p_7, p_8\}$ and $F(T(X)) = 14$.

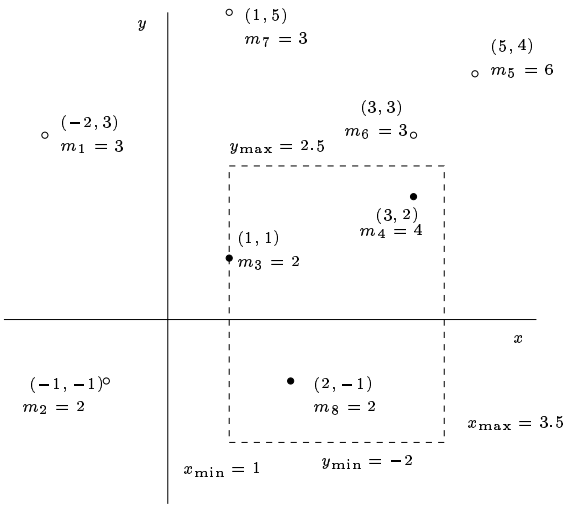

Figure 4: For $X = [1, 3.5]$ and $Y = [-2, 2.5]$, $T(X, Y) = \{p_3, p_4, p_8\}$ and $F(T(X, Y)) = 8$.

FINDSPLITNODE-V1$(z, x_{\min}, x_{\max})$ ▷ Keys stored at leaves.

```
1  if z = NIL
2     then return NIL
3  if leaf[z]
4     if (x_min ≤ key[z] ≤ x_max) return z
5     else return NIL
6  if (x_max ≤ key[z])
7     then return FINDSPLITNODE-V1(left[T], x_min, x_max)
8  if (key[z] < x_min)
9     then return FINDSPLITNODE-V1(right[T], x_min, x_max)
10 return z
```

FINDSPLITNODE-V2$(z, x_{\min}, x_{\max})$ ▷ Keys stored at every node.

```
1  if z = NIL
2     then return NIL
3  if (x_max < key[z])
4     then return FINDSPLITNODE-V2(left[T], x_min, x_max)
5  if (key[z] < x_min)
6     then return FINDSPLITNODE-V2(right[T], x_min, x_max)
7  return z                          ▷ If we get here, x_min ≤ key[z] ≤ x_max.
```

If FINDSPLITNODE returns NIL, then $F(T(X)) = 0$ because there are no points in the interval. Otherwise, to compute $F(T(X))$, we can use the function FWITHXMIN

from part (a) and a corresponding function FWITHXMAX (this function computes $F(T([-\infty, x_{\max}]))$).

```
FWITHXMAX-V1(z, x_max)        ▷ Keys stored at leaves.
1  if z = NIL
2     then return 0
3  if leaf[z]
4     then if key[z] ≤ x_max return g(z)
5     else  return 0
6  if key[z] < x_max
7     else  return g(left[z]) + FWITHXMAX-V1(right[T], x_max)
8     then return FWITHXMAX-V1(left[T], x_max)
```

```
FWITHXMAX-V2(z, x_max)        ▷ Keys stored at every node.
1  if z = NIL
2     then return 0
3  if key[z] > x_max
4     then return FWITHXMAX-V2(left[T], x_max)
5     else  return f(z) + g(left[z]) + FWITHXMAX-V2(right[T], x_max)
```

If we have a split node w, we know all nodes in the left subtree of w have keys at most $x_{\max}$, and all nodes in the right subtree have keys at least $x_{\min}$.
If all keys are stored at the leaves, we have

$$F(T(X)) = \text{FWITHXMIN-V1}(\textit{left}[w], x_{\min}) + \text{FWITHXMAX-V1}(\textit{right}[w], x_{\max}).$$

When keys are stored at both leaves and internal nodes, we have

$$F(T(X)) = f(w) + \text{FWITHXMIN-V2}(\textit{left}[w], x_{\min}) + \text{FWITHXMAX-V2}(\textit{right}[w], x_{\max}).$$

```
FWITHX-V2(z, x_min, x_max)
1  if z = NIL
2     then return 0
3  w ← FINDSPLITNODE-V2(z, x_min, x_max)
4  if w = NIL
5     then return 0
6     else  return f(w) + FWITHXMIN-V2(left[z], x_min) + FWITHXMAX-V2(right[z], 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.

(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.

Solution: We use a 2-d range tree with primary tree keyed on the x-coordinate. Each node z in this x-tree contains a pointer to a 1-d range tree, keyed on the y coordinate. We augment each of the nodes in the y-trees with the same $g(z)$ as in part (a).

To perform a query, we first search in the primary x-tree for all nodes/subtrees whose points have x-coordinates that fall in the interval $[x_{\min}, x_{\max}]$. Then, for each of these nodes/subtrees, we query the y-tree to compute F for all the points in that y-tree whose y-coordinate falls in the interval $[y_{\min}, y_{\max}]$.

We can use the exact same pseudo code from parts (a) and (e) if we replace every occurrence of $g(z)$ with FWITHY(*ytree*$[z], y_{\min}, y_{\max}$), where FWITHY is the query on a 1-d range tree.

The algorithm for queries is the same as in part (a), except replacing the constant-time $g(z)$ function calls with the calls to query *ytree*$[z]$. Since each query in a y-tree takes $O(\lg n)$ time, the total runtime for a query is $O(\lg^2 n)$.

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

Solution:

Since we are only adding a constant amount of information at every node in a y-tree or x-tree, the space used is asymptotically the same as a 2-d range tree, i.e., $O(n \lg n)$. For a 2-d range tree, the space used is $O(n \lg n)$ because every point appears in $O(\lg n)$ y-trees.

A simple algorithm for constructing a 2-d range tree takes $O(n \lg^2 n)$ time. First, sort the points by x-coordinate and construct the x-tree in $O(n \lg n)$ time as in part (c). Then, for every node z in the x-tree, construct the corresponding y-tree, using the algorithm from (c) by sorting on the y-coordinates. The recurrence for the time to construct all y-trees is $T(n) = 2T(n/2) + O(n \lg n)$, or $T(n) = O(n \lg^2 n)$.

It is possible to construct a 2-d range tree in $O(n \lg n)$ with a more clever solution. Notice that in the 2-d range tree, for a node z in the x-tree, the y-trees for the left and right subtrees of z are disjoint. This observation suggests the following algorithm.

First, sort all the points by their x-coordinate and construct the primary range tree as before. Then, sort the points by y-coordinate, and use that ordering to construct the largest y-tree for the root node z. To get the y-trees for *left*$[z]$ and *right*$[z]$, perform a *stable* partition on the x-coordinates of the array of points about *x-key*$[z]$, the key of z in the x-tree. This partition allows us to construct the y-trees for the left and right subtrees of z without having to sort the points by y-coordinate again. Therefore, the recurrence for the runtime is $T(n) = 2T(n/2) + O(n)$, or $O(n \lg n)$.

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)?

Solution: We can't generalize the argument in part (d) to a 2-d range tree because we can not easily update the x-tree. Performing a normal tree insert on a single x or y tree can be done in $O(\lg n)$ time. Performing a rotation on a node in the x-tree, however, requires rebuilding an entire y-tree. For example, consider the diagram of left-rotation in CLRS, Figure 13.2 on pg. 278. The y-tree for y after the rotation contains exactly the same nodes as y-tree for x before the rotation. To fix the y-tree rooted at x after the rotation, however, we must remove all the points in γ from y's original y-tree and add all the points in α.

In the worst case, if x is at the root, then the subtrees α and γ will have $O(n)$ nodes. Thus, performing a rotation in x-tree might require $\Omega(n)$ time. Since an update to a red-black tree requires only a constant number of rotations, the time is $O(n)$ for an update.

(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?

Solution:

If we do not have to worry about maintaining balance in any of the x-trees or y-trees, then we can use a normal tree insert to add each new point into the x-tree and the corresponding y-trees. This algorithm requires $O(\lg^2 n)$ time per update because we have to add a point to $O(\lg n)$ y-trees.

As a side note, when we are performing a small number of updates, it is actually possible to perform queries in $O(\lg^2 n)$ time and handle updates in $O(1)$ time! In this scheme, we augment the range tree with an extra buffer of size $O(\lg^2 n)$. Insert and delete operations are lazy: they only get queued in the buffer. When a query happens, it searches the original 2-d range tree and finds an old answer in $O(\lg^2 n)$ time. It then updates its answer by scanning through each point in the buffer. Thus, we can actually support $O(\lg^2 n)$ updates without changing the asymptotic running time for queries.

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

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

Solution:

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

Set $\partial f / \partial \rho$ equal to 0 to find ρ^* and θ^*, we have

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

(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

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

Solution: This question is also doable, assuming the solution given above is actually correct. You may want to double-check the math for this solution.
We compute $\partial f / \partial \theta$ and set it equal to 0.

$$\begin{aligned}
\frac{\partial f}{\partial \theta} &= \sum_{i=1}^{n} \left(2m_i(x_i \sin\theta - y_i \cos\theta + \rho)(x_i \cos\theta + y_i \sin\theta)\right) \\
&= \sum_{i=1}^{n} \left[2m_i \left((x_i^2 + y_i^2)\sin\theta\cos\theta - x_i y_i(\cos^2\theta - \sin^2\theta) + \rho(x_i\cos\theta + y_i \sin\theta)\right)\right] \\
&= \sum_{i=1}^{n} \left[m_i \left((x_i^2 + y_i^2)\sin 2\theta - x_i y_i \cos 2\theta + 2\rho(x_i\cos\theta + y_i \sin\theta)\right)\right] \\
&= \left(M_{x2} + M_{y2}\right)\sin 2\theta - 2M_{xy}\cos 2\theta + 2\rho\left(M_{x1}\cos\theta + M_{y1}\sin\theta\right).
\end{aligned}$$

To find the extrema (ρ^*, θ^*) where both partial derivates are 0, we plug in the expression from the previous part to eliminate ρ^*.

$$\rho^* = -\frac{M_{x1}\sin\theta^* - M_{y1}\cos\theta^*}{M_0}$$

$$\begin{aligned}
\left(M_{x2} + M_{y2}\right)\sin 2\theta^* - 2M_{xy}\cos 2\theta^* - \\
\frac{2\left(M_{x1}\sin\theta^* - M_{y1}\cos\theta^*\right)\left(M_{x1}\cos\theta^* + M_{y1}\sin\theta^*\right)}{M_0} &= 0 \\
\left(M_{x2} + M_{y2}\right)\sin 2\theta^* - 2M_{xy}\cos 2\theta^* - \\
\frac{2(M_{x1}^2 - M_{y1}^2)\sin\theta^*\cos\theta^* - 2M_{x1}M_{y1}(\cos^2\theta^* - \sin^2\theta^*)}{M_0} &= 0
\end{aligned}$$

$$\begin{aligned}
\left(M_{x2} + M_{y2} - \frac{M_{x1}^2 - M_{y1}^2}{M_0}\right)\sin 2\theta^* &= 2\left(M_{xy} - \frac{M_{x1}M_{y1}}{M_0}\right)\cos 2\theta^* \\
\tan 2\theta^* &= \frac{2\left(M_0 M_{xy} - M_{x1}M_{y1}\right)}{M_0\left(M_{x2} + M_{y2}\right) - \left(M_{x1}^2 - M_{y1}^2\right)}
\end{aligned}$$

(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.

Solution: Use $f(p_i) = m_i(1, x_i, y_i, x_i y_i, x_i^2, y_i^2)^T$.