Data Structures Homework #4

Ari Feiglin

Question: 4.1:

We are given n columns in a row, the height of each column is given in an array $A[1 \dots n]$. We want to find the maximum area of a rectangle tha can be constructed within the columns.

Construct an algorithm where given an array A finds this maximum area. Prove the algorithm works and compute its time complexity.

Let's try and rephrase the question. What exactly is the area of a rectangle spanned by a column? It is the height of that column multiplied by the length of the interval in which it is the local minimum. So the question, phrased in a way to make it easier for us to find an algorithm for is:

> *"What is the maximum value of a column multiplied by the interval in which it is the local minimum?"*

So the question just reduces to finding an algorithm the length of the interval in which an element in the array is a local minimum. We will do this by finding the bounds of this interval.

We can do this in two steps: the first step will find the left bound and the second step will find the right bound. These two steps are very similar, almost identical even.

In order to find the left bound, we create an array which will store the left bounds for each element in the array. We fill up this array by iterating over $A[1 \dots n]$ and finding their bounds. Suppose we are on the ith element in $A[1 \dots n]$. Notice that if $A[i-1] < A[i]$, then the left bound is i itself (or $i-1$, depending on how you want to think about it). Otherwise, we already know what the left bound of $i-1$ is, suppose its j. So we know that for every element between j and $i-1$, $A[i-1]$ is smaller than it. And since $A[i-1] \geq A[i]$, this means that $A[i]$ is smaller than all those elements as well, so we don't need to check those, we can skip to element j, find its left bound (if it is not smaller than $A[i]$), and repeat the process.

In pseudocode this algorithm is:

Input : An array A.
Output: An array, leftBounds, which has the left bound of the interval of where each element in A is a loca minimum.

```
1 leftBounds.append(0);                  // Since the left bound of the first element will be 0
2 for i ← 2 to A.length do
3 |   j ← i − 1;
4 |   while j ≠ 0 and A[j] ≥ A[i] do
5 |   |   j ← leftBounds[j];
6 |   end
7 |   leftBounds.append(j)
8 end
```

We do a similar process for right bounds, this time starting from the end of $A[1 \dots n]$ and moving left:

```
Input  : An array A.
Output: An array, rightBounds, which has the right bound of the interval of where each element in A is
        a local minimum.
1 rightBounds [A.length ]=A.length +1;  // Since the right bound of the last element will be
   n + 1
2 for i ← A.length − 1 to 1 do
3 |   j ← i + 1;
4 |   while j ≠ A.length + 1 and A[j] ≥ A[i] do
5 |   |   j ← rightBounds[j];
6 |   end
7 |   rightBounds[i] ← j;
8 end
```

So now we have to arrays, `leftBounds` and `rightBounds` which store the left and right bounds of the "minimum interval" of each element in A (the "minimum interval" being the interval where the element is a local minimum). Notice that these bounds are the first elements smaller than the element, so the length of the "minimum interval" of the ith element in A is:

$$\texttt{rightBounds}[i] - \texttt{leftBounds}[i] - 1$$

So now we iterate over these three arrays (`leftBounds`, `rightBounds`, and A) and multiply $A[i]$ by this length, while keeping track of the maximum area.

```
Input  : An array A.
Output: The maximum area of the rectangle inscribed in A.
1 maxArea ← 0;
2 for i ← 1 to A.length do
3 |   if A[i] · (rightBounds[i] − leftBounds[i] − 1) > maxArea then
4 |   |   maxArea ← A[i] · (rightBounds[i] − leftBounds[i] − 1);
5 |   end
6 end
```

When finding the left bound, we can think of going from $A[i]$ to $A[j]$ (where j is the left bound of $A[i-1]$) as a "path" from i to j. Each path can only be traversed twice: the first time when finding the left bound of $A[i]$, and a second time if another element is checking $A[i]$'s left bound. Once another element, say $A[j]$, checks $A[i]$'s left bound, it must be smaller than $A[i]$ (otherwise its left bound would be i) so all elements after j will not check $A[i]$, as if they're larger than $A[j]$, they will stop checking at most by $A[j]$, and otherwise they may reach $A[j]$ and then skip to $A[i]$'s left bound, skipping over $A[i]$ and therefore its path.

And since there can be at most n paths (from each element to its adjacent element), the number of total traversals cannot exceed $2n$. And since each traversal takes constant time (since its just changing a constant number of variables and doing a constant number of checks), the time complexity of finding a left bound is $\mathcal{O}(2n) = \mathcal{O}(n)$.

And since finding the right bound is almost identical to finding the left bound, its time complexity is also $\mathcal{O}(n)$.

Finally, the last step of iterating through the three arrays also takes $\mathcal{O}(n)$ time, since it just iterates through array A.

So all in all the time complexity of this algorithm is $\mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$.

Question: 4.2:

Given a set of $n = 2^k$ elements, a **Perfect Skip List** is a skip list where on the ith level there are exactly 2^{k-i} elements (along with $\pm\infty$), and the distance between adjacent elements is 2^i.

Construct a data structure which implements a perfect skip list which allows for the following methods:

(1) The construction of the data structure given a sorted array of n elements, with a time complexity of $\mathcal{O}(n)$.

(2) Normal querying: given a number y, finds it if it exists, in $\mathcal{O}(\log n)$ time.

(3) Pointed querying: given a pointer p to an element x_i in the datastructure and a value $y = x_j$, returns a poimter to x_j in $\mathcal{O}(\log |j - i|)$ time.

My idea for this data structure is to implement a normal skip list, but each node is quad-directional. That is, each node has a pointer to the next node and previous node in its layer, as well as a node to the node in the layer above it (if there is any), and the node in the layer below it.

Thus this skip list will have the same functionality as a normal skip list since it has the two pointers a normal skip list has (the "next" pointer and the "bottom" pointer), as well as two more.

(1) In order to construct the data structure, we iterate over the array and insert each element into the base level of the skip list. For even elements, we elevate them to the next level, and if we can divide them by 2 again, to the next, and so on.

```
Input  : An array A of n = 2^k sorted elements.
Output: A perfect skip list of A.
levels ← {−∞};
prevNodes ← {−∞};
A.append(∞);                          // In order to append ∞ to the end of the skip list
for i ← 1 to n + 1 do
    j ← i;
    currLevel ← 0;
    prevBottom ← null;
    do
        Node currNode = Node(A[i]);
        prevNodes[currLevel].next ← currNode;
        currNode.previous ← prevNodes[currLevel];
        prevNodes[currLevel] ← currNode;
        prevBottom.top ← currNode;
        currNode.bottom ← prevBottom;
        prevBottom ← currNode;
        levels++;
        j ÷= 2;
    while 2 | j;
end
```

This constructs all the nodes in the skip list, each node taking constant time to construct. Since there are:

$$\sum_{i=0}^{k} 2^i + 2k = 2^{k+1} - 1 + 2k = 2n - 1 + 2\log n$$

nodes in the skip list (the $2k$ come from the $\pm\infty$s), it takes:

$$\mathcal{O}(2n - 1 + 2\log n) = \mathcal{O}(n)$$

Time to construct, as required.

(2) Normal searching can be done as is normally in a skip list: Start at the topmost level's $-\infty$, and check if the next node is greater than the goal value. If it is, then go down a level and repeat. Otherwise, go to the next node and repeat.

At each iteration of this, we halve the number of elements we are searching over. This is because if we go to the next node, we can ignore the half between the previous node and up to the node we moved to. And if we stay at the current node, we can ignore all the elements past the next node as well. And since this is a perfect skip list, we're ignoring half the elements either way.

Once we reach the base level, if the node we're on and the next node are not equal to our target, then the target is not in the skip list and we can throw an error or whatever.

Since at each step we're removing half the elements to search over, the time complexity of this algorithm is:

$$\mathrm{T}(n) = \mathrm{T}\left(\frac{n}{2}\right) + \mathcal{O}(1)$$

Which, by the master theorem, means that the algorithm has a time complexity of:

$$\mathcal{O}(\log n)$$

As required.

(3) The first step of this algorithm is to get to a point where we can use the previous algorithm. The easiest case is if $x_i = y$, then we return p, the pointer to x_i. This takes constant time and thus has a time complexity of $\mathcal{O}(\log|j - i|)$ trivially.

Otherwise, suppose $x_i < y$, if the current node is less than y and the next node is less than y, then we go to the current node's top node, and if none exist, the next node's top node (one must exist since every other node has a top node). We do this until we get to a point where the next node is greater than or equal to y.

During each iteration we halve the number of nodes between the current node and y (or a node greater than y), and since the distance starts off at $|j - i|$, it takes $\mathcal{O}(\log_2|j - i|)$ iterations to reach a point where the next node is greater than or equal to y.

At this point, we're at a point which simulates a normal skip list query (we can think of x_i or less as $-\infty$ and values greater than y as ∞). This is because we're at a height of $\log_2|j - i|$ over a list which has a length of $|j - i|$, and since the original skip list was a perfect skip list, so is this one.

And by our time analysis of the previous subquestion, the time complexity from here on out is $\mathcal{O}(\log|j - i|)$. Since the time complexity of the first part of this algorithm (ascending the skip list) is also $\mathcal{O}(\log|j - i|)$, the total time complexity is:

$$\mathcal{O}(\log|j - i|) + \mathcal{O}(\log|j - i|) = \mathcal{O}(\log|j - i|)$$

As required.

This is the case if $x_i < y$, and if $x_i > y$ we do a similar process of ascending the skip list, but instead of moving forward, we move backwards. And once we get to a point where we can do a normal skip list query, we again move backwards and down instead of forwards and down.

This is almost identical (save the direction) to the previous process, and for the same reason has a time complexity of $\mathcal{O}(\log|j - i|)$, as required.

Question: 4.3:

Prove that an in-order traversal of a binary search tree prints all the elements of the tree in order.

Let's prove this by induction on the height of the tree, h.

Base case: $h = 0$. In this case, the tree is but a single node, and an in-order traversal of it just prints that node, which is trivially in order since there is only one node.

Inductive step: Suppose the inductive hypothesis is true for binary search trees of heights strictly less than h.

Suppose the root of the tree is r and the left and right subtrees of the root are T_ℓ and T_r respectively. An in-order traversal will print:

$$`T'_\ell \; `r' \; `T'_r$$

(Where $`X'$ denotes the object X printed.) By definition.

And since T_ℓ and T_r are binary search trees, and an in-order traversal of the root will traverse them too in-order and since their heights are strictly less than h, $`T'_\ell$ and $`T'_r$ are printed in order by our inductive hypothesis.

Since the tree is a binary search tree, we know $T_\ell \leq r \leq T_r$, so the printing of:

$$`T'_\ell \; `r' \; `T'_r$$

Is also in order (since each tree is printed in order, and the order of the printing, T_ℓ then r then T_r is in order as well), as required.

Question: 4.4:

Construct a data structure which supports the following methods:

(1) Insertion

(2) Erasure

(3) Querying: given a value, returns a pointer to the value in the data structure

(4) Sum-Querying: given two values $x < y$, returns the sum of all values in the interval $[x, y]$ in the data structure.

All methods must have a time complexity of $\Theta(\log n)$.

My idea is to construct this data structure using an AVL tree. Along with all relevant AVL information, each node will also have a field which stores the value of the sum of its right and left subtrees. Using this field we will be able to accomplish Sum-Querying in the required time.

(1) Insertion will be done as is normally in an AVL tree, with two differences:

- At each node, if the value we're inserting is larger than that node (so if we're inserting it into its right subtree), increase that node's right subtree sum field by that value, and if its less than that node (so we're inserting it into its left subtree), increase that node's left subtree sum by the value we're inserting.

  Note that since this takes constant time, it doesn't affect the time complexity of insertion.

- During rotations, we need to save the fields of each node we're rotating and redistribute the sums accordingly.

  So the algorithm for a left rotation will look like:

Input : B, the root of a subtree to perform a left rotation on.
Output: A, the new root of the rotated subtree.

```
1  A ← B.right;
2  T1 ← B.left;
3  T2 ← A.left;
4  T3 ← A.right;
   /* For readibility, I defined the following                    */
5  T1value ← B.leftSum;
6  T2value ← A.leftSum;
   /* Rotate the tree normally during a left rotation, fixing the balance factor
      as well.                                                    */
   /* Fix the sum fields                                          */
7  A.leftSum ← B.value + T1value + T2value;
8  B.rightSum ← T2value;
   /* A's right sum and B's left sum don't change through a left rotation.  */
9  return A;
```

Algorithm 1: Left-rotations of the tree

And the algorithm for a right rotation is similar:

Input : B, the root of a subtree to perform a right rotation on.
Output: A, the new root of the rotated subtree.

```
1 A ← B.left;
2 T_1 ← A.left;
3 T_2 ← A.right;
4 T_3 ← B.right;
  /* For readibility, I defined the following                          */
5 T_2value ← A.rightSum;
6 T_3value ← B.rightSum;
  /* Rotate the tree normally during a right rotation, fixing the balance factor
     as well.                                                          */
  /* Fix the sum fields                                                */
7 A.rightSum ← B.value + T_2value + T_3value;
8 B.rightSum ← T_2value;
  /* A's left sum and B's right sum don't change through a right rotation.  */
9 return A;
```

Algorithm 2: Right-rotations of the tree

Both of these take $\mathcal{O}(1)$ time since the rotation of the nodes, the redistribution of the sums, and all other checks and computations take constant time, and there are a constant amount of them. So rotations don't affect insertion time.

These two differences don't affect insertion time, and otherwise, insertion is the same as a normal AVL insertion (finding where to insert by doing a normal search of a binary tree, then inserting and fixing height disparities throuh rotations). This takes $\Theta(\log n)$ time since the search and fixing take $\Theta(\log n)$ time (since its an AVL tree, its height is $\Theta(\log n)$, so searching takes $\Theta(\log n)$ time, and fixing does as well since we have to fix $\Theta(\log n)$ trees, and each fix takes constant time, this was discussed in recitation).

So insertion passes the requirements.

(2) Erasure is similar to insertion. First we search for the element, and for every node we pass over, if the target is greater than the node, decrease the right sum of the node by the target, and otherwise decrease the left sum of the node by the target. Once we find the target, delete it as normal, and fix the height disparities through rotations (using the algorithm defined in the previous subquestion).

If the target isn't found, it doesn't exist, so after each recurrence call from the nodes we traversed in search of the target, increase the node's left or right sum by the target (since we originally decreased it, so adding it back brings the field to what it was previously).

Finding the target takes $\Theta(\log n)$ time (as discussed above), and fixing does as well (dicussed above as well). So all in all erasure takes $\Theta(\log n)$ time, as required.

(3) Searching through the AVL tree is just simply searching through a balanced binary search tree, so we can search the tree with a simple binary search of the tree, which takes $\Theta(\log n)$ time.

(4) Notice that the sum of all elements between x and y is equal to the sum of all elements greater or equal to x minus the sum of all elements strictly greater than y. So this problem reduces to finding the sum of all elements greater than some arbitrary value x in $\Theta(\log n)$ time.

We can do this by performing a DFS of the tree. For every node we visit, if its value is greater than x, then everything in its right subtree is greater than x as well, so we can add the sum of its right subtree plus the value of the node itself to the sum greater than x. So then we need to continue searching for elements greater than x in its left subtree.

Otherwise, if its value is less than x, then everything in its left subtree is less than x as well, so we can ignore it and search through its right subtree.

If the node is equal to x, then we know everything in its right subtree is greater than x, and everything in its left subtree is less than x, so we add the sum of its right subtree (plus the node's value, x, if we're searching for the sum of all elements in the tree greather *or equal* to x) to the sum greater than x. Then we can stop, since the left subtree is less than x.

Suppose the root of the tree is node, then the algorithm for summing all values greater than x in the data structure is:

```
   Input  : target, the target value; and include, a boolean value.
   Output: The sum of all elements in the data structure greater than target, including target if
            include is true.
 1 while node ≠ null and node.value ≠ target do
 2 |  if node.value > target then
   |  |  /* The node is greater than the target, so so is its entire right subtree,
   |  |     so add it (and the node) to the sum.                                    */
 3 |  |  sum ← sum + node.rightSum + node.value;
 4 |  |  node ← node.left;
 5 |  else
   |  |  /* The node is less than the target, so so is its entire left subtree, so we
   |  |     can ignore it.                                                          */
 6 |  |  node ← node.right;
 7 |  end
 8 end
 9 if node.value = target then
   |  /* The node is equal to the target, so its right subtree is greater than the
   |     target, and its left subtree is less than the target.                     */
10 |  sum ← sum + node.rightSum;
11 |  if include then sum ← sum + node.value;
12 end
13 return sum;
```

Algorithm 3: getSumLarger - gets the sum of values larger than a target value in the tree.

This algorithm just descends down the tree, as if searching for x, spending $\mathcal{O}(1)$ time on each depth. Since the height of the tree is $\Theta(\log n)$, since it's an AVL tree, this algorithm takes $\Theta(\log n)$ time.

And finally, the algorithm for summing the elements in the interval $[x, y]$ is simply, as discussed before:

```
   Input  : x < y, two numbers.
   Output: The sum of all numbers in the datastructure in the interval [x, y].
   /* The sum of numbers between x and y (inclusive) is equal to the sum of numbers
      greater or equal to x minus the sum of numbers strictly greater than y.    */
 1 return getSumLarger(x, true) - getSumLarger(y, false);
```

And since getSumLarger has a time complexity of $\Theta(\log n)$, this algorithm has a time complexity of:

$$\Theta(\log n) + \Theta(\log n) = \Theta(\log n)$$

As required.