COMP 252 Assignment 4

Christopher Hundt (110220945)

March 27, 2003

Question 1

Note that in the following explanation, the "size" of a tournament tree refers to the number of its leaves.

When a single item is added to an LLT, the element is made into a tournament tree of size 1, and then put on the linked list. When making an entire LLT of n items, this operation has to be repeated n times. This takes time $\Theta(n)$, because elements are added to the end of the list.

However, there may be merges in each step in addition to the adding to the linked list. For instance, the first time a size 1 tree is added to the list, no merges are necessary. But when the second size 1 tournament tree is added, those two must now be merged to form a single tree of size 2. We note, however, that merges always result in a larger tree; there are no size 1 trees produced or "left over." We also note that the smallest trees, the only ones with a possibility of merging, stay on the end of the linked list, so it is still time $\Theta(1)$ to find them.

Analyzing the insertions, we consider that, immediately before each insertion, the tree must be a valid LLT (that is, no two trees of the same size exist, and all trees have sizes equal to powers of 2). This means that, when inserting a single element, the only kind of merge that can happen immediately is a merge of 2 size 1 trees. After this happens, there are no more size 1 trees, so the next insertion will not result in a merge, but the one after that will. We know that two more insertions is the only way to get two size 1 trees because, as noted above, merges (the only other thing we are doing) do not result in smaller or same-size trees. So we can say that the number of merges of size 1 trees is equal to $\lfloor \frac{n}{2} \rfloor$.

We quickly see, though, that one merge may lead to another. For instance, after 3 insertions we will have 1 tree of size 1 and 1 tree of size 2. Adding a fourth element results in a merge of the two size 1 trees, which creates a second size 2 tree, which now must be merged with the other size 2 tree, resulting in a single size 4 tree. But we see that a merge of this size will not be performed again until 2 more size 2 trees exist, which means two more merges of size 1 trees, which means 4 more insertions, as noted above. So the number of merges of size 2 trees is equal to $\lfloor \frac{n}{4} \rfloor$. By similar logic, the number of merges of size 4 trees is equal to $\lfloor \frac{n}{8} \rfloor$.

It quickly becomes apparent that the number of merges of size i trees (where i is a power of 2) after n insertions is equal to $\lfloor \frac{n}{2i} \rfloor$. The maximum i after n insertions is $\frac{n}{2}$ (because two size $\frac{n}{2}$ trees merge to form a size n tree, which holds all the elements). So, the total time T for all the merges is

$$\begin{aligned} T &\leq \lfloor \frac{n}{2} \rfloor + \lfloor \frac{n}{4} \rfloor + \lfloor \frac{n}{8} \rfloor + \ldots + 1 \\ &\leq n\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{n}\right) \\ &< n\sum_{i=1}^{\infty} \frac{1}{2^i} = n \end{aligned}$$

Thus, the number of merges is clearly less than n. Since each merge involves simply comparing the roots of the two trees being merged and making a new root it takes constant time. This means that the total time for both merges and insertions is $\Theta(n)$, and thus $O(n)$.

Question 2

This solution uses two trees: one (S) to hold the integers, and one (R) to hold their ratios. Both trees are binary search trees; their exact implementations can vary. The specific requirements for the implementations are:

- S has the operations `successor(x)` and `predecessor(x)`, which return the next-greatest and next-smallest, respectively, or null if such an element doesn't exist. This can be implemented via extra pointers (which do not add significantly to the time of any operations) or an algorithm, as discussed in class, which finds the requested element in time $O(\log s)$, where s is the size of S.
- R will accept multiple items of the same value, and will delete only one of them if asked to delete an item of that value. This can be accomplished, for instance, by adding "counter" attributes to the nodes.
- R keeps a pointer to its minimum item, accessed by `min[R]`. This can be maintained trivially, including after inserts and deletes, without adding significantly to the time.

Now, the following procedure can be used to insert a new integer k to the set.

```
if search(S, k) is not null then exit
insert(S, k)
x <-- node at which k was inserted
if successor(x) and predecessor(x) are both not null then
  delete(R, key[successor(x)]/key[predecessor(x)])
if successor(x) is not null then
  insert(R, key[successor(x)]/k)
if predecessor(x) is not null then
  insert(R, k/key[predecessor(x)])
```

The next procedure deletes an existing integer j.

```
x <-- search(S, j)
if x is null then exit
if successor(x) and predecessor(x) are both not null then
  insert(R, key[successor(x)]/key[predecessor(x)])
if successor(x) is not null then
  delete(R, key[successor(x)]/j)
if predecessor(x) is not null then
  delete(R, j/key[predecessor(x)])
delete(S, j)
```

To search for an item in the set, just search in S. To get the minimum ratio, just ask for `key[min[R]]`.

Running time for searching, deleting, and inserting are all clearly linear with the time to search, insert, and delete in R, S, or both. Thus, we can just pick an appropriate implementation of trees for R and S and assure that the worst-case or average-case running time is $O(\log n)$, because the number of elements in S is n, and the number of elements in R (the number of ratios) is $n-1$. The time for min-ratio, because we have a pointer to it, is $\Theta(1)$.

Question 3

We can create a tree T, each node of which has two keys, *start* and *slope*, each of which are single numbers. The nodes are ordered by *start*. The tree supports the operations `successor(x)` and `predecessor(x)`, which return the next-greatest and next-smallest nodes, respectively (by the *start* key, of course), or null if such an element doesn't exist. Finally, the tree has one very special attribute: it recalculates values for *start* whenever it checks it. Any algorithm, including the search and delete algorithms, must use not the `start[T]` key but the `start(T, x)` method, which expects a difference x in the x-coordinate at the current time. It then performs the operation `start[T] <-- start[T] + slope[T] * x` and returns the result. This means that the arguments expected by some methods change:

- `insert(T, a, b, x)`: Inserts a new node in T, and sets the key *start* to a, and *slope* to b. Sends x to `start()` when that method is called.
- `delete(T, a, b, x)`: Deletes a node in T with $start = a$ and $slope = b$. Sends x to `start()` when that method is called. b is specified only as a check. The tree is searched by a.
- `search(T, a, b, x)`: Finds a node in T with $start = a$ and $slope = b$. Sends x to `start()` when that method is called. b is specified only as a check. The tree is searched by a.

First, we create a list L of all the x-coordinates of the endpoints (both) of each line segment, along with which segment each coordinate is associated with. The list is size $2n$ and obviously takes time $O(n)$ to make. We sort the list by x-coordinate, which takes time $O(n \log n)$.

Next, we go through that list from beginning to end, following this procedure:

```
x0 <-- null
for i = 1 to 2n
  s <-- line segment related to L[i]
  x1, y1 <-- left endpoint of s
  x2, y2 <-- right endpoint of s
  m <-- slope of s ((y2-y1)/(x2-x1))
  if x0 is null then dx <-- 0
  else dx <-- x1 - x0
  if L[i] is a left endpoint (= (x1, y1)) then
    insert(T, y1, m, dx)
    x0 <-- x1
  if L[i] is a right endpoint (= (x2, y2)) then
    x <-- search(T, y2, m, dx)
    if x is null then return ``INTERSECTION''
    if predecessor(x) is not null then
      if start(predecessor(x), dx) >= y2 then return ``INTERSECTION''
    if successor(x) is not null then
      if start(successor(x), dx) <= y2 then return ``INTERSECTION''
    delete(T, y2, m, dx)
    x0 <-- x2
next i
return ``NO INTERSECTION''
```

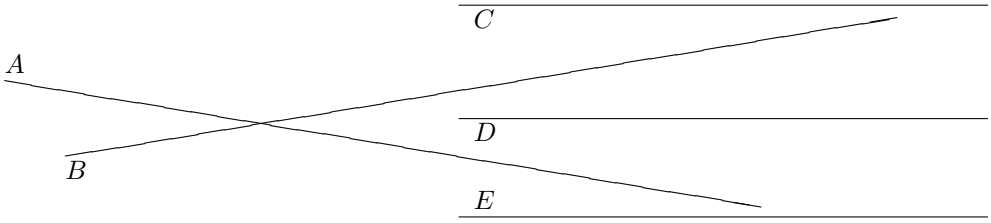
To convince ourselves that the preceding algorithm is correct, we first consider what it means for two lines to intersect. From a left-to-right perspective, it means that the order of the segments from top to bottom at some horizontal point is different from the order at another horizontal point. That is, at least two line segments have "switched" in position.

This shows that we just need to find out if the order of the line segments ever changes to know whether there is an intersection. The best time to do this is clearly when a line segment ends, because we can

now compare to the order it had when it started. So we say that when segment l ends, we check to see if it switched with other lines. However, we cannot check all of the line segments quickly. So we consider checking only the closest ones. Specifically, we want to check only the "neighbours" of l—the closest segments on either side.

This begs the question: What if a line segment which is *not* a neighbour of l crosses l? We consider that, assuming l, its neighbours, and the line crossing it have not ended, in order for a line that is not a neighbour of l to cross l, it must cross a neigbour of l first. And if that neighbour of l is not a neighbour of the crossing line, there must be another one between them. By following this logic to its conclusion, we see that the crossing line must be a neighbour of some line that is has crossed, so the crossing will be "detected" by one of them, or by the crossing line itself.

The remaining issue is the fact that the neighbours might change, if nodes are inserted or deleted. If nodes are deleted (that is, if our assumption in the previous paragraph, that l's neighbours have not ended, fails), this does not present a problem, as long as care is taken to make sure the `successor` and `predecessor` functions get the right nodes. Insert, though, seems to have potential for causing a problem. Consider the following situation:



Segments C, D, and E are inserted *after* segments A and B have crossed, but before either of them ends. It appears that A and B now have new neighbours, and will not detect a problem when the delete algorithm is run.

However, this is not in fact the case. Actually, because of the recalculation of the *start* keys, D will be placed to the right of A (it is greater than A at its start point), while B is of course on the left of A. C will be placed to the right of both, and E will be placed to left of both. This example serves to demonstrate that you simply can't get a new element between two existing elements if those elements are out of order in the tree, because the one it encounters first would send it the wrong way, in effect. Thus, those out-of-order elements are guaranteed to trigger detection of the switching when they are eventually removed.

Now that we have seen that we only need to look at the neighbouring segments upon deletes, we can observe that this is exactly what the algorithm does. If it encounters a left endpoint, it just adds that segment to the tree (remember, the *start* keys are updated every time they are checked, so it will go in the right place). If it encounters a left endpoint, it first checks to make sure it can find it in the tree. If it can't, then something in the tree is not in the right order, which means line segments have switched order. If it can find it, then it just checks its neighbours to make sure it hasn't switched with any of them, and then, if all goes well, deletes the node; it is not needed any more. Finally, at the end, if all endpoints were added and removed without incident, it reports that there were no intersections.

As for the running time, we already used time $O(n)$ to make the list of endpoints and time $O(n \log n)$ to sort them. The algorithm for looking for intersections iterates, but there are no nested loops and all the operations are either $\Theta(1)$ operations or tree operations. The tree never has more than n nodes in it, so if we pick an appropriate implementation (such as red black trees) we can get time $O(\log n)$ for each tree operation. This includes the `predecessor` and `successor` operations. Also, the operations are not added to significantly by the new method for getting the *start* keys, because that only adds $\Theta(1)$ operations to each check.

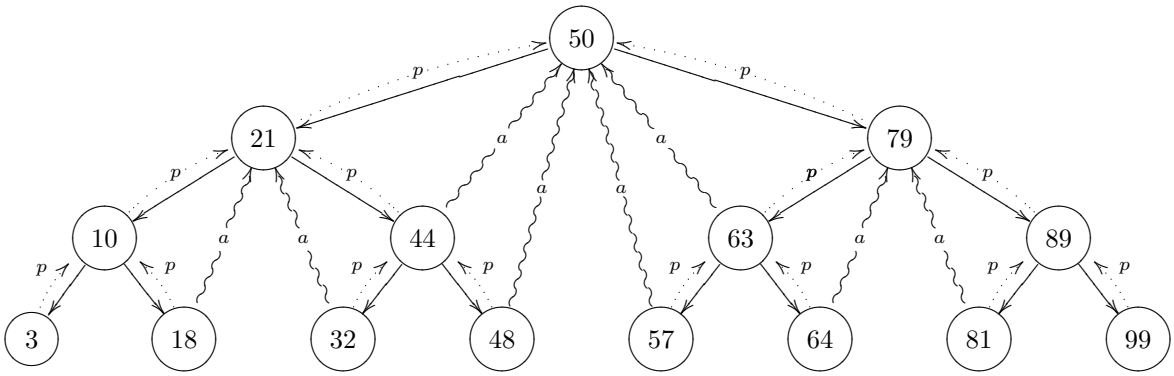
Since the algorithm iterates $2n$ times, and each time performs only $O(\log n)$ operations, the total time is $O(n) + O(n \log n) + 2nO(\log n) \in O(n \log n)$.

Question 4

To solve the least common ancestor problem, we can augment the red black trees with an additional field on each node which holds the first (deepest) ancestor of the node on the side of the node opposite that of the node's parent. For instance, if the node is a *right* child of its parent, then its ancestor field holds a pointer to the deepest node whose *left* child is an ancestor of the node.

We will call this field a, as opposed to the parent field p. We note that we do not need to distinguish which is pointing to the left and which is pointing to the right because, in $\Theta(1)$ time, we can simply check the keys of the nodes pointed to and use the fact that the one on the right must have a greater key and the one on the left must have a smaller key than that of the node itself.

The following is a picture of a tree augmented with these pointers, with parent and ancestor pointers shown, but not colouring or rank.



It is clear that, for a node n, if n is in a position where an a pointer is valid, then n's ancestor pointer a is either its parent's a or its parent's p. We can prove this by definition of a binary tree.

$p[p]$ is on one side of p, and $a[p]$ is on the other. Thus, n is either between p and $a[p]$ or between p and $p[p]$, depending on which ancestor of p is on the same side of p as n, because, if n were on the other side of one of p's ancestors, then it would not be a child of p. Since $a[p]$ is by definition the closest ancestor on one side of p, and $p[p]$ is obviously the closest ancestor on its side, there are no closer ancestors of p, and thus no closer ancestors of n (except for p itself, but it gets its own pointer). This means that one of those must be $a[n]$. So the pointer is easy to create and maintain in regular tree operations.

Now, given two nodes x and y, the following algorithm will find the least common ancestor of the two. It uses a method `up(node1, node2)`, which returns either $p[node1]$ or $a[node1]$—whichever is *in the direction of* node2. This function is given below the algorithm.

```
while x and y are not the same node do
  if rank[x] = rank[y] then
    if x is red then x <-- up(x, y)
    else y <-- up(y, x)
  else if rank[x] < rank[y] then up(x, y)
  else up(y, x)
return x

up(node1, node2):
  if key[node1] < key[node2] then
    if p[node1] is on the right of node1 then return p[node1]
    else return a[node1]
  else
    if p[node1] is on the left of node1 then return p[node1]
    else return a[node1]
```

The algorithm moves the pointers towards each other, but always ascending levels. Because the pointers they are following are the first ancestors they have in the proper direction, we know that they can't move towards each other any faster. Because they are ascending levels, we know that both pointers will at some point reach their least common ancestor.

The algorithm makes sure that one of the pointers doesn't pass by this ancestor. This would happen if a pointer reaches the least common ancestor and then advances again. If the other pointer hasn't reached the least common ancestor yet, then it must be below (at greater depth than) the second pointer. The algorithm uses this idea to prevent "passing over" the least common ancestor by only allowing the pointer with the smaller rank to advance. If they both have the same rank, it lets the one who is red move, in case it is the child of the other one (which would be black) at the time. If they are both the same color and rank (and not the same node), then one can't be the parent of the other, so either one can move safely.

Since they are both moving toward their least common ancestor, and since the algorithm never lets the higher one advance, the pointers must eventually both reach the least common ancestor.

The only remaining issue is time. We note that, if a node n has rank r, then its subtree has rank at least $r-1$. As discussed in class, a tree whose root is rank r has at least 2^r external nodes, and thus at least 2^r-1 internal nodes. Because rank can only go down at most 1 in going to a descendant, we can say that the left or right subtree of a node must have at least $2^{r-1}-1$ internal nodes.

We now consider the act of progressing towards the least common ancestor. Each time we move up a level, we "discard" all the nodes in the left and right subtrees of the node we just left. What's more, one of those two subtrees contains elements among those k elements between the two nodes x and y for whose least common ancestor we are searching. We never consider those nodes again because we only move toward the least common ancestor. So each movemement eliminates some elements from those we are considering. Since we keep moving up, r keeps increasing, so we also keep eliminating more and more elements as we go along. Once we have eliminated all the elements but 1, we are done.

The rank increases by 1 at least every second time we go up a level. So the ith time we go up a level, we must throw out at least $2^{\lfloor \frac{i}{2} \rfloor - 1} - 1$ nodes (more if we did not start from a leaf). We are done when we have thrown out $k-1$ nodes. We do the following manipulation:

$$\begin{aligned} k-1 &= 2^{\frac{i}{2}-1}-1 \\ \log_2 k &= \frac{i}{2}-1 \\ i &= 2\log_2 k + 2 \end{aligned}$$

and we see that we will definitely not need to go up $2\log_2 k + 2$ levels in a single string to get to the least common ancestor, because we would throw out at least $k-1$ nodes just on the last step; there will be fewer steps, even if we are starting with a leaf.

Considering that we will actually be going up levels on both sides (and we'd have to keep track of two different i's in the above equations), we have not completely bounded the number of steps. But we can say that, if we won't go up $2\log_2 k + 2$ steps in a single chain, no matter which side it's on, then we certainly won't have to go up that many steps on *both* sides of the least common ancestor. So we can say:

$$\begin{aligned} T &< 4\log_2 k + 4 \\ T &\in O(\log k) \end{aligned}$$