COMP251 Assignment 3

Adam Hooper
260055737

March 15, 2005

1. Exercise 12.1-5

We know that sorting n elements takes $\Omega(n \lg n)$ in the worst case. We are asked to prove that creating a binary search tree takes $\Omega(n \lg n)$ time in the worst case as well.

This is a proof by contradiction. Assume there is algorithm for constructing a binary search tree in less than $\Omega(n \lg n)$ worst-case running time. Then construct the following sorting algorithm:

FAST-SORT(L)
1 Build a BST from L in less than $\Omega(n \lg n)$ worst-case running time
2 Perform an inorder traversal to create an ordered list from the BST

Since the second step of this algorithm runs in $O(n)$ time, this sorting algorithm runs in less than $\Omega(n \lg n)$ worst-case running time. But this contradicts our claim that no comparison-based sorting algorithm runs in less than $\Omega(n \lg n)$ running time. Therefore, building a binary search tree using a comparison-based algorithm must take at least $\Omega(n \lg n)$ worst-case running time.

2. Problems 13-2: Join operation on red-black trees

a. Maintain $bh[T]$ in RB-INSERT and RB-DELETE

Within RB-INSERT we maintain property 5 of red-black trees: the path from the root of T to its leaves contains $bh[T]$ black nodes. Within RB-INSERT-FIXUP, however, this property may change: $bh[T]$ *may* increase by at most 1. This can only occur when line 16 of RB-INSERT-FIXUP, $color[root[T]] \leftarrow$ BLACK.

Prior to line 16 (i.e., in the **while** loop), the black-height of the tree is guaranteed to remain the same. This follows rather trivially from the assumptions implicit in the loop. Inside the loop, the algorithm does not know whether it is dealing with a subtree or the entire tree. So the algorithm cannot modify the black-height of the subtree: if it did so, then

the black-height of the root node would be inconsistent if other subtrees were present in the tree (which the algorithm must take into account, whether or not there are any).

Therefore, the **while** loop may not modify the black-height of the tree; only line 16 may do so. $bh[T]$ may be easily maintained: if the root of the tree is red before that step, then the black-height of the tree is about to be increased, and so $bh[T]$ should be incremented.

The next trick occurs in RB-DELETE, and it is just as straightforward. The tree can only decrease in black-height within RB-DELETE-FIXUP. This statement is easy to justify. At most one node will be spliced out during RB-DELETE. If the spliced-out node is red, then the black-height of the tree remains the same (indeed, RB-DELETE-FIXUP is never called). Otherwise, the black-height may change, but RB-DELETE-FIXUP will be called.

Within RB-DELETE-FIXUP, the textbook speaks of the current node as being either "doubly-black" or "red-and-black." This simplification makes the next statement easy to justify: as the "doubly-black" or "red-and-black" node is pushed up the tree, the black-height of the tree will only decrease if the root node is "doubly-black" before line 23 of RB-DELETE. At line 23, we set $color[x] \leftarrow$ BLACK no matter what. If x is not the root, this signifies that we have ended fixing the tree. If x *is* the root, then we are satisfying property 2 of red-black trees. But using the "doubly-black" simplification, we can easily see that if x is the root of T and is "doubly-black", we are decreasing the black-height of the tree. So within RB-DELETE-FIXUP, we can put a conditional before line 23: **if** $x = root[T]$ and $color[x] =$ BLACK, then decrement $bh[T]$.

Finally, we are asked to show that while decending through T we can determine the black-height of each node in $O(1)$ time. This is trivial: keep a counter starting at $bh[T]$ and decrement it as black nodes are reached while descending through T.

b. Find largest black node y with black-height $bh[T_2]$

LARGEST-IN-T1-WITH-BLACK-HEIGHT-T2(T_1, T_2)

```
1  x ← root[T1]
2  h ← bh[T2] − 1
3  while h > 0
4        do x ← right[x]
5           if color[x] = BLACK
6                 do h ← h − 1
7  return x
```

This algorithm runs in $O(\lg n)$ time since the height of the trees is $O(\lg n)$. We know that $right[x]$ will always exist since we are told $bh[T_1] \geq bh[T_2]$.

c. Replace T_y with $T_y \cup \{x\} \cup T_2$

Given a T_y and a T_2, and knowing that each element of T_y is less than x, which is less than each element in T_2, we can easily see how to merge the two subtrees. Simply make a new tree with root x, and set $left[x] \leftarrow root[T_y]$ and $right[x] \leftarrow root[T_2]$.

d. Maintain red-black properties

Assuming y was found as in part (b), then we know $bh[T_y] = bh[T_2]$. So no matter what the color of x, we know that properties 1, 3, and 5 will be maintained. To maintain property 2, x must be made black.

However, what we *really* want to do in RB-JOIN is join all of T_1 and T_2. To do this, we can simply replace the root of T_y with our new $T_y \cup x \cup T_2$ tree, making x red. Doing so will preserve properties 1, 3 and 5, since the black-height of the unioned tree will be the same as the black-height of T_y and T_2. But this may break properties 2 and 4.

Does this look familiar? Yes, it's exactly as if we'd called RB-INSERT! And so running RB-INSERT-FIXUP(T, x) will solve the problem. This is because the loop invariant described on page 283 is achieved. And as proven in the textbook, RB-INSERT-FIXUP runs in $O(\lg n)$ time.

e. Defending assumption in (b)

When $bh[T_1] \leq bh[T_2]$, the same situation arises. Instead of finding the subtree of T_1 with the largest root with black-height $bh[T_1]$, we will search for the subtree of T_2 with the smallest root with black-height $bh[T_1]$ (let's call it T_z). Then we will replace z in T_2 with $T_1 \cup x \cup T_z$ and call RB-INSERT-FIXUP(T, x).

f. $O(\lg n)$ running time

RB-JOIN contains only $O(1)$ operations and $O(\lg n)$ operations. Maintaining the black-height of a tree does not affect the asymptotic running-time of RB-INSERT or RB-DELETE, and so finding the black-height of a tree can be done in $O(1)$ time. Finding the largest subtree in T_1 with black-height $bh[T_2]$ takes $O(\lg n)$ time. Joining the two binary trees takes $O(1)$ time, and running RB-INSERT-FIXUP on the result takes $O(\lg n)$ time. Therefore, the entire running time is $O(\lg n)$.

3. Problem 13-3: AVL trees

a. Prove an AVL tree with n nodes has height $O(\lg n)$

Let us find the minimum number of nodes in an AVL tree of height h. For $h = 0$, $n = 0$. For $h = 1$, $n = 1$. And to find the least number of nodes in an AVL tree of height $h > 1$, we can recurse: $N_h = 1 + N_{h-1} + N_{h-2}$: 1 for the root node, N_{h-1} because one of the subtrees must have height $h - 1$ for the height of the tree to be h (by definition of height), and N_{h-2} because the other subtree (which we'd want to have the fewest nodes possible) cannot have height less than $h - 2$.

So N_h grows faster than the Fibonacci sequence when $h > 2$. This means the most unbalanced tree possible of height h has more than F_h nodes. Liberally applying inequalities,

we can see that $F_h \geq 2F_{h-2}$ and so $F_h \geq 2^{\lfloor \frac{h}{2} \rfloor}$. So in a tree of height h, there are at least $2^{\lfloor \frac{h}{2} \rfloor}$ nodes.

$$\begin{aligned} n &\geq 2^{\lfloor \frac{h}{2} \rfloor} \\ \lg n &\geq \left\lfloor \frac{h}{2} \right\rfloor \\ 2 \lg n + 1 &\geq h \end{aligned}$$

And so h grows in order $O(\lg n)$.

b. Balance

```
BALANCE(x)
 1  T ← tree[x]
 2  if left[x] = NIL and right[x] ≠ NIL and h[right[x]] = 1
 3     then h[x] ← 0
 4          LEFT-ROTATE(T, right[x])
 5  elseif right[x] = NIL and left[x] ≠ NIL and h[left[x]] = 1
 6     then h[x] ← 0
 7          RIGHT-ROTATE(T, right[x])
 8  elseif left[x] ≠ NIL and right[x] ≠ NIL
 9     then if h[left[x]] = h[right[x]] + 2
10             then h[x] ← h[right[x]] + 1
11                  RIGHT-ROTATE(T, left[x])
12          elseif h[right[x]] = h[left[x]] + 2
13             then h[x] ← h[left[x]] + 1
14                  LEFT-ROTATE(T, right[x])
```

After rotation, we know that $h[left[x]] = h[right[x]]$; so $h[x]$ is simply one greater than that.

c. AVL-Insert

```
AVL-Insert(x, z)
 1  if key[z] ≤ key[x]
 2     then if left[x] = NIL
 3             then left[x] ← z
 4                  p[z] ← x
 5                  h[x] ← 1 ▷ will be corrected by Balance if incorrect
 6             else AVL-Insert(left[x], z)
 7     else if right[x] = NIL
 8             then right[x] ← z
 9                  p[z] ← x
10                  h[x] ← 1
11             else AVL-Insert(right[x], z)
12  Balance(x)
```

d. Running time of AVL-Insert

Observe the following:

- Balance runs in $O(1)$ time (since Left-Rotate and Right-Rotate run in $O(1)$ time).
- In each recursive step, AVL-Insert is called on a node with a height of at least 1 less than the height of the current node.
- AVL trees have a height in the order of $O(\lg n)$.

Since AVL-Insert does not loop, we can easily deduce that it runs in $O(\lg n)$ time.

Next, we are to prove that at most $O(1)$ rotations are made in AVL-Insert. Imagine a case in which Balance is necessary on a subtree T_x. The subtree contains a child subtree T_y with height h and another child subtree T_z with height $h-2$. Therefore the height of T_x is $h+1$. This situation could only arise if the height of T_y had been $h-1$ before x was added to T_y with AVL-Insert (deduced from the definition of AVL trees). In other words, the height of T_x had been h before the additional node was added. After rotation, the height of the entire subtree (now rooted at T_y's root) will be h; therefore the heights of all higher subtrees will be valid. This means no more than one rotation will be made per insertion.

4. Problem 15-4: Planning a company party

To solve the large problem of maximizing conviviality, we must first solve the sub-problems. From a given tree of height 2, we may take one of two approaches:

- Add the maximum convivialities of each subtree; or

- Add the root's conviviality to the maximum convivialities of each sub-subtree.

The latter may give us a larger sum than the former, but because of the problem's restrictions it may cause the higher-up problems to give smaller sums; therefore, we must store both of these numbers for each sub-tree.

At each step of the problem, we must also store the list of names which make up the conviviality ratings. Let us therefore define, for a node n, the properties $names[n]$ and $convsum[n]$ to be the list of names and sum of convivialities for a node; let $namessub[n]$ and $convsumsub[n]$ be the list of names and sum of convivialities for a node when ignoring direct children. Let us assume each node n already has the properties $name[n]$ and $conv[n]$, representing the name of a particular employee and his conviviality. The algorithm should look like this:

PLAN-PARTY(T)

```
 1  for d ← depth[T] downto 1
 2        do
 3           for n ← first node of depth d to last node of depth d
 4                 do if n is a leaf
 5                       then namessub[n] ← {}
 6                            convsumsub[n] ← 0
 7                            names[n] ← {name[n]}
 8                            convsum[n] ← conv[n]
 9                       else tmpnames ← {}
10                            tmpsum ← 0
11                            namessub[n] ← {}
12                            convsumsub[n] ← 0
13                            for m ← first child of n to last child of n
14                                  do namessub[n] ← namessub[n] + names[m]
15                                     convsumsub[n] ← convsumsub[n] + convsum[m]
16                                     tmpnames ← tmpnames + namessub[m]
17                                     tmpsum ← tmpsum + convsumsub[m]
18                            if tmpsum + conv[n] > convsumsub[n]
19                              then convsum[n] ← tmpsum + conv[n]
20                                   names[n] ← tmpnames + {name[n]}
21                              else convsum[n] ← convsumsub[n]
22                                   names[n] ← namessub[n]
23  return names[root[T]]
```

The running time of this algorithm will be $O(n)$. That is because it iterates over each node of the tree a fixed number of times: on each node, *namessub*, *convsumsub*, *names* and *convsum* are written and read exactly once, because after each iteration of the loop all sub-children are ignored forever.

There is some slight hand-waving in this claim of $O(n)$ running time: how do we iterate over all elements of the tree in the order defined (highest depth to lowest)? Well, we could assume that we are given a structure which already allows us to iterate in such a manner. In general, though, it can still be done without affecting asymptotic running time. What follows is as a simple, messy sample algorithm: allocate an array of size $depth[T]$, each element corresponding to a list, and then traverse the tree, inserting the nodes into the appropriate lists (like bucket sort). PLAN-PARTY could then use this array of lists for its iterations. Since this algorithm would take place outside of PLAN-PARTY and since it is $O(n)$ as well (it is a tree traversal), the entire running time would remain $O(n)$.

5. Problem 15-6: Moving on a checkerboard

As with the assembly-line problem described in class, the checkerboard problem is easiest to solve backwards. We will create an $n \times n$ grid called P to hold dollar sums and an $n \times (n-1)$ grid called M to hold moves. $P_{x,y}$ will store the maximum dollar sum one could achieve if starting on the board from square (x, y). $M_{x,y}$ will store the next move required to achieve that sum.

The solution is straightforward and extremely similar to the assembly-line problem in class, so instead of trying to explain I will simply let the code speak for itself:

```
RICHEST-PATH(p)
 1  P ← n × n matrix of 0s
 2  M ← n × (n − 1) uninitialized matrix
 3  for r ← n − 1 downto 1
 4        do for c ← 1 to n
 5              do P_{c,r} ← P_{c,r+1} + p((c, r), (c, r + 1))
 6                 M_{c,r} ← UP
 7                 if c > 1 and P_{c−1,r+1} + p((c, r), (c − 1, r + 1)) > P_{c,r}
 8                    then P_{c,r} ← P_{c−1,r+1} + p((c, r), (c − 1, r + 1))
 9                         M_{c,r} ← UP-LEFT
10                 if c < n and P_{c+1,r+1} + p((c, r), (c + 1, r + 1)) > P_{c,r}
11                    then P_{c,r} ← P_{c+1,r+1} + p((c, r), (c + 1, r + 1))
12                         M_{c,r} ← UP-RIGHT
13  c ← 1
14  m ← P_{1,1}
15  for i ← 2 to n ▷ Find the start column c
16        do if P_{i,1} > m
17              then m ← P_{i,1}
18                   c ← i
19  PRINT(Starting square: (c, 1))
20  for r ← 1 to n ▷ Print the list of moves
21        do PRINT(Next move: M_{c,r})
22           if M_{c,r} = UP-LEFT
23              then c ← c − 1
24           elseif M_{c,r} = UP-RIGHT
25              then c ← c + 1
```

(The double loop in RICHEST-PATH may end up setting a value to $P(c, r)$ three times, which is rather inelegant. The algorithm could obviously be adjusted, but that would make it less legible. Also, making P an $n \times n$ matrix is rather overkill: really, only two rows are accessed at a time so the algorithm could use an $n \times 2$ matrix instead. But that would make the algorithm slightly more confusing.)

The running time analysis is simple. RICHEST-PATH is divided into three main sections: one to populate the P and M matrices, one to find the starting row, and one to print the list of moves. The first part is $O(n^2)$ since it contains a loop within a loop, both of size n. The second part is $O(n)$ and the third part is $O(n)$. Therefore, the running time of this algorithm is $O(n^2)$.