# COMP 252 Assignment 2

Christopher Hundt (110220945)

February 11, 2003

## Question 1

Since our drawing area is wider than it is high, it is obvious that the longest strings must run horizontally in our hv-drawing. Also, we can plan that, if a tree has only a left subtree or only a right subtree, the subtree should go to the right of the tree, not below it. In order to draw longer strings horizontally, we need to determine the comparitive length of the subtrees in our tree. It is convenient to use a kind of size attribute, but a special one. We consider the following reasoning, based on the number of columns $h$ ("horizontal size") that a tree $t$ requires in the hv-drawing:

- If $t$ is a leaf, then $h = 1$; it will require 1 column to show the leaf, for it is just one node.
- If $t$ has only one child, then that child will be drawn directly to the right of $t$. But $t$ takes up one horizontal unit for its own node. So $h = h_{subtree} + 1$, where $h_{subtree}$ is the horizontal size of whichever child $t$ has.
- If $t$ has two children, then one will be drawn to the right and one below. Each child will have its own width. However, because $t$'s node will be directly above one of its children's nodes, $t$'s node will not need its own column. So $h = h_{child_1} + h_{child_2}$.

We now define an algorithm to traverse the tree (starting at the root), setting our special attribute "hsize" at every node.

```
setsize(t)
  if t is a leaf then
    hsize[t] <-- 1
  else if t.left is null then
    hsize[t] <-- setsize(right[t]) + 1
  else if t.right is null then
    hsize[t] <-- setsize(left[t]) + 1
  else
    hsize[t] <-- setsize(left[t]) + setsize(right[t])
  return size[t]
```

Because each node (except the root, which is not a child) can be either the left child or right child (but not both) of exactly one other node, this algorithm cannot visit a node more than once. It performs a finite number of $\Theta(1)$ operations at each node. Clearly, this algorithm takes time $O(n)$ (where $n$ is the number of nodes in the tree).

Now that we have set the hsize atrribute of every node, we can continue to the algorithm for actually drawing the hv-trees. The following algorithm would be called as `hvtree(root, x, y)`, where `x` and `y` are the coordinates of the top-left corner of the drawing grid. It assumes that x values of the coordinate system increase as you go from left to right, and the y values increase as you go from bottom to top. It also assumes that there exists some external, $O(1)$ procedure `draw(x, y)` which draws a node at an arbitrary point $(x, y)$ on the coordinate system.

```
hvdraw(t, x, y)
  draw(x, y)
  if t is a leaf then end
  else if left[t] is null then
    hvdraw(right[t], x + 1, y)
  else if right[t] is null then
    hvdraw(left[t], x + 1, y)
  else
    if hsize[left[t]] > hsize[right[t]] then
      hvdraw(right[t], x, y-1)
      hvdraw(left[t], x + hsize[right[t]], y)
    else
      hvdraw(left[t], x, y-1)
      hvdraw(right[t], x + hsize[left[t]], y)
```

Like the `setsize` algorithm, this algorithm visits every node once, and there are no loops. At each "visit," it performs only recursive calls, checking for null, calls to the $O(1)$ `draw` procedure, and arithmetic. Its time is thus clearly $O(n)$. So the total time to run both algorithms is $O(n) + O(n) = O(n)$.

Covering rectangles of the subtrees will not overlap, because any tree that is drawn to the right of its parent is offset by the width of the parent's other child.

The only remaining issue is analysis of the physical size of the resulting hv-drawing. Is its width no more than $n$ and is its height no more than $\lceil \log_2 n \rceil$?

## Width

The `hvdraw` algorithm uses the numbers found by the `setsize` algorithm to determine where to draw each subtree. `setsize` uses the aforementioned recursive definition of the horizontal size of a subtree. So let us prove that the total width of the resulting drawing of a tree of size (number of nodes) $n$ will not be more than $n$.

If we draw a single leaf, it will be placed by itself in a column. It takes up one column and $n = 1$. Our assertion holds.

We now assume that a subtree of size $n$ has a drawing of width no more than $n$. There are now two ways to build on this base case:

- We consider a node with one child of size $n$ (thus, the child has width no more than $n$). Total size is $n + 1$. We draw the child directly next to the node. The width is increased only by 1, so it is clearly no more than $n + 1$.
- Finally, we consider a node with two children, one of size $m$ and one of size $n$. We say that they have drawing widths $a$ and $b$, $a \leq b$. Total size is $m + n + 1$. We draw the child of size $m$ directly below the node, and the child of size $n$ to the right of the node, offset by $a$. So the total width of the drawing is $a + b$. Since $a \leq m$ and $b \leq n$, $a + b \leq m + n + 1$ and our assertion holds.

## Height

The only time the algorithm goes down a level (increases the height by 1) is when it reaches a node with two children. However, it always draws the *smaller* node at the lower level. Since the size (number of nodes) of a subtree with two children is defined as the sum of the sizes of its two children, plus 1, the size of the smaller child is less than half the size of the current subtree. Thus, each time the height of the hv-drawing increases, the size of the subtree being dealt with is halved. So, if the total size of the tree is $n$, and you start with height of 1 for a single node (the root), the maximum size of a subtree at any level $l \geq 1$ is $n/\left(2^{l-1}\right)$. At $l = \lceil \log_2 n \rceil$, the maximum size of a subtree is $n/\left(2^{\lceil \log_2 n \rceil - 1}\right) = n/\left(2^{-1} 2^{\lceil \log_2 n \rceil}\right) \leq 2$. A subtree of size 2 has only one child, which can be drawn to its right (thereby not increasing the height of the drawing). So the drawing's height will not exceed $\lceil \log_2 n \rceil$. The one exception is when $n = 1$. Obviously, even though $\log_2 n = 0$, we will still need a grid height of 1.

# Question 2

First, we determine what information needs to be known in order to produce a proper sorting of the elements. First of all, the order of the $k$ different values must be known. Without finding this information, an algorithm cannot hope to sort the $n$ elements. Finding the order of $k$ values is equivalent to sorting $k$ unique elements, which has $k!$ possible answers.

The next issue is that of the "duplicate" elements. Obviously, these cannot be put just anywhere. However, there is more than one correct position for each duplicate element. For instance, if there are two elements of equal value, those two can be switched after sorting without affecting the correctness of the sort. Thus, there is no unique positioning of elements in the sorted output that the algorithm must know. Rather, it simply needs to know, for each duplicate element, which of the $k$ unique elements it is equal to. Once the equal values are grouped together in some way, and the unique values sorted, the correct answer can be output without any more comparisons. For each of the $n-k$ duplicate elements, there are $k$ possible unique elements that the duplicate could be equal to, leading to $k^{n-k}$ possible groupings for each sorting of the $k$ unique values. Thus, the total number of possible answers is

$$k!k^{n-k}.$$

By the information-theoretic method, and considering our oracle to be comparison, which has two possible answers, the minimum number of comparisons is thus

$$\begin{aligned} & \lceil \log_2 \left(k!k^{n-k}\right) \rceil \\ = \; & \lceil \log_2 k! + \log_2 k^{n-k} \rceil \\ = \; & \lceil \log_2 k! + (n-k)\log_2 k \rceil \\ \sim \; & \lceil k\log_2 k + n\log_2 k - k\log_2 k \rceil \\ = \; & \lceil n\log_2 k \rceil . \end{aligned}$$

# Question 3

For this task we can use a binary tree with the following properties:

1. All the leaves are at the lowest level (this implies that the number of leaves must be a power of 2).
2. All the $n$ integers are stored at leaves on the tree, in order, with the leftmost leaf holding $T[1]$ and the rightmost integer-storing leaf holding $T[n]$.
3. If $n$ is not a power of 2, then empty leaves are added until the number of leaves is a power of 2.
4. The value stored at each non-leaf node `t` is the sum of the integers at all the leaves whose path to the root includes `left[t]`.

So the number of leaves would be $m = 2^k$ for some $k$, the total number of nodes would be $2^{k+1} - 1$, and the number of levels in the tree would be $k + 1$. As an example, if $n = 8$ (and thus $m = 8$), and the nodes are numbered, starting with the root, going left to right and then down (so the root is 1, the root's left child is 2, the root's right child is 3, and so on), then the first 7 nodes will have the following values:

1. $\sum_{i=1}^{4} T[i]$
2. $\sum_{i=1}^{2} T[i]$
3. $\sum_{i=5}^{6} T[i]$
4. $\sum_{i=1}^{1} T[i] = T[1]$
5. $\sum_{i=3}^{3} T[i] = T[3]$
6. $\sum_{i=5}^{5} T[i] = T[5]$
7. $\sum_{i=7}^{7} T[i] = T[7]$

Nodes 8 through 15 would have $T[1]$ through $T[8]$ as their values.

Updating an entry and finding a partial sum could be done with the following algorithms:

```
sum(k)
  pointer <-- root
  sum <-- 0
  continue <-- false
  i <-- 1
  minvalue <-- 0
  while continue is false do
    if isleaf(pointer) then
      sum <-- sum + value[pointer]
      continue <-- true
    else if k = minvalue + m/(2^i) then
      sum <-- sum + value[pointer]
      continue <-- true
    else if k > minvalue + m/(2^i) then
      sum <-- sum + value[pointer]
      pointer <-- right[pointer]
      minvalue <-- minvalue + m/(2^i)
      i <-- i + 1
    else
      pointer <-- left[pointer]
      i <-- i + 1
  return sum
```

```
update(i, x)
  pointer <-- leaf associated with element i
  difference <-- x - value[pointer]
  value[pointer] <-- x
  while parent[pointer] is not null do
    if left[parent[pointer]] = pointer then
      value[parent[pointer]] <-- value[parent[pointer]] + difference
    pointer <-- parent[pointer]
```

the `sum` algorithm works by descending the tree, and checking at each level to see if the value stored by the current node is included in the sum it is trying to find. For isntance, if it were trying to find $S_7$ when $n = 8$ (and thus $m = 8$ as well), it would follow these steps:

1. (at the root) $k$ (7) is greater than $\frac{m}{2^1}$ (4), so add the value at the root ($\sum_{i=1}^{4} T[i]$), set *minvalue* to 4, and proceed to the right child node of the root.
2. $k$ is greather than $minvalue + \frac{m}{2^2}$ (6), so add the value at this node ($\sum_{i=5}^{6} T[i]$), set *minvalue* to 6, and proceed to the right child node.
3. $k$ is equal to $minvalue + \frac{m}{2^3}$ (7), so add the value at this node ($T[7]$) and terminate.

The `update` algorithm ascends the tree, starting with a leaf, and makes its way up to the root, updating each node's value if its path up the tree included that node's left child. This makes the tree match the aforementioned definition.

Both algorithms perform a set maximum number of operations in each iteration of their loops, and both visit no more than one node at each level of the tree, and no nodes twice. Each line of each algorithm can be said to have time $\Theta(1)$. So the time for both is $O(k)$, where $k + 1$ is the number of levels in the tree. Since $m = 2^k$, $k = \log_2 m$. Also, $m \geq n > \frac{m}{2}$ (because $m$ is the smallest power of 2 greater than or equal to $n$), so

$$\begin{aligned} \log_2 m \geq \quad & \log_2 n \quad > \log_2 \frac{m}{2} \\ k \geq \quad & \log_2 n \quad > k - 1 \end{aligned}$$

Thus, the worst-case running time for both of these algorithms is $O(\log n)$.

Note that there were several assumptions made in the algorithms about the data structure used, the most notable one being the ability to find the leaf associated with any single element in time $\Theta(1)$. This, as well as the ability to find parents, children, and the root in time $\Theta(1)$ can be provided by using an array to represent the tree. Because the tree used is complete, we can number the nodes (as discussed earlier), and have the following properties (we call the array $A$, and index it starting with 1):

- $parent[i] = \lfloor \frac{i}{2} \rfloor$
- $left[i] = 2i$
- $right[i] = 2i + 1$
- $root = 1$
- $T[i] = A[m - 1 + i]$

This would also make the complexity of the data structure quite low. An array of size $2m - 1 < 4n$ is all that would be needed.

One final note: it is true that, for all odd $i$, $T[i]$ appears twice on the tree—once at its associated leaf, and once at the leaf's parent (because it is a left child). The `sum` algorithm never gets down to the leaf level for such elements (see the example above). This is clearly a waste, so it would improve the data structure to change the definition of the tree so that odd-numbered elements do not appear on leaves; they could appear on nodes in the second lowest level, each of which would have only one child (the even-numbered element that comes after the associated odd-numbered one). This complicates the tree somewhat, though, and provides a very small improvement, and only half the time, so I have chosen to leave it out of my basic definition.

# Question 4

We can make a simple non-recursive algorithm with two stacks to solve this problem.

```
S1, S2 <-- new stacks
t <-- root
while S1 is not empty or t is not null do
  if t is null then
    t <-- pop(S1)
  if t is a leaf then
    size[t] <-- 1
    t <-- null
  else if left[t] is null then
    push(t, S2)
    t <-- right[t]
  else if right[t] is null then
    push(t, S2)
    t <-- left[t]
  else
    push(t, S2)
    push(right[t], S1)
    t <-- left[t]
  t <-- left[t]

while S2 is not empty do
  t <-- pop(S2)
  if left[t] is null then
    size[t] <-- 1 + size[right[t]]
  else if right[t] is null then
    size[t] <-- 1 + size[left[t]]
  else
    size[t] <-- 1 + size[left[t]] + size[right[t]]
```

This algorithm has two loops. The first one goes through every node, and each node does one of three things:

1. If the node is a leaf, set its size to 1.
2. If the node has only one child, add that node to the stack of nodes whose size must be determined later (`S2`), and go on to its child.
3. If the node has two children, add the node to `S2`, add one child to the stack of nodes to be dealt with later in the same loop (`S1`), and go on to the other child.

Each node that is added to `S1` is dealt with in the same way and then removed from the stack. Every node must be visited exactly once, because we start with the root and visit the children of each node once. So the time is linear with $n$, because a set number of operations are performed in each iteration, and the number of iterations is equal to the number of nodes.

At the end of the first loop, all the nodes which are leaves have been given a size of 1, and all other nodes have been added to `S2`. The second loop goes through `S2`, and sets each node's size to the sum of the sizes of its children, plus 1. In the first loop, because it works its way from the top down through the tree, it is clear that, for any node with a child, either the child was a leaf and was assigned size 1, or it was added to `S2` later and will be assigned a size before its parent (because stacks are last-in, first-out). So the size attribute will be defined for the children of any node before the second loop gets to that node in the stack.

The time for the second loop is $O(n)$, because the number of nodes on the stack is less than $n$, and a set number of operations are performed on each item in the stack before it is "discarded." So the total time is $O(n) + O(n) = O(n)$.