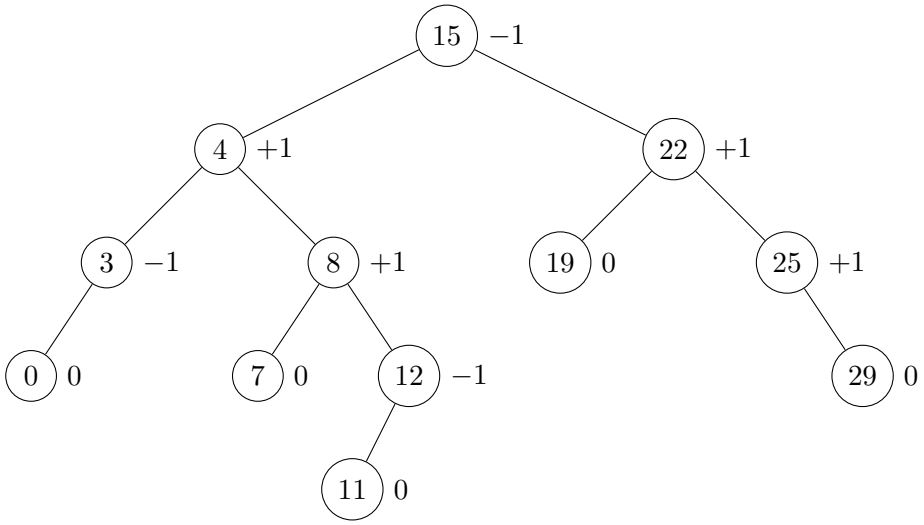
Assignment 2

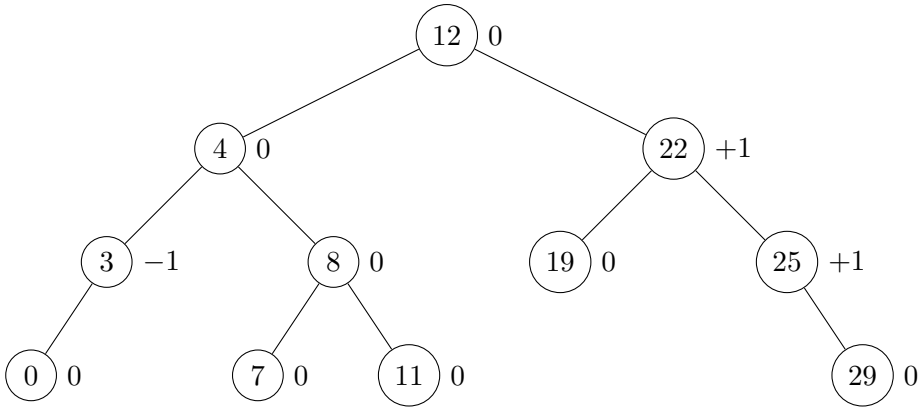
Michael Kozakov

Feb 10, 2013

1. a)



b)



2. (a) Let n be the total number of nodes in T1 and T2
   (b) Conduct an in-order traversal on T1 and T2 ($O(n)$) and save the two sorted lists in l_1 and l_2
   (c) merge the two sorted lists into one sorted list using the MERGE function of the merge sort $O(n)$
   (d) if the merged list is empty do nothing
   (e) let k be the element at floor(list.length /2)

(f) otherwise take k and make it the root of an AVL tree. Take every element to the left of k, run this algorithm starting with step d on that list, and attach the returned subtree on the left of the root $O(1)$.

(g) take everything on the right of k and run this algorithm starting with step d on that list, and attach the returned subtree on the right of the root $O(1)$.

(h) return the root

(i) Since the tree constructed from the elements on the left of k is ideally hight balanced, and the tree constructed from the elements on the right of k is also ideally height balanced (can be shown by induction), the tree rooted at k is also ideally height balanced.

3. a) For each node we can store the number of nodes in its subtree ($numNodes$), and the sum the values of each node in its subtree(sum). Then to calculate the average for any subtree we just perform one arithmetic operation, dividing the sum of values by the number of nodes. That takes $\theta(1)$.

   b) INSERT: after inserting an new node we need to go up the tree from the node all the way to the root and for each node on the way ($logn$ nodes) we increment $numNodes$ by one, and change sum to the value + sum of left child + sum of right child. The arithmetic operation ($O(1)$) is performed $logn$ times so the complexity is $O(logn)$

   DELETE: When deleting a node, we need to go up the tree from the node all the way to the root and for each node on the way ($logn$ nodes) we decrease $numNodes$ by one, and change sum to the value + sum of left child + sum of right child. The arithmetic operation ($O(1)$) is performed $logn$ times so the complexity is $O(logn)$

   [a)]

4.

5. When inserting a new node into a binomial heap, the number of node comparisons is equal to the number of number of new edges that will be created.

6. The number of edges before the insertion of k elements is $n - \alpha(n)$

7. The number of edges after the insertion of k elements is $(n + k) - \alpha(n + k)$

8. So the average cost of insertion is

$$\frac{((n+k) - \alpha(n+k)) - (n - \alpha(n))}{k} = \frac{k - \alpha(n+k) + \alpha(n)}{k}$$

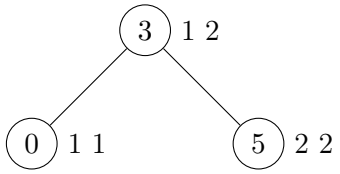
9. BY PERFORMING INEQUALITIES, SHOW THAT THIS IS LESS THAN A CONSTANT when $k > logn$

10. Therefore the average case is $\Omega(1)$

a) The key of each node represents the index of an available fragment. With each node we want to store some additional information: the size of the fragment (fragSize), and the size of the largest available fragment in that subtree (maxSize). Here is an illustration of a block of size 7, with an occupied fragment of size 2 at index 1, and another occupied fragment of size 1 at index 4:

Table 1: Memory diagram

0	1	2	3	4	5	6
Free	Taken	Taken	Free	Taken	Free	Free



In the tree we show the three unoccupied fragments at indexes 0, 3 and 5. Each is accompanied by the size of the available fragment at that index, and the size of the largest available fragment in its subtree.

b) ALLOCATE(root, blockSize). We are going to be using the First-fit approach. The algorithm is as follows:

(a) If maxSize of the root is less than blockSize, return "reject"

(b) Otherwise, attempt to allocate on the left child (if there is one).

(c) If allocate on the left child returned "reject", check if fragSize at the root is larger than or equal to blockSize. If that's the case, update fragSize to fragSize - blockSize. If the new fragSize is 0, remove that node from the tree by performing the AVL DELETE function.

(d) If the root fragSize was too small to host blockSize, allocate on the right child.

(e) After successful allocation, update the maxSize of the root to be the largest of the maxSize of the left child, the maxSize of the right child or the fragSize of the root.

ALLOCATE is $O(logn)$ because it essentially performs AVL FIND ($O(logn)$), AVL DELETE ($O(logn)$) and some constant work updating the maxSize of the root node.

RELEASE(root, (a,l).

1. Insert a node with the key a into the AVL tree, using the AVL INSERT function.

2. For that node, set fragSize to l. Set maxSize to the largest of l, maxSize of the right child (if there is one) or maxSize of the left child (if there is one).

3. let *toConsolidate* be the node with the key a.

4. let $left$ be the rightmost leaf in the left subtree of *toConsolidate*, $left$.index be its key, $left$.fragSize be its fragSize.

5. let $right$ be the leftmost leaf in the right subtree of *toConsolidate*, $right$.index be its key, $right$.fragSize be its fragSize.

6. if $left$ exists, and $left$.index + $left$.fragSize == a, then set $left$.fragSize and $left$.maxSize to $left$.fragsize + l. Go up from $left$ all the way to the root, and update maxSize for every node on the way. Perform AVL DELETE on *toConsolidate*. Assign $left$ to toConsolidate

7. if $right$ exists, and *toConsolidate*.index + *toConsolidate*.fragSize == $right$.index, then set *toConsolidate*.fragSize to *toConsolidate*.fragSize + $righ$.fragSize. Go up from *toConsolidate* all the way to the root, and update maxSize for every node on the way. Perform AVL DELETE on $right$

RELEASE is $O(logn)$ because it performs AVL INSERT once ($O(logn)$), AVL DELETE ($O(logn)$) and then does constant work of updating the maxSize for at most $logn$ nodes (the height of the tree, $O(logn)$