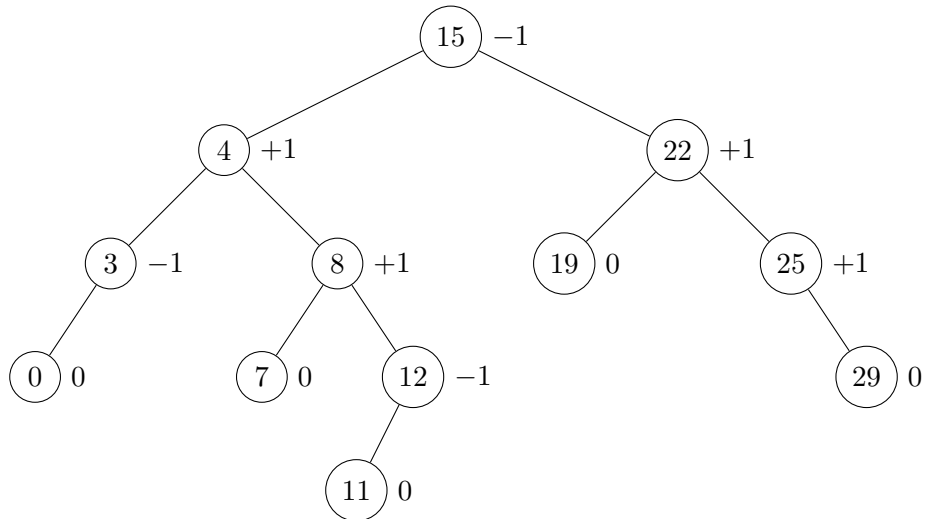


# Assignment 2

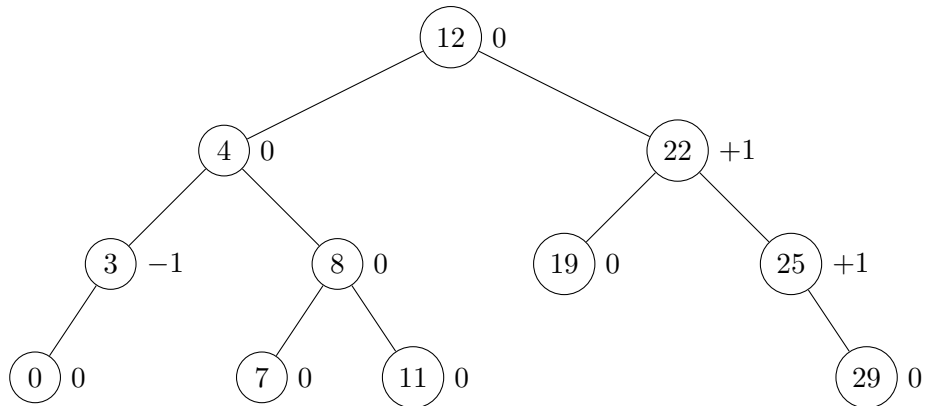
Michael Kozakov

Feb 10, 2013

1. a)



b)

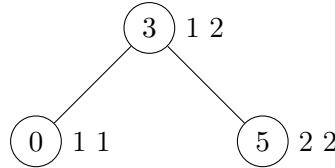


2. (a) Let  $n$  be the total number of nodes in  $T_1$  and  $T_2$
- (b) Conduct an in-order traversal on  $T_1$  and  $T_2$  ( $O(n)$ ) and save the two sorted lists in  $l_1$  and  $l_2$
- (c) merge the two sorted lists into one sorted list using the MERGE function of the merge sort  $O(n)$
- (d) if the merged list is empty do nothing
- (e) let  $k$  be the element at  $\text{floor}(\text{list.length} / 2)$

- (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 height 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 ( $\log n$  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  $\log n$  times so the complexity is  $O(\log n)$
- 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 ( $\log n$  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  $\log n$  times so the complexity is  $O(\log n)$
4. (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:

- i. If *maxSize* of the root is less than *blockSize*, return "reject"
- ii. Otherwise, attempt to allocate on the left child (if there is one).
- iii. 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.
- iv. If the root *fragSize* was too small to host *blockSize*, allocate on the right child.
- v. 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  $O(\log n)$  because it essentially performs AVL FIND ( $O(\log n)$ ), AVL DELETE ( $O(\log n)$ ) and some constant work updating the *maxSize* of the root node.

(c) RELEASE(*root*, (*a*,*l*)).

- i. Insert a node with the key *a* into the AVL tree, using the AVL INSERT function.
- ii. 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).
- iii. let *toConsolidate* be the node with the key *a*.
- iv. let *left* be the rightmost leaf in the left subtree of *toConsolidate*, *left.index* be its key, *left.fragSize* be its *fragSize*.
- v. let *right* be the leftmost leaf in the right subtree of *toConsolidate*, *right.index* be its key, *right.fragSize* be its *fragSize*.
- vi. 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*
- vii. if *right* exists, and *toConsolidate.index* + *toConsolidate.fragSize* == *right.index*, then set *toConsolidate.fragSize* to *toConsolidate.fragSize* + *right.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  $O(\log n)$  because it performs AVL INSERT once ( $O(\log n)$ ), AVL DELETE ( $O(\log n)$ ) and then does constant work of updating the *maxSize* for at most  $\log n$  nodes (the height of the tree,  $O(\log n)$ )