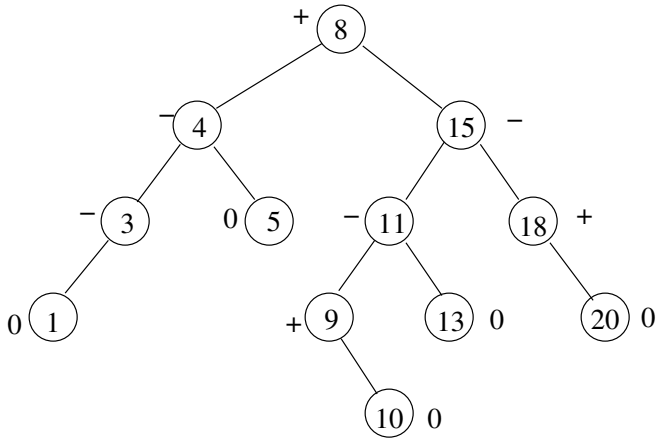
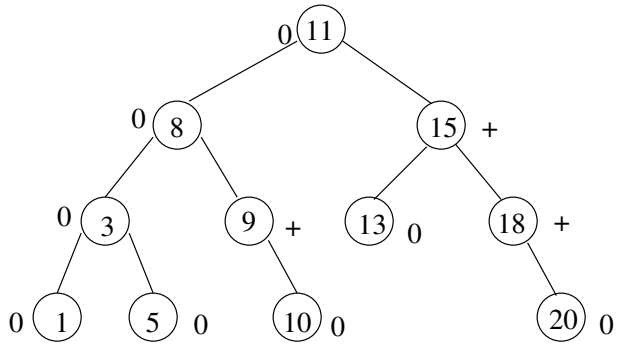
Solutions for Homework Assignment #3

Answer to Question 1. [Simple AVL]



After all the insertions



After the deletion of 4

Answer to Question 2. [Augmented AVL] The set S can be implemented as an augmented AVL tree T. Each node u of T contains the fields:

- key: contains the integer i that the node u is associated with;
- size: contains the number of nodes in the subtree rooted at u, including the key of u itself; this field satisfies the identity
$$\text{size}(u) = \text{size}(\text{lchild}(u)) + 1 + \text{size}(\text{rchild}(u)) \tag{1}$$
- sum: contains the sum of the keys of all the nodes in the subtree rooted at u, including the key of u itself; this field satisfies the identity
$$\text{sum}(u) = \text{sum}(\text{lchild}(u)) + \text{key}(u) + \text{sum}(\text{rchild}(u)) \tag{2}$$
- lchild, rchild, parent: pointers respectively to the left, right child, and parent of u;
- BF the balance factor of u.

Note that key(u) is the key of the AVL tree, i.e. an in-order traversal of T gives the elements of S in non-decreasing order.

To implement ADD(i):

1. using the usual BST search algorithm, search for a node in T with key i; if such a node exists, quit;
2. prepare a node u with fields key(u) = i, size(u) = 1, and sum(u) = i, and perform ordinary BST insertion of u, after which u is a leaf of T;
3. go up the path from u to the root of T, updating the size and sum fields of every node along the way using the identities (1) and (2); note that after this step, the "size" and "sum" fields of every node now satisfies the identities (1) and (2), respectively, but the tree may be unbalanced;

4. go up the path from u to the root of T, again, and perform rotations and update balance factors as in the usual AVL insertion algorithm, and use the identities (1) and (2) to also update the size and sum fields of the nodes involved in each rotation along the way.

The running time of the ADD(i) operation is equal to the running time of the ordinary AVL insertion procedure plus the additional time to update the size and sum fields. The number of updates to the size and sum fields is bounded by a constant times the depth of the tree T, and each update takes constant time. Since T is an AVL tree, its depth is $O(\log n)$, so the total running time of ADD(i) is bounded by $O(\log n)$.

To implement the AVERAGE(t) operation, we first define a procedure SIZEANDSUM(t, u) that returns a tuple (size,sum) where:

- u is (a pointer to) a node of T.
- "size" is the *number* of all the elements with keys at most t *in the subtree of* T *rooted at* u.
- "sum" is the *sum of the keys* of all the elements with keys at most t *in the subtree of* T *rooted at* u.

The SIZEANDSUM(t, u)procedure is given by the following pseudocode:

```
SizeAndSum(t, u)
    if u = Nil then return (0, 0)
    if t < key(u) then
        return SizeAndSum(t, lchild(u))
    else                    (* key(u) ≤ t *)
        (rigthsize,rightsum) := SizeAndSum(t, rchild(u))
        size := size(lchild(u)) + 1 + rightsize
        sum := sum(lchild(u)) + key(u) + rightsum
        return (size, sum)
    end if
```

With the above procedure, it is easy to implement the AVERAGE(t) procedure as follows. Let r be a pointer to the root of T; first execute (size, sum) := SIZEANDSUM(t, r), and then return $\frac{\text{sum}}{\text{size}}$.

Note that SIZEANDSUM(t, u) performs a constant number of operations, and recurses on *at most one* child node of u. So the depth of the recursion is bounded by the height of the node u, which is $O(\log n)$, since u is a node in an AVL tree. Therefore, the total running time of SIZEANDSUM(t, u) is $O(\log n)$.

Answer to Question 3. [Hashing]
a. [ ] We will use hashing with chaining, with table $T[0..m-1]$ where m is $\Theta(n)$, and a hashing function h. We assume the simple uniform hashing assumption (SUHA), namely, that the distinct integers in the input list L are equally likely to hash in any of the m slots of T. Let $L = x_1, x_2, \cdots, x_n$.

1. We first create a table $T[0..m-1]$ where each entry is a linked list of pairs of the form (y, u), where integer y is in L and u is the number of times y occurs in L.

   To do so we initialize each entry in the table T to the empty list. We then process the list of integers $x_1, x_2, \ldots$ of L as follows: To process x_i, search the list in $T(h(x_i))$ for a pair of the form (x_i, u); if such a pair is found, increase u by 1; if no such pair is found, add $(x_i, 1)$ to the list.

   Since there are at most n distinct integers in L that enter table T, by the SUHA assumption, the expected length of each chain in T is $O(\frac{n}{m}) = O(1)$ (because m is $\Theta(n)$). So the expected time to process each integer x_i in L as explained above is also $O(1)$. Thus, the expected time to process all the n integers in L is $O(n)$.

2. Next, go through the hash table T forming a list L' of all the pairs in T, $L' = (y_1, u_1), (y_2, u_2), \ldots, (y_k, u_k)$; this can easily be done in time $O(n)$.

3. Lastly, sort L' using the u field as the key. Note that $k \leq n$, and the u_i are (not necessarily distinct) integers in the range 1 through n.

   It remains to show how to sort the u_j's of L' in time and storage $O(n)$. To do so, we use a table $C[1..n]$, where $C(v)$ will hold a list of all the y_j's such that $u_j = v$. We construct C by initializing each entry to the empty list, and then by scanning L' as follows: for each j, $1 \leq j \leq k$, insert y_j into the list in $C(u_j)$. We then output the list of y's in $C(n)$, followed by those in $C(n-1), \cdots, C(1)$, in that order.

The worst-case time complexity of the algorithm is $\Theta(n^2)$:

1. To see that it is $\Omega(n^2)$, consider the run when all the n integers of L are distinct and they collide (i.e., they all hash into the same slot of T). In this case, entering the list $x_1, x_2, \cdots, x_n$ into T creates a chain of size n, and so processing these n elements (i.e., searching for each one before entering it into T) takes $\Omega(n^2)$ time.

2. The worst-case time complexity of the algorithm is $O(n^2)$. To see this note that entering each x_i of L in T takes at most $O(n)$ time in the worst-case (because the list at $T(h(x_i))$ has at most n pairs), and so Step 1 of the above algorithm takes at most $O(n^2)$ time in the worst-case. As we already explained, Steps 2 and 3 take at most $O(n)$ time in the worst-case.

b. [ ] We can easily create the list L' defined in part (a) in $O(n \log n)$ time in the worst-case. To do so, first sort the list L in $O(n \log n)$ time (e.g., use heapsort for this). Then scan the sorted list L once to determine for each integer y in L the number of times u that y appears in L: this is easy because all the instances of y in L are now adjacent. After building L', proceed as in part (a). This takes at most $O(n)$ additional time.