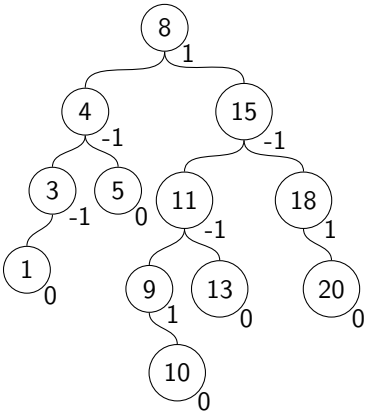
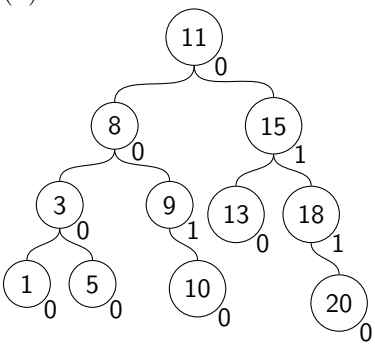
Q. 1 Written by: Anna Lieu, Read by: Xingyu Wang
(a)



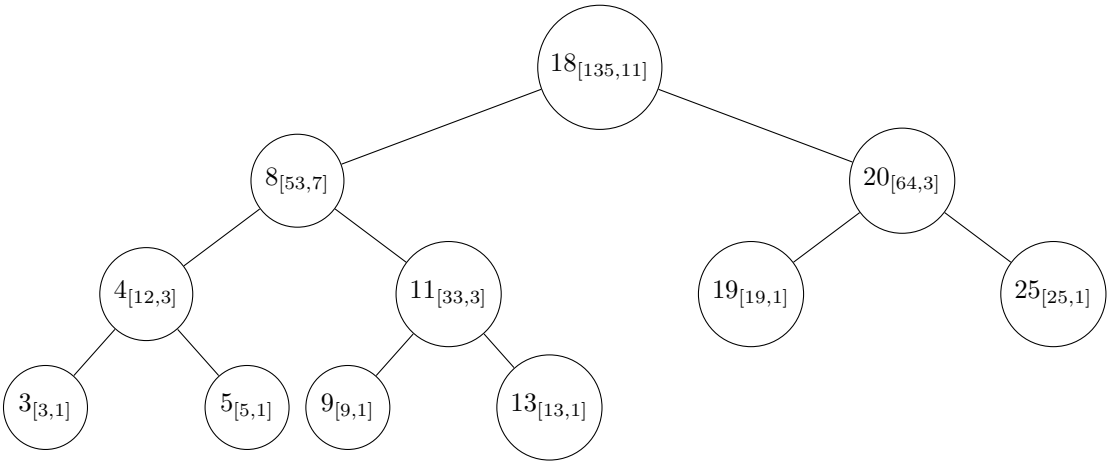
(b)



Q. (2) Written by: Crystal Yip, Read by: Xingyu Wang

Let the AVL tree be our underlying data structure.
Additional information for this D.S.
Let's define:
sum(x) <- the total sum of node values at root x
size(x) <- the total number of nodes including root x
Consider a node x in our augmented AVL. Node x holds the following data:
- the *sum* of itself and it's subtrees [*sum*(x) := *sum*(right(x)) + *sum*(left(x)) + x]
- the *size* of x [*size*(x) := *size*(right(x)) + *size*(left(x)) + 1]

ex. suppose we insert [18, 8, 20, 4, 11, 19, 25, 3, 5, 9, 13] into our augmented AVL



Add(i):
(1) Add element i
- update size factor: costs O(logn) because we update all the ancestors of the inserted node and this path would be at most the height of the tree. While traversing up the tree, increment each node's size by 1
- update sum factor: costs O(logn) because we update all the ancestors of the inserted node and this path would be at most the height of the tree

(2) In the case where tree needs to be rebalanced:
- updating the size and sum information costs constant time because only the pivot nodes of each rotation need to be updated during the rebalance. On an AVL tree, there are at most 2 rotations, so updating size and sum factor after rebalancing takes constant time

Therefore, total cost: O(logn + 2) = O(logn)

Consider the following definitions:
Relative rank of a node x [RRX] := *size*(left(x)) + 1
Relative total of a node x [RTX] := *sum*(left(x)) + x

Average(t):
(1) Search for the element smaller or equal to t by traversing the tree similar to an AVL search algorithm:
- this algorithm is similar to a AVL search algorithm, except we track the value most recently traversed node that is smaller than t.
- we keep traversing until we find a node with value t or until we reach the end of the tree - in this case, our node is the tracked value. Let's call the node, x.
- note: if there is no node with a value less than t, node x is NIL
- this costs O(logn) time - traversing down a path at most the height

(2) Starting from the node x we found in (1), we traverse back up the tree
- if x is NIL, return 0
- initialize variables **total** to node x's RTX and **count** to node x's RRX
- if x is not the root and if it is the right child of its parent, y, we add y's RTX to **total** and y's RRX to the **count**
- set x to the parent (i.e. traverse up one node) and repeat as long as x is not the root
- when you reach the root and **total** is not 0, return **total** / **count**; else return 0
- this costs O(logn) time - traversing up a path at most the height

Therefore, total cost: O(logn + logn) = O(2logn) = O(logn)

*/*find_leq finds the element equal to t or if t is not in the tree, it finds the largest node smaller than t*/*
find_leq(*root*, t):
```
x = NIL; curr = root
if root.value == t: return root // if the root value is t
while curr != NIL: // traverse to find node with closest value ≤ t
    if curr.value == t: return curr
    if curr.value < t and curr.value > x.value: x = curr;   curr = right(curr)
    else: curr = left(curr)
return x
```

Average(*root*, t):
```
curr = find_leq(root, t)
if curr == NIL: return 0
count = size(curr.left) + 1
total = sum(curr.left) + curr.value
while curr != root:
    if curr == right(p[curr]): // if curr is the right child, add RTX and RRX to total and count
        count += size(left(p[curr])) + 1
        total += sum(left(p[curr])) + p[curr].value
```

curr = p[curr] // *move up the tree*
return total/count

Q 3. Written by: Xingyu Wang, Read by: Crystal Yip, Anna Lieu
(a) We define our input to be a list L of n, not necessarily distinct, integers.

1. Let T_1 denote a hash table of size n, and let $h_1(x)$ be a hash function that follows the Simple Uniform Hashing Assumption, and maps a given integer x to an index in the Hash Table T_1.

2. Insert each element in l into the table. We use chaining to resolve possible collision.

   There are n elements, and each insertion takes $\mathcal{O}(1)$, so altogether it takes $\mathcal{O}$(n).

3. Create an empty list $freq$.

   This takes $\mathcal{O}(1)$.

4. Go through L, and operate $search(L, i)$ for all i in L. Instead of yielding on finding the first i in $T[h(i)]$, we traverse through the linked list, recording the number of occurrences of i. Since $h(i)$ always returns the same value, every occurrences of i will be in that linked list. Then this number of occurrences is the frequency of i in L Denote this frequency as f.

   Since we have assumed the SUHA to be true, each search operation would cost $\mathcal{O}(1)$. We put the list [i, f] into freq.

   There are n elements, so in total it costs $\mathcal{O}(n)$ in average.

5. Create another hash table T_2 of size n. Let $h_2(x) = x\%n$ be a hash function. Simple Uniform Hashing Assumption, and maps a given integer x to an index in the Hash Table T_2.

   This takes $\mathcal{O}(1)$.

6. We use this hash table to sort all frequency pairs in $freq$. To do that, for each pair $[i, f]$ in $freq$, we insert i into the $h_2(f)$-th index. We resolve collision (elements with the same frequencies) by a linked list.

   There are n integers insertions, so it takes $\mathcal{O}(n)$.

   Because of the nature of our hash function, the integers will be placed at the index which is equal to their frequencies. This means the integer with the lowest frequency will be placed at the first non-empty slot, and vice versa. In other words, they will be sorted by their frequencies in the hash table.

7. We traverse through each slot in the table from the smallest index. We record each integer in the linked list at that slot to a list named $result$.

   There are at most n distinct integers in the hash table, so this will take $\mathcal{O}(n)$.

8. Reverse $result$, and this will be integers in L sorted by decreasing frequency.

   This takes $\mathcal{O}(n)$.

Altogether, the algorithm takes $\mathcal{O}(n)$.

Worst case complexity:
Upper bound: In the worst case, step 1, 2 and 3 still takes $\mathcal{O}(n+1) \in \mathcal{O}(n)$.
In step 4, one search will take $\mathcal{O}(n)$. n searches will give us $\mathcal{O}(n^2)$.
Step 5, 6, 7, 8 altogether still takes $\mathcal{O}(n)$, because we are merely traversing through linked lists where their total length is n.
So the worst case of the algorithm takes $\mathcal{O}(n^2)$
Lower bound: Let L_0 be a list of integers such that all integers have the same hash value by $h_1(x)$.
Then in T_1, there would only be one slot that is occupied, with a linked list of size n.
Each time we try to find the frequency of one element in L_0, we have to traverse through the entire linked list.

n searches will give us $\mathcal{O}(n^2)$.
All other steps have the same running time as the average case.
Then, in this case, the running time $\in \mathcal{O}(n^2)$.

Altogether, the worst case of the algorithm is $\Theta(n^2)$.

(b) Let l, a list of n integers, be an input for the algorithm below.

```
sort_by_frequency(l):
1:      if l is []:
2:          return []
3:      sorted = heap_sort(l)
4:      i, freq = 0, [[0, sorted[0]]]          \\freq is a list of [frequency, element] pairs.
5:      while i < len(sorted):
6:          if freq[len(freq) - 1][1] == sorted[i]:     \\i is another occurrence of the previous element
7:              freq[len(freq) - 1][0] += 1             \\increment frequency
8:          else:                                       \\A new element that has not been recorded in freq
9:              freq.append([1, sorted[i]])
10:         i += 1
11:     result = decreasing_heap_sort(freq)
12:     return [i[1] for i in result]
```

heap_sort() is function deploying the heap sort algorithm from CLRS section 6.4.
decreasing_heap_sort() uses the same strategy, but instead it uses a min-heap and sort by the first element in the nested lists elements.
Both heap sorting algorithm takes worst case $\mathcal{O}(n \log n)$.
Line 1, 2 and 4 takes $\mathcal{O}(1)$.
Line 3 takes $\mathcal{O}(n \log n)$.
The **while** loop on line 5 will run n times.
Within the **while** loop, everything takes constant time.
So line 5 to 10 $\in \mathcal{O}(n)$.
Line 11 takes $\mathcal{O}(n \log n)$.
Line 12 takes $\mathcal{O}(n)$.
Altogether, the algorithm takes $\mathcal{O}(2n \log n + 2n) \in \mathcal{O}(n \log n)$.