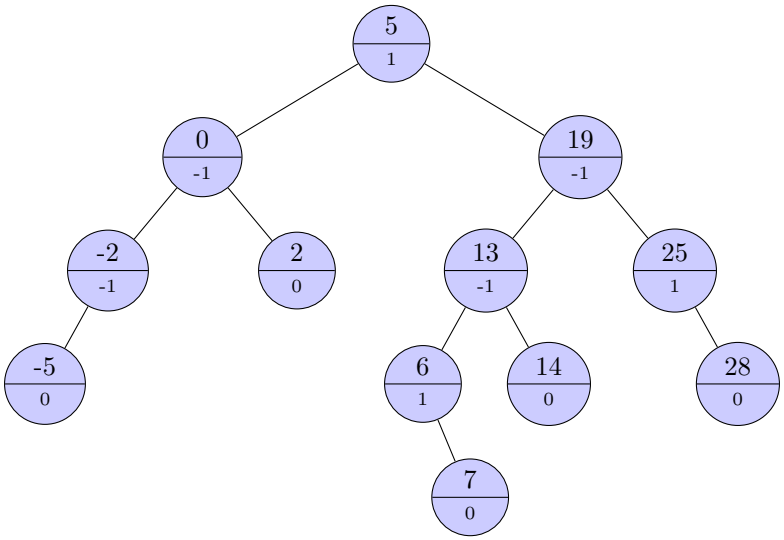
By: Eric Koehli, Jacob Chmura, Conor Vedova

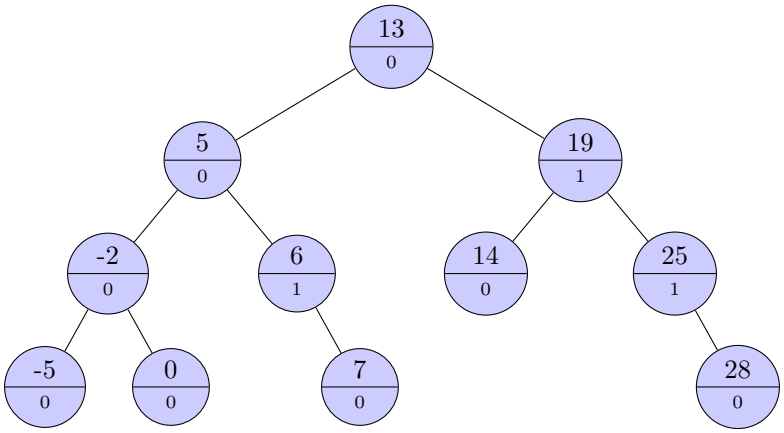
Question 1. [1 MARK]

For parts a and b of this question, note that the key value is in the upper part of the node and the balance factor is in the lower part.

a. Insert keys 0, 25, 19, 5, -2, 28, 13, -5, 2, 6, 14, 7 (in this order) into an initially empty AVL tree.



b. Delete key 2 from the above AVL tree T.



Answered by: Eric
Verified by: Jacob, Conor

Question 2. [1 MARK]

a. Let D be an AVL Tree data structure that stores a set of Books b. Each node b in D is a Book type that has the following attributes:

- id: A unique positive integer that represents a key in the AVL Tree D
- price: A positive real number that represents the price of this Book b
- rating: A positive real number in the range [0, 5] that represents how good the book is

The standard operations we are using to implement `addBook(D, b)` and `searchBook(D, d)` are the AVL algorithms `insert(D, b)` and `search(D, id)`, respectively.

b. Let us also include an AVL tree in our data structure where the key is the price of the book. We will traverse this tree when performing this algorithm. This AVL tree will be augmented so that each node stores the best rating of the books in its left subtree (including the node itself). We also include a pointer to another AVL tree. This separate AVL tree has as keys the ratings of all the books with this price. This tree is augmented with a count of the number of books in D that have this particular price and particular rating. In this way, we can keep track of books with the same price, and also books with the same price and same rating. The algorithm works by traversing down the left children of the price AVL tree, until we reach a price that is less than or equal to the max price input. When we reach this point we return the better rated book from the current nodes left best rated book, and the recursive best rated book on the current nodes right subtree. As a base case we return -1 if we never reach a node that is below the price limit.

```
1       def bestBookRating(D: AVLTree, p: int) -> realNumber:
2           """
3           Return the maximum rating among all books in <D> whose price is
4           at most <p>. If no Book has price at most <p>, then return -1.
5           """
6           # Access price AVL tree
7           PRICE_AVL = D.pricetree
8           curr_node = PRICE_AVL
9
10          # Traverse down the tree until within price range
11          while(curr_node != NULL and curr_node.price > p):
12              curr_node = curr_node.leftchild
13
14          # No books within price range
15          if curr_node == NULL:
16              return -1
17
18          # Return best rated book from left and right subtrees
19          return max(curr_node.best_rated_left.rating,
20          bestBookRating(curr_node.rightchild, p)
```

This algorithm achieves logarithmic runtime, since AVL trees by definition have a height that is proportional to the logarithm of the number of nodes. Since this algorithm traverses to the bottom of the tree in the worst case, we have logarithmic runtime. To add book, we will also have to add the book into the price AVL tree. This can be achieved by inserting according to standard insert algorithm, with the key as the price of the book. In addition though, on every visit up the tree, we must also set the pointer to the best rated book in the left subtree to the new book, if we are coming from the left child, and the new book is strictly better than the previously best rated book.

c. This algorithm works by adding another AVL tree to our data-structure: a Rating AVL tree, (an avl tree with the key as the ratings of books). Each node will also store an AVL tree that contains the book IDs of all books with that particular rating. In this way, we can account for duplicate keys. The algorithm begins by finding the best rated book by running algorithm from part (b), with return value r'. Next traverse down the Rating AVL tree until we find the r' key (r' is necessarily in this tree). We can traverse the AVL tree like a normal search algorithm. Once we reach the node with key r', we return all elements in the AVL tree that this node contains. This algorithm achieves logarithmic runtime since calling the algorithm from part (b) takes logn time, and searching an AVL tree is at most logn. Returning the AVL tree with bookIDs takes constant time. Together, we have O(logn).

d. Define a variable called *priceadder*, and initialize it to 0. Every time you call the function `IncreasePrice(D, p)`, simply increment *priceadder* by p. It is clear that this is constant time. Since the relative price of each book is the same, our price AVL tree is fine as it is. However, when calling SearchBook(D, id), when returning the price, return the price of the book plus the value stored in *priceadder*. This is an extra single operation, so the asymptotic runtime is unaffected and remains logn. All other operations remain the same. Also, when calling BestBookRating(D, p), begin by redefining input p:$p = p - price_adder$. This preserves the true price of a book for the algorithm.

e. Begin by augmenting our original AVL tree, so that each $book_id$ points to the location of that book in both the price AVL tree and rating AVL tree. Next, delete the book with id from original AVL tree as standard delete procedure.

Follow the pointer into the rating AVL tree. If the node in the rating AVL tree does not contain any duplicates, simply delete this node. However, if there are duplicates, traverse the AVL tree that this node contains until we find the correct bookID, then remove this node from the inner AVL tree, while leaving the original node in the rating AVL tree.

Lastly, follow the pointer into the price avl tree and remove the node with the standard delete procedure with the following considerations: every time you delete a node that is a left child which has no duplicates, delete this node and update the parent's best rated left child book to be the max of the deleted node's left child's max rating and the ratings found in the inner AVL tree contained by the parent node. If the node being deleted has duplicates, simply traverse the inner AVL tree to find the node with the correct rating, and decrement it's count attribute. If the count attribute reaches 0, remove the node entirely from the inner AVL tree using the standard procedure. The original node remains, but one must update the max rating of the parent and the current node by taking the max of the left child's max rating and the maximum rating found in the inner AVL tree. This will preserve the augmented rating property.

Deletion from the rating tree and bookID tree are simple and cost O(logn). However, deletion from the price AVL tree is a bit more complicated. However, the maximum number of steps cannot exceed O(logn) time because the maximum height of the two AVL trees (the price tree and the inner AVL tree) is logn which results in 2logn steps. Since all three delete executions cost logn, the total is 3logn which is O(logn).

Answered by: Conor
Verified by: Jacob, Eric

Question 3. [1 MARK]

The algorithm is as follows: Put the elements of set B into a hash table, with a hash function that has *constant runtime complexity and satisfies SUHA principle*. Then for each element in A, say A[i], check if A[i] is in the hash table for B. if it is not, then print it.

Notice that the *expected* time complexity is proportional to n: The for loop on line 31 takes n steps, each iteration taking a constant amount of time by the assumption that the hash function is O(1). By the assumption that our hash function satisfies SUHA, all hashes are uniformly distributed in each bucket, so that the for loop on line 38 is expected to run in constant time. As a result, the entire block of code on line 35 is O(n). Together, linear runtime is achieved as needed.

In the worst case, all n elements of B hash to the same bucket, and we have to loop through all n elements of B, to see if any given element of A is in the set difference. This would mean that the for loop on line 38 takes n steps, and as a result the block of code would be $O(n^2)$. One could make this a tight bound, by choosing elements of B that disrupt the uniform distribution of hashfunction h, that is choosing elements that lead to the same hash. This shows that the worst case running time is $\Theta(n^2)$

```
1       def setDifference(A: Set, B: Set) -> Set:
2           """
3           Return every element of A that is not in B.
4           (i.e. return the set A - B).
5           """
6           #Create an empty hashtable, and define a hashfunction that
7           #satisfies SUHA and is O(1).
8           define htable[];
9           define h = hashfunction(int num) -> int index
10
11          #Put all the elements of B into the hashtable
12          for i in range |B|:
13              htable[h(B[i])] = B[i]
14
15
16          for i in range |A|:
17              seen_element = false
18
19              #If element is not in the bucket, print it
20              for j in range h(A[i]):
21                  if A[i] == h(A[i])[j]:
22                      seen_element = true
23                      break
24              if !seen_element:
25                  print(A[i])
```

Answered by: Jacob
Verified by: Eric, Conor