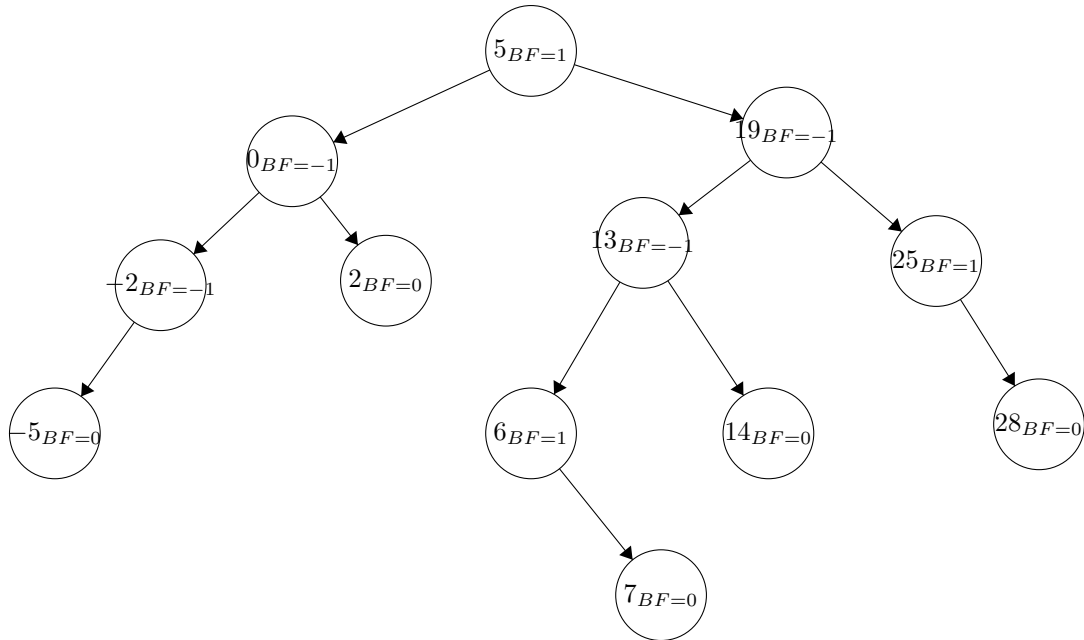


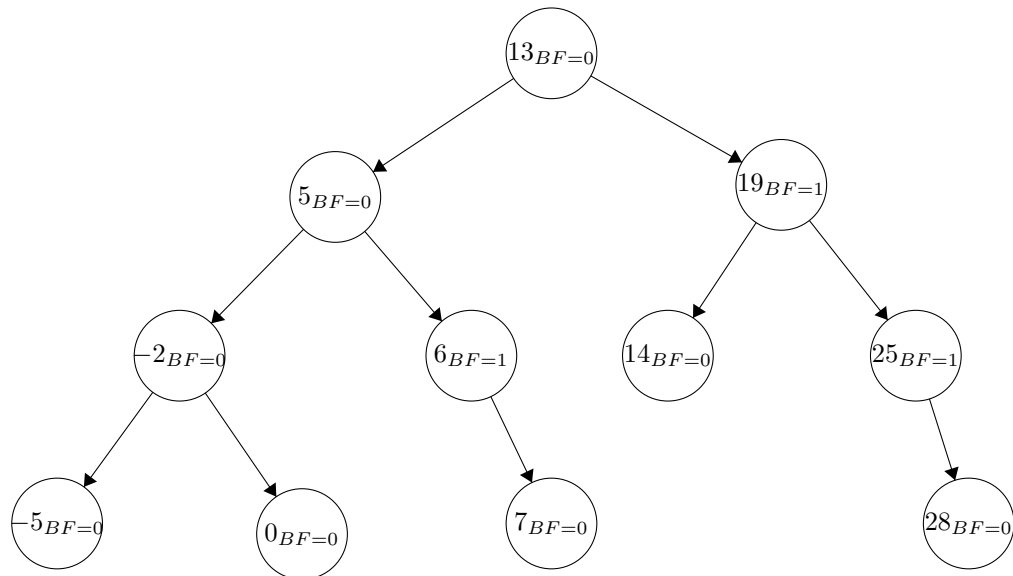
Question 1.

Quan Xu, Zijin Zhang wrote the solution to this question and Zijin Zhang, Quan Xu read this solution to verify its clarity and correctness.

(a)



(b)



Question 2.

Quan Xu wrote the solution to this question and Zijin Zhang read this solution to verify its clarity and correctness.

(a)

Name the AVL structure as a Structure D

- Each node of structure D contains the tuple for each book, (eg. identifier, price, rating).
- The key I use is the identifier of each book when I adding a new book.

Implement $ADDBOOK(D, x)$:

- Assume we need to add n books into D . Let $a \in \mathbb{R}$.
- D is a structure which same as a AVL Tree structure. So when we add x to D , we actually use the identifier of x as a key and use the $INSERT(D, x)$ operation of AVL Tree to insert x to D . Since the worst case running time of $INSERT$ operation is $\mathcal{O}(\log n)$ where n is number of n to add. So the worst case running time of $ADDBOOK(D, x) = a \log n \in \mathcal{O}(\log n)$.

Implement $SEARCHBOOK(D, id)$:

- Assume there are n element in D , let $a, b, c_1, c_2 \in \mathbb{R}$
- D is a struture which is the same as AVL Tree. So structure D is also an Balanced Binary Search Tree. There are maximum two children for each node. So we use the $Search(D, id)$ operation of AVL Tree structure to search the target node by using the key, id . Since $Search$ operation of AVL take $\mathcal{O}(\log n)$, where n is the number of elements. In worst case running time, the $Search$ action will take at most $a \log n$. After we find the node by id , we would take a constant time, c , to get the prive and rating. So the worst case running time for finding a Book successfully takes $a \log n + c_1 \in \mathcal{O}(\log n)$. For finding a Book un-successfully, the searching process also takes at most $\mathcal{O} \log n$, and it would take constant time, c_2 , to return NULL, if there is no such Book in D . So the worst case running time of finding Book un-successfully is $b \log n + c_2 \in \mathcal{O}(\log n)$. So in both case, the worst case running time is $\mathcal{O}(\log n)$.

(b)

```
1 BESTBOOKRATING(D, p)
2   if D is NULL:
3       return -1;
4   elif D.price <= p and D.rchild != NULL and D.rchild.price > p:
5       return max(D.rating, D.lchild.max_r)
6   elif D.price > p:
7       return BESTBOOKRATING(D.lchild, p)
8   elif D.price <= p and D.rchild != NULL and D.rchild.price <= p:
9       if D.lchild is not NULL:
10          left_max_r = max(D.rating, D.lchild.max_r)
11          return max(left_max_r, BESTBOOKRATING(D.rchild, p))
12   else:
13       return D.max_r
```

- I changed the structure of D . In part (a) there is only one AVL Tree in structure D contains two AVL Tree, T_{id} and T_{price} . And each node has one more attribute max_r .

- I changed $ADDBOOK(D, x)$: now when we add Book x into structure D , we add each Book x into T_{id} and T_{price} respectively. So there are two copies of each Book x , one in T_{id} and one in T_{price} and each copy have the same pointer. When adding a Book x into T_{id} , we use insert opeartion to insert Book by using id as a key; and when we add Book x into T_{price} , we use insert operation by using price as a key. Also, we add one more attribute, max_r of each node in T_{price} . The new attribute max_r stores the maximum rating among the tree rooted by the node. If there is a same price, we store it as the left child. When we do rotate action, we can still get the max_r of the nodes which change some pointer recently, since $node.max_r = \max(node.rating, node.rchild.max_r, node.rchild.max_r)$ and it takes constant time to update.

- So the worst case running time of $ADDBOOK$ is $a \log n + c + b \log n \in \mathcal{O} \log n$.

- *SEARCHBOOK* operation is not changing, it still search id in the T_{id} by using key identifier. So the worst case running time will not change.

Description and running time of *BESTBOOKRATING*:

- Assume there are n nodes stored in structure D .
- Line 3 - 4 takes constant time c_1 and *return* - 1 when no book has the price at most p .
- Line 5 - 6 takes constant time c_2 . These two line return the maximum rating of $D.rating$ and the $D.lchild.max_r$ where there is no node on the $D.rchild$ subtree has $price \leq p$. So for line 5 - 6, it at most iteration h times where h is the height of T_{price} . Since it will propagate down one level during each iteration. So it at most iterations $\mathcal{O}(\log n)$.
- For line 7 - 8, since $D.price > p$, so we should go into the $D.lchild$ to find the node whose price is less than or equal to p .
- For line 9 - 12, line 11 takes constant time c_3 and there is a recursive call on line 12 and it propagate one level down on each iteration of this recursive call. So it would iterate at most h times, where h is the height of T_{price} which is also $\log n$. So line 9 - 12 takes at most $c_3 + \log n \in \mathcal{O} \log n$.
- Line 13 - 14 takes constant times.
- Therefore, the running time of *BESTBOOKRATING* is $\mathcal{O} \log n$ in worst case scenario.

(c)

- Base on the augmenting of structure B on question (a) (b), we want to add one more AVL Tree, T_{rating} , in Structure D. Each node in T_{rating} contains a rating, $rate_r$, and a double linked list, dll, which contains all the Books whose rating equal to $rate_r$. And the key for T_{rating} is $rate_r$.

- *ADDBOOK* operation: I changed the *ADDBOOK* operation, since there is a new AVL Tree, T_{rating} , in structure D. When we add a Book to structure D. We take then rating of this book as a key to insert it to T_{rating} by using Insert operation of AVL Tree. And we add the book into the dll of the corresponding node. And if the Book's rating has same value of one of the node's key in T_{rating} , we just find the same rating node and adds the Book into the dll of this node. Also, we add "front" and "next" as attributes of each Book in dll in order to double linked them. So the Insert operation takes $\mathcal{O}(\log n)$ in worst-case runtime. And adding Book in a double linked list takes $\mathcal{O}(1)$. Linking the front and next take constant time, so the worst-case runtime is $\in \mathcal{O}(\log n)$.

- *SEARCHBOOK* operation still unchanged, it still search Book on T_{id} by using id as a key. It still $\in \mathcal{O}(\log n)$.

- *BESTBOOKRATING* operation still unchanged, it still $\in \mathcal{O}(\log n)$.

- *ALLBESTBOOK(D, p)*: We firstly run *BESTBOOKRATING(D, p)* to get the r which is the maximum rating among all Books whose price is at most p in D . If r is '-1', then return NIL. If r is not '-1', then we use Search operation of AVL Tree to search the node whose key has same value of r . After we find the node, we return the dll of the node (the dll is an attribute of node in T_{rating} , and dll pointes to a double linked list). The *BESTBOOKRATING* operation takes $\mathcal{O}(\log n)$, and Search operation takes $\mathcal{O}(\log n)$, and Search operation takes $\mathcal{O}(\log n)$ and return statement takes constant time, C.

- So the worst-case runtime is $\in \mathcal{O}(\log n)$

(d)

- We change the structure of Structure D. We add a variable, *added_price*, to the structure D. And we initialize it as 0. So when we call *INCREASEPRICE(D, p)*, *added_price* += p . So it takes constant time. It is $\in \mathcal{O}(1)$. And if we want to get the price of a Book from structure D, we can just return the price as (node.price + *added_price*), and addition also takes constant time, which is $\in \mathcal{O}(1)$.

- For *ADDBOOK(D, x)*: Since when we want to get the price of a Book, we return its price as (node.price + *added_price*). Thus when we want to add new books, we need to substract the Book's price by *added_price*. Then we can get the true price of a Book, when the price of a Book in structure D returned as (node.price + *added_price*). This also means that the new Book adds into the structure should not be

influenced by the "old" *added_price*. This additional step would take constant time, C. Therefore, when we add a book in T_{id} , in T_{price} and T_{rating} in Structure D, this operation remains $O(\log n)$ time in worst-case.

- SEARCHBOOK operation still unchanged, it still search Book on T_{id} by using id as a key. It still $\in O(\log n)$.

- For BESTBOOKRATING(D, p), when we want to call it, we should firstly subtract *added_price* from p, then we get a new price, p_{new} . Then we pass the p_{new} in function as BESTBOOKRATING(D, p_{new}), since the subtraction takes constant time, so BESTBOOKRATING operation still take $O(\log n)$ in worst-case runtime.

- For ALLBESTBOOK(D, p), when we want to call it, we should firstly subtract *added_price* from p, then we get a new price, p_{new} . Then we pass the p_{new} in function as ALLBESTBOOK(D, p_{new}), since the subtraction takes constant time, so ALLBESTBOOK operation still take $O(\log n)$ in worst-case runtime.

(e)

High-level-idea of DELETEBOOK(D, id):

- Firstly using Delete(D, id) Operation of AVL Tree to delete the nodes by using id as the key in T_{id} of structure D. If rotation happens after deletion, still cost constant time to update. This step takes runtime at most $O(\log n)$

- After we delete the node, d, with id in T_{id} , we use the pointer to locate the node, d, in T_{price} , and using Delete(D, d.price) operation of AVL Tree to delete the node, d, in T_{price} . If rotation happens, it takes constant time to update max_r of ancestors, since $node.max_r = \max(node.rating, node.rchild.max_r, node.rchild.max_r)$. Locating the node takes $O(1)$, delete it takes $O(\log n)$, and updating takes $O(1)$. So the total runtime of worst-case is $\in O(\log n)$

- We also can get the node, d (which is the node we want to delete), by deferencing the pointer, and in T_{rating} we use Search Operation of AVL Tree to search the node which has the same rating as node, d, has. This will take $O(\log n)$. Then we can find the double linked list, dll. Then we use Delete(dll, node_d) operation of double linked list to delete the Book form the double linked list which will takes $O(1)$. If the dll is empty after we deleted the Book from it, we should delete this node, since there is no more Book has the same rating in Structure D. So we use Delete operation of AVL Tree to delete this node, which would take at most $O(\log n)$. Therefore, the Worst-case runtime of this step is $\in O(\log n)$.

- Overall, the worst-case runtime of DELETEBOOK is $\in O(\log n)$

And there is nothing changed for the previously defined operation, so the worst-case time complexity of previous operation reman unchanged.

Question 3.

Zijin Zhang wrote the solution to this question and Quan Xu read this solution to verify its clarity and correctness.

(a)

```

1 SOMEFUNCTION(A, B):
2   Hash_Table: T[0, ..., m-1] # Create a empty Hash Table T and each slot is a linked list
3   Hash_Function: H           # The hash function for the Hash Table T that follows SUHA
4   for element in B:
5     hashing_key = H(element)
6     T[hashing_key].prepend(element) # add the element to the front of the linked list
7   for element in A:
8     hashing_key = H(element)
9     if (element in T[hashing_key]): # check if the element is in the linked list
10      print(element)

```

- First of all, we will create a Hash Table T that has m slots that each slot is a linked list with a hashing function H that follows the Simple Uniform Hashing Assumption($SUHA$).
- For each element in set B , we find the *hashing_key* for each element by using the hashing function H and add the element into the corresponding slot(linked list) in T .
- Then for each element in set A , we also compute the *hashing_key* of each element using the same hashing function H and check if the element is in that slot. If the element is in the slot(linked list) which means the element is not in set B , we will print the element as required. If the element is not in the slot(linked list) which means the same number from set B is added to the hash table T in previous step, we will skip this element and continue checking next element until the elements from set A are all processed.

(b)

- In part (a), we create a Hash Table T in line 2 that has m slots that each slot is a linked list where m is $\Theta(n)$ and a hashing function H that follows the Simple Uniform Hashing Assumption($SUHA$) in line 3. These two steps will take constant time and the expected running time is $\mathcal{O}(1)$.
- For the first loop on line 4, since there are exactly n elements in set B that will enter T , by $SUHA$, we know that the expected length of the each linked list is $\mathcal{O}(\frac{n}{m}) = \mathcal{O}(1)$ (because m is $\Theta(n)$). So the expected time to process each element is $\mathcal{O}(1)$. Thus, the expected running time to process all element in set B is at most $\mathcal{O}(n)$.
- For the second loop on line 7, since there are exactly n element in set A that we need to compute the *hashing_key* by using the hashing function H and then check the value of the corresponding slots, by $SUHA$, the expected running time to process the each element is $\mathcal{O}(1)$ (because m is $\Theta(n)$). Since in pervious step, we proved that the length of each linked list is $\mathcal{O}(1)$, the expected running time for the if statement on line 9 is also $\mathcal{O}(1)$ in each iteration. Also, the expected running time for print operation is constant which is $\mathcal{O}(1)$. Thus, the total expected running for the second loop to process all element in set A is at most $\mathcal{O}(n)$.
- Therefore, the total expected running time for our algorithm under the condition of $SUHA$ is $\mathcal{O}(n)$ since each steps take at most $\mathcal{O}(n)$.

(c)

The worst case running time $T(n)$ of our algorithm under the condition of $SUHA$ is $\Theta(n^2)$. We now show $T(n)$ is both $\mathcal{O}(n^2)$ and $\Omega(n^2)$.

1. $WTS : T(n) \in \mathcal{O}(n^2)$

- Line 1 and line 2 takes constant time $\mathcal{O}(1)$.
- Consider the first loop on lines 4 - 6, the loop will at most iterate n times since the total number of elements in set B is n . Inside the loop, the each operation will take constant time $\mathcal{O}(1)$ as we have discussed in class. So the total worst case running time of the first loop is at most $\mathcal{O}(n)$.
- Consider the second loop on lines 7 - 10, the loop will at most iterate n times since the total number of elements in set A is n . Inside the loop, we assume for the worst case scenario that all the elements that we added in last step is in the same slot(linked list) and each element from set A is not in B but also mapped to that slot. So the if statement on line 9 will at most take $\mathcal{O}(n)$ since there is a linked list with length n that we need to check the every element of it in order to determine if the current element was in the set B . The print operation will take constant time $\mathcal{O}(1)$. So the total worst case running time of the second loop is at most $\mathcal{O}(n^2)$.

Thus, the total running time in worst case scenario $T(n)$ is at most $\mathcal{O}(n^2)$.

2. $WTS : T(n) \in \Omega(n^2)$

Consider the input set $A = 1, 2, \dots, n-1, n$ and set $B = 1, 2, \dots, n-1, n$ that both contain n elements.

- Line 1 and line 2 takes constant time $\Omega(1)$.
- Consider the first loop on lines 4 - 6, the loop will at least iterate n times since the total number of elements in set B is n . Inside the loop, the each operation will take constant time $\Omega(1)$ as we have discussed in class. So the total worst case running time of the first loop is at least $\Omega(n)$.
- Consider the second loop on lines 7 - 10, the loop will at least iterate n times since the total number of

elements in set A is n . Inside the loop, we assume for the worst case scenario that all the elements that we added in last step is in the same slot(linked list) and since $A = B$, all element from A will also mapped to that slot. So the if statement on line 9 will at most take $\sum_{i=1}^n i = \frac{n(n-1)}{2} \in \Omega(n^2)$ since the n elements in A are also in that linked list with length n . The print operation will take constant time $\Omega(1)$. So the total worst case running time of the second loop is at least $\Omega(n^2)$.

Thus, the total running time in worst case scenario $T(n)$ is at least $\Omega(n^2)$.

Therefore, we can conclude that the worst case running time $T(n)$ of our algorithm is $\Theta(n^2)$.