CSC263 Assignment 3

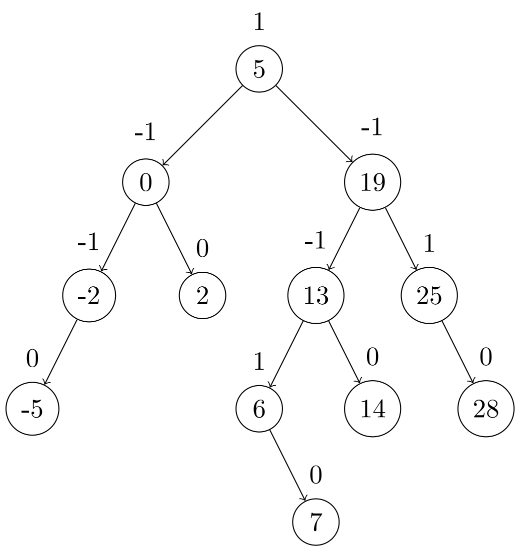
Xinyi Liu, Kewei Qiu, Yongzhen Huang

February 15, 2019

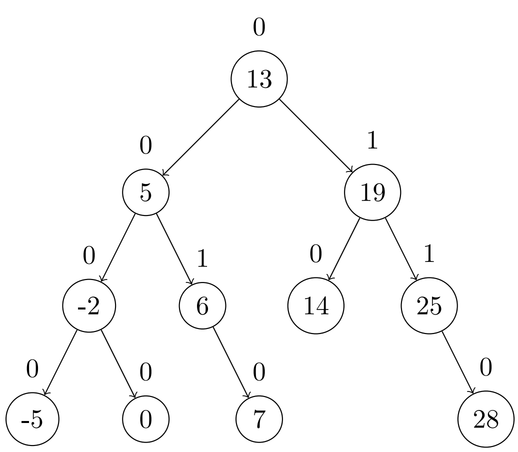
Question 1

Written by: Xinyi Liu, Yongzhen Huang Checked by: Kewei Qiu

a)



b)



Question 2

Written by: Xinyi Liu, Kewei Qiu Checked by: Yongzhen Huang

(a) Data structure used: AVL tree with each book's identifier as the key (call this $T1$)

Each node represents one book (call it b), what we store in each node are: ① b's $identifier$ as key, ② b's *price*, ③ b's *rating*, ④ pointer to left child ($left$), ⑤ pointer to right child ($right$), ⑥ pointer to parent ($parent$), ⑦ balance factor (bf)

We use AVL tree's Insert operation as AddBook, and Search operation as SearchBook. As we discussed in lecture, both of these two functions have running time $\mathcal{O}(\log n)$.

(b) For this question we add a new augmented AVL tree ($T2$). For this augmented AVL tree, each node represents a book and the key will be the *price* of the book. Furthermore, each node x will contain a field called *max_rating* and that is the maximum rating present in the subtree rooted at node x.

BestBookRating(D, p): Suppose the root of D is x. First compare p with $x.price$. If $p \geq x.price$ and x has no children, then rating of x is the maximum. If $p < x.price$ and x has no children, then every book exceeds the price p, return -1. If $p < x.price$, then the max-rating must be in the left subtree, so call BestBookRating(D, p) on the left subtree of x. Otherwise, compare $x.rating$ with left child's *max_rating* and the max-rating of the right subtree by calling BestBookRating(D, p) on the right subtree of x; return the maximum of the three.

Algorithm 1: BestBookRating(D, p)

$x \leftarrow root(D)$
if $p >= x.price$ *AND x has no children* **then**
  return $x.rating$
else if $p < x.price$ *AND x has no children* **then**
  return -1
else if $p < x.price$ **then**
  BestBookRating($x.left, p$)
else if $p >= x.price$ **then**
  $left\_max \leftarrow x.left.max\_rating$
  $right\_max \leftarrow$ BestBookRating($x.right, p$)
  return max($left\_max, right\_max, x.rating$)

The algorithm will take $O(\log n)$ since it either stops or goes down one level each time; and each time the execution takes constant time, since there are $O(\log n)$ levels in an AVL tree, the runtime is $O(\log n)$.

For AddBook(D, x): When adding new book, the book will also be added to the new AVL tree. In particular, it uses $x.price$ as key; it does not matter whether the books with same price is on the left or right the algorithm will work. Furthermore, it updates $x.max\_rating$ as it inserts. If the new book has *rating* higher than the root of the current subtree, it updates the *max_rating* of the root of the current subtree and proceed as it normally does. In the case of rotation, things can be updated easily as well similar to what was described in class. The addition operation is constant time and does not affect overall time complexity of $O(\log n)$.

For SearchBook(D, id): This algorithm remains unchanged as we described in part(a) except that D will refer to the AVL tree in part (a), which has worst-case time complexity $\mathcal{O}(\log n)$

(c) We use another AVL tree ($T3$) with ratings of books as the key. This can be accomplished by storing more data other than those we discussed in part(b).

For each node, what we store in it are: ① a *rating* as key, ② a doubly linked list of *identifier* of all books with this *rating*, ③ pointer to left child (*left*), ④ pointer to right child (*right*), ⑤ pointer to parent (*parent*), ⑥ balance factor (*bf*)

To implement ALLBESTBOOKS(D, p): First we use BESTBOOKRATING(D, p) implemented in part(b) on $T2$, to find the maximum *rating* among all books in D whose *price* is at most p, call it r. Then we use AVL tree's operation SEARCH(D, r) on $T3$ to find the node in $T3$ with key r, and report the linked list stored in it.

For ALLBESTBOOKS(D, x): the running time cost of BESTBOOKRATING(D, p) is $\mathcal{O}(\log n)$, as we shown in part(b), the running time cost of SEARCH(D, r) is also $\mathcal{O}(\log n)$ as we discussed in lecture. Hence the total worst-time complexity of this algorithm is $\mathcal{O}(\log n)$

The implementation of the previously defined operations:

Consider ADDBOOK(D, x). Each time we want to add a book, we first insert it in $T3$. The insert operation is basically like the standard operation INSERT$(T3, x)$ but there is a difference: if there are books with the same *rating*, their *identifier* will be stored and inserted into the head of linked list (which is the value) of the node that has the same key as their *rating*s. Then we insert x into $T1$ and $T2$ using INSERT$(T1, x)$, INSERT$(T2, x)$.

Since inserting an element to the head of the linked list costs $\mathcal{O}(1)$, the time that inserting books into $T3$ is still $\mathcal{O}(\log n)$. Then insert into $T1$, $T2$ costs $\mathcal{O}(\log n)$ as we discussed in part(a), part(b) since they stay unchanged. Adding these three steps together, the worst-case time complexity of ADDBOOK(D, x) is still $\mathcal{O}(\log n)$.

Using SEARCHBOOK(D, id) on $T1$ is enough to finish the task. So this algorithm remains unchanged and the worst-cast time complexity is still $\mathcal{O}(\log n)$.

(d) We use the same data structure as we described in part(c), by storing one more independent value: *base_value*, which is initialized as 0. The components of each node remains unchanged.

To implement INCREASEPRICE(D, p) we just add p to the number stored in *base_value*. Later, when reporting one book's *price*, all we need to do is get its *price* stored in node, then plus the value stored in *base_value*.

Since INCREASEPRICE(D, p) just performs an addition for one time, the cost is an constant. Hence the worst-case time complexity is $\mathcal{O}(\log n)$.

Note that for operations involving price, namely, BESTBOOKRATING, ADDBOOK, subtract the p or *price* by *base_value* to be consistent with original prices. For ALLBESTBOOKS, since it uses BESTBOOKRATING, we do not need to worry about the price there. Since we do not modify anything except subtract which is constant time, The worst-case time complexity of the previously defined operations remains $\mathcal{O}(\log n)$.

(e) We use the same data structure as we described in part(d), by storing two more values to each element in $T1$ (the AVL tree we described in part(a)) *pointer_to_T*2: the pointer to its corresponding element in $T2$ (the AVL tree we described in part(b)) and *pointer_to_T*3 (tree in part(c)). This can be implemented if we insert the book into $T2, T3$ first and we get the pointer to the corresponding element in $T2, T3$, then we insert the book into $T1$ with these pointers.

To implement DELETEBOOK(D, id), we first use AVL tree's DELETE(id) operation to delete the book in

$T1$ which cost $\mathcal{O}(\log n)$. When deleting the book in $T1$, we will get and store the value *rating*, *pointer_to_T2* and *pointer_to_T3*. To delete x from $T2$, first we use *pointer_to_T2* to find the node, then remove the node and update parent's *max_rating* by comparing the *max_rating* for the new node replacing the position of x as well as the *max_rating* of the other child of the parent of x. This ensures that the correct *max_rating* is updated. In the case where x is the root, simply compare the *max_rating* of the children of x and update according for the node replacing x.

After the deletion on $T2$, we use *pointer_to_T3* to find the corresponding element in $T3$. If the element in $T3$ is not the only element in the linked list that contains the element, we will delete it directly from the linked list which cost $\mathcal{O}(1)$. Else, if the element is the only element in the linked list, we will perform the AVL tree's operation DELETE(*rating*) on $T3$ which cost $\mathcal{O}(\log n)$. Then the deletion is finished and the total cost is still $\mathcal{O}(\log n)$.

Here we only modify the operation ADDBOOK(D, x): we insert the book into $T2$ and $T3$ first, get the pointers, then insert x into $T1$ and store the pointer into it, which is nearly the same procedure as we described in part(c). Since the only difference is storing two extra pointers, which costs a constant time, the time of ADDBOOK(D, x) is still $\mathcal{O}(\log n)$. Other operations stay unchanged so their worst-case time complexity also remain unchanged.

Question 3

Written by: Kewei Qiu Checked by: Xinyi Liu, Yongzhen Huang

(a) We use hash table as our data structure, call the table we use as H.

Algorithm implementation: First insert all elements of set B into the hash table using INSERT(H, e_1) for all e_1 in B. Then do SEARCH (H, e_2) for all e_2 in set A. If e_2 is already in the hash table, which means that e_2 is also an element in B, we ignore it. Else, the algorithm will print e_2, as an element of the set $A - B$.

Algorithm 2: DIFFERENCE

create an empty hash table H with size m, where m/n is a constant, for number of elements in the set B as n
for (e_1 in B)
    INSERT(H, e_1)
end

for (e_2 in A)
    if (SEARCH(H, e_2) == NIL)
        then print e_2
end

(b) Assume the hashing function of our hash table satisfies SUHA, i.e. any key k is equally likely to be hashed into any of the m slots of H. Also, assume H's size m equals $\theta(n)$ where θ is a constant number, i.e. $\alpha = n/m$ is a constant number.

The INSERT(H, e_1) operation, as we discussed in lecture, costs $\mathcal{O}(1)$ for each element, hence the total cost of inserting n elements of B into the hash table depends on the size of set B, and is $\mathcal{O}(n)$.

By our assumption, the searching operation SEARCH (H, e_2) costs $\mathcal{O}(1)$ for each e_2 in set A, since there are n elements in A, and the print operation costs a constant time, the total cost of this step is also $\mathcal{O}(n)$.

Hence the expected running time of our algorithm is $\mathcal{O}(n)$.

(c) In the worst case, all elements of the set B are hashed into the same position. This does not change the running time of inserting step, but makes the cost of searching-printing step change to n^2 since each SEARCH (H, e_2) will traverse the list which contains all elements in B, with length n, and do this n times gives us a running time $\mathcal{O}(n^2)$. Hence the worst-case time complexity is $\mathcal{O}(n^2)$.

Choose a particular hashing function $h(k) = 1$ for any key k. By using this hashing function, all elements of the set B are hashed into position 1 in the hash table, costs $\mathcal{O}(n)$ in total. Next, all element in A will also be hashed to position 1, and the algorithm will go through all elements stored in position 1 (there are n of them) to check if e_2 is already in the hash table, which costs $\mathcal{O}(n)$. Doing this step n times costs $\mathcal{O}(n)$, and the worst-case time complexity is $\Omega(n^2)$.

In conclusion, the worst-case running time of our algorithm is $\Theta(n^2)$.