CSC263H1 Assignment 3

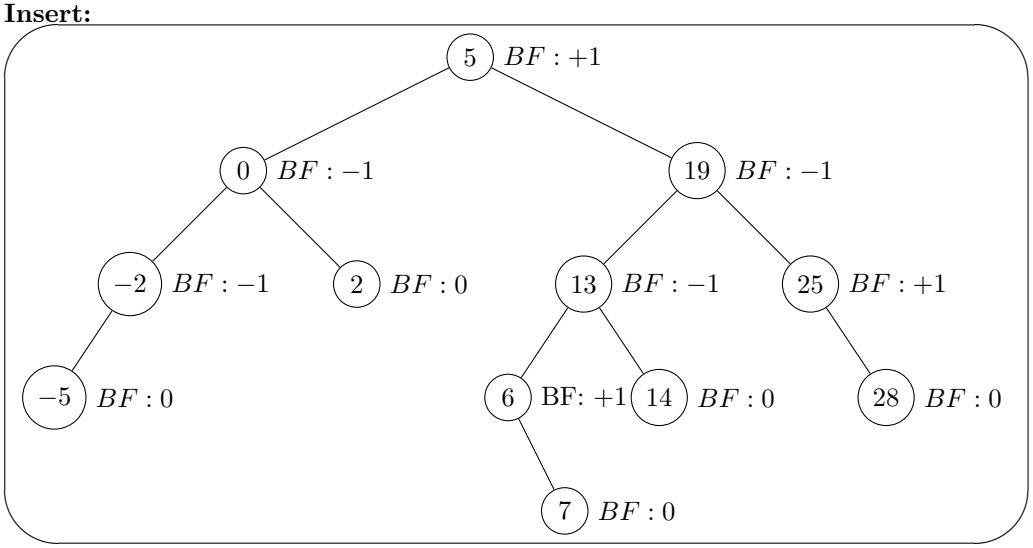
Jiatao Xiang, Xu Wang, Huakun Shen

February 14th, 2019

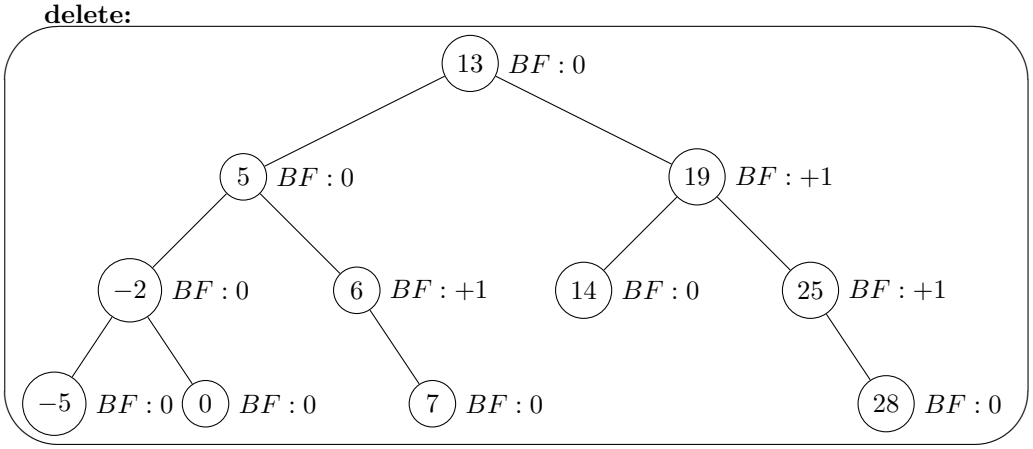
Question 1

Written by Jiatao Xiang; Checked by Huakun Shen, Xu Wang

Insert:



delete:



Question 2

Written by Xu Wang; Checked by Huakun Shen, Jiatao Xiang

Notice:

For all AVL tree that will be defined in this question, we assume each node in the tree all have these basic attributes: balance factor, parent, left and right child. We assume all insertion and deletion operation of AVL tree we defined are those have been discussed in lecture. Which all take $\mathcal{O}(\log n)$.

a. Define a new data structure, D, which is a AVL tree. Let's call this **id_tree**. Each node has attributes in the assumption at the beginning and these attributes: identifier, price and rating. We set its identifier as

key.

Description of the Algorithm

Notice that: D is the root node of the AVL tree.

AddBook(D, x): we perform the insert operation(which also includes rebalancing) of an AVL tree that we discussed in lecture. This takes $\mathcal{O}(\log n)$.

SearchBook(D, id): we perform the search operation of an AVL tree that we discussed in lecture. This takes $\mathcal{O}(\log n)$.

b. We add one more AVL tree to our existed data structure such that each of its node contains price (which is also the **key**), rating, the max_right_rating (this stores the maximum rating among all nodes in the subtree rooted at a node's right child), and the max_left_rating (this stores the maximum rating among all nodes in the subtree rooted at a node's left child) and attributes in the assumption. Let's call it **price_tree**.

Modification to existed data structure:

AddBook(D, x):

We now need to add one more attribute, **price_alias**, to nodes in id_tree. This is the pointer that points to the node in price_tree which has the same identifier.

In this case, AddBook(D, x) defined previously also needs to add x to the price_tree, and a pointer to the x in price_tree is returned and stored in the corresponding node in id_tree ($\mathcal{O}(\log n)$). Then, rebalancing both trees ($\mathcal{O}(1)$), and update the max_left_rating and max_right_rating of each node in price_tree ($\mathcal{O}(\log n)$). Thus, the total time taken is still $\mathcal{O}(\log n)$.

SearchBook(D, id):

The operation SearchBook(D, id) remains same.

Description of the Algorithm:

We start with the root of D, there exists three conditions:

First, when D has only right children: if $D.price \leq p$, we need to check D's rating and its max_right_rating, and return the greater one. If $D.price > p$, -1 is returned, since there is no node satisfying the requirement.

Second, when D has only left children: if $D.price \leq p$, we need to check D's rating and its max_left_rating, and return the greater one. If $D.price > p$, call BestBookRating recursively to the maximum rating of D's left subtree.

Third, when D has both left and right child, if $D.price > p$, call BestBookRating recursively to find the maximum rating of the left subtree. Otherwise, we return the max{D.max_left_rating, D.rating, BestBookRating(D.right, p)}.

Simply speaking, the algorithm will traverse downwards from the root of the tree and it will go in exactly one direction (either to the left side or the right side) or return value directly for each call, so the max path it will go through is less than or equal to the height of the tree. Thus, run time is $\mathcal{O}(logn)$.

Let $T(n)$ be the run time of the algorithm.

$$T(n) = \begin{cases} c, & if\ leaf \\ T(\lfloor \frac{n}{2} \rfloor) + c_1, & if\ left\ is\ NULL \\ T(\lfloor \frac{n}{2} \rfloor) + c_2, & if\ right\ is\ NULL \\ T(\lfloor \frac{n}{2} \rfloor) + c_3, & if\ has\ both\ left\ and\ right \end{cases}$$

```
def BestBookRating(D, p):
  if D is leaf:
    if D.price <= p:
      return D.rating
    return -1
  elif D has only right child:
    if D.price <= p:
```

```
      return max(BestBookRating(D.right, p), D.rating)
    return -1
  elif D has only left child:
    if D.price <= p:
      return max(D.max_left_rating, D.rating)
    return BestBookRating(D.left, p)
  elif D has both left and right child:
    if D.price > p:
      return BestBookRating(D.left, p)
    return max(D.max_left_rating, BestBookRating(D.right, p), D.rating)
```

c. Description:

AllBestBooks(D, p): Based on Part a and b, we create another AVL tree, where rating is the key. Let's call it rating_tree. For each node, there is a new attribute called RatingFamily which is a pointer points to an AVL tree (let's call it inner_tree) that stores every node with this rating. The inner_tree is sorted with their identifier. Whenever we find the rating r from question b, we can search this rating in rating_tree and return RatingFamily, this gives all node with rating r. The largest depth of rating_tree is $\log(n)$, thus this will cost at most $\mathcal{O}(log(n))$.

Modification to existed data structure:

AddBook(D, x):

In addition to what we already defined, we now have some new features to add. First, we need to add a new attribute, called **price_alias**, to the node in id_tree. After adding x to id_tree and price_tree, we find the node with same rating as x in the rating_tree. If there is no such node exist in rating_tree, we add a new node for it. So far, the process still cost $\mathcal{O}(\log(n))$. Next, we add x to the node's RatingFamily. This also cost $\mathcal{O}(\log(n))$. And store the pointer points to this node in the id_tree. Total time taken is $\mathcal{O}(\log(n))$.

SearchBook(D, id):

The operation SearchBook(D, id) remains same.

d. Description:

IncreasePrice(D, p)

To increment the price of every book in **D** by p in $\mathcal{O}(1)$, we cannot traverse through **D** and increment the price of every element by p, because that will definitely take at least $\mathcal{O}(n)$ of time.

We will define a variable called ***price_inflation*** that is initially set to 0.

Every time *IncreasePrice(D, p)* is called, *price_inflation* is incremented by p.

In addition, we will modify the price attribute of each node to be ***original price + price_inflation***, note that here ***price_inflation*** is a pointer, so that every time the variable is modified, we don't have to modify the price of every node.

In addition, the **insert(x)** function needs to be modified a little. When inserting a new book, we modify its original price to be ***original price - price_inflation*** (here, ***price_inflation*** is not a pointer, but just the value of current ***price_inflation***). Then Also add the ***price_inflation*** pointer to the new "**original price**". In brief, when we ask for the price of a book x, the returned value is actually: **x's original price - value of price_inflation at the time when x is inserted + current price_inflation.**. In the way, the price of a new book wouldn't be affected by previous ***price_inflation***.

e. Description:

DeleteBook(D, id): we first delete the node with the input id from the AVL tree we defined in Part a, this takes $\mathcal{O}(\log n)$. Since for each node in first AVL tree, it has a pointer points to the corresponding node in second AVL tree defined in Part b, then we delete this node from the second AVL tree and this also takes $\mathcal{O}(\log n)$. Lastly, since each node in first AVL tree also contains the rating, so we search in the third AVL tree by rating which takes $\mathcal{O}(\log n)$, and delete the node that has the same identifier from the inner AVL tree which takes $\mathcal{O}(\log n)$. Thus, the whole process takes $\mathcal{O}(\log n)$.

Question 3

Written by Huakun Shen; Checked by Xu Wang, Jiatao Xiang

a. Our data structure is based on **hash table**.

Idea: Put every element of set **B** into a hash table. Then hash every element of set **A** to a slot in the hash table, and check whether the element of **A** is in this slot. If it is not in the slot, then it means that the element of set **A** is not in set **B**.

Pseudo Code:

Let $\alpha \in \mathbb{N}$ that is sufficiently large, i.e. each slot contains a linked list with size of at most approximately α elements.

Suppose we have a hash table $\boldsymbol{T}$ with a size of $\frac{n}{\alpha}$ (number of slots).

Suppose we have a hashing function ***h(x)*** that would return the index of one of the slots of **T** given $\boldsymbol{x}$ as a input. Also assume **SUHA**.

```
# Put elements of B into Hash Table T
for element in B:
    linked_list = T[h(element, len(T))]
    linked_list.append(element)

# Check whether a element of A is in B
# if so, print it
for element in A:
    linked_list = T[h(element, len(T))]
    for item in linked_list:
        if element == item:
            break
    print(element)
```

b. Assumptions:

- The linked list in each slot of hash table has approximately a length of α.
- Hash table **T** has a length of $\frac{len(A)}{\alpha} = \frac{n}{\alpha}$
- SUHA for hash function ***h(x)***

Explanation: Part I: put **B** into **T**

1. Hashing function costs constant time
2. There are n elements in **B**, performing hashing function $\boldsymbol{h}$ for each element of **B** costs $\mathcal{O}(n \times 1)$ of time.

Part II: match element of **A** to **T**

1. Each linked list in each slot of **T** has a size of at most α (by **SUHA**), which is constant. In the worst case, we have to traverse through every linked list, which costs $\mathcal{O}(\alpha) = \mathcal{O}(1)$ of time.
2. There are n elements in **A**, performing step 1 for each of them costs $\mathcal{O}(\alpha n) = \mathcal{O}(n)$ of time.

Part I and Part II altogether costs $\mathcal{O}(n)$ of time.

c. In The worst case scenario, **SUHA** may not hold.

Suppose every element of B are hashed into one single slot of **T**. Then, assume every element of set **A** are also hashed into the same slot, and no element of **A** is in **B**. Then the program needs to traverse through the entire linked list (would not exit early because $A - B = \emptyset$), which takes n steps. Since there are n elements in **A**, each search takes n steps. It takes n^2 steps, so $WC_{RT} = \Omega(n^2)$.

Since there are n elements in $\mathbf{A}$ and n elements in $\mathbf{B}$, comparing each one of the element in one set to every element in the other set takes at most n^2 comparisons. So $WC_{RT} = \mathcal{O}(n^2)$.

$$WC_{RT} = \Omega(n^2) \wedge WC_{RT} = \mathcal{O}(n^2) \Rightarrow WC_{RT} = \Theta(n^2)$$

Q.E.D.