CSC263 Assignment 3

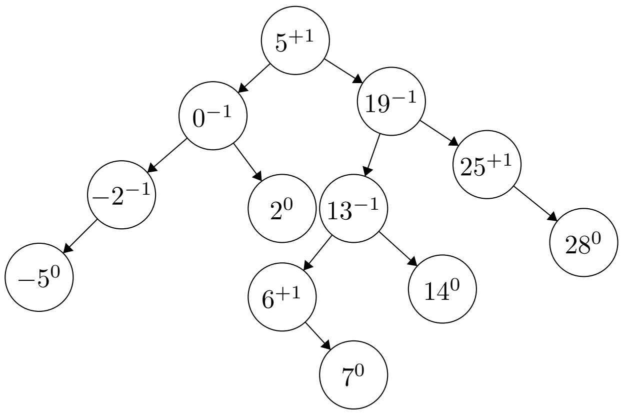
Angela Zhu, Xinyi Ji, Zhuozi Zou

February 14, 2019

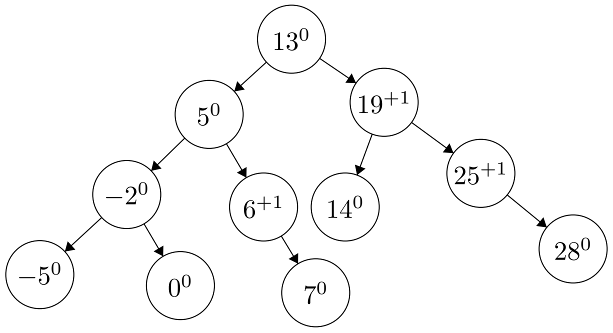
1 Problem 1

(Written by Xinyi Ji, read by Angela Zhu & Zhuozi Zou)

a. question a



b. question b



2 Problem 2

(a, d, e written by Zhuozi Zou; b, c written by Angela Zhu; all parts read by everyone)

a. Data Structure D:

The data structure D uses an AVL tree, call it T_{id}. Each node u of T_{id} contains a 3-tuple $(identifier, price, rating)$ of a book, a pointer $left$ to its left child, and a pointer $right$ to its right child. T_{id} uses the $identifier$ of a $book$ as the key when adding a new $book$.

AddBook - Description:

Since T_{id} is the only data structure in D, AddBook(D, x) is the same as AddBook(T_{id}, x). Use Insert$(T_{id}, x.identifier)$ (of an AVL tree) to add a node containing a 3-tuple $(identifier, price, rating)$ of x, into T_{id}. From the lecture we know that Insert(T_{id}, x) is $\mathcal{O}(\log n)$. Thus AddBook(D, x) is $\mathcal{O}(\log n)$.

SearchBook - Description:

Since T_{id} is the only data structure in D and $identifier$ is the key of T_{id}, SearchBook(D, id) is the same as SearchBook(T_{id}, id). So we use Search(T_{id}, id) (of an AVL tree) to find the node in T_{id} whose $identifier$ is id. Then return the *price* and *rating* in this node. If there is no such a node in T_{id} whose identifier is id, return NIL. From the AVL handout we know that Search(T_{id}, id) is $\mathcal{O}(\log n)$. Thus SearchBook(D, id) is $\mathcal{O}(\log n)$.

b. Data Structure D:

D constructs of 2 AVL trees. The first one same as the one from part a), denoted in this part as T_{id}. Each node in T_{id} contains an additional field in this part: *price_ptr*, which points to the node that represents the same *book* in T_{price}.
The second one is an AVL tree that uses the *price* of all *books* as a key, denoted as T_{price}. Each node in T_{price} contains the *price* (the key), max_left, which is the max rating of all *books* in its left subtree including the node itself, a pointer to its left child, $left$, a pointer to its right child, $right$, and a pointer to its parent, *parent*. There's also another field *books*, a pointer to a max heap. This max heap contains all the *books* with this *price*, and is ordered using *rating* as the key. Each node in the max heap contains all information of the *book*: $identifier$, *price*, and *rating* (the key).

BestBookRating - Pseudocode:

```
BestBookRating($D, p$)
    $r$ = -1
    $temp = T_{price}.root$
    while ($temp.left$ != NIL or $temp.right$ != NIL)
        if ($temp.price <= p$)
            if ($r < temp.max\_left$)
                $r = temp.max\_left$
            $temp = temp.right$
        if ($temp.price > p$)
            $temp = temp.left$
    return $r$
```

BestBookRating - Description:

This algorithm first sets r to -1, and stores the root of T_{price} into a temporary variable, call it *temp*. The algorithm compares the *price* of root to p. If it is greater than p, move on to the left child and store it in *temp*. If it is less than p, this means the max_left value is a possible greatest rating of all *books* with *price* at most p, therefore store the larger of these two into r. Move on to the right child and store it in *temp*. The comparison stops until there are no more valid children to compare to. This indicates that the last node that has *price* less than p must have the largest *price* of all *books* that is less than p. And the current r value must be the maximum *rating* of all *books* with *price* less than p. Or, if there was no such *price*, r remains -1. And r is returned.

Time Complexity:

The worst-case time complexity of this algorithm is $\mathcal{O}(\log n)$. The while loop in the algorithm starts from the root and traverses down the tree, and terminates when reaching a leaf. Since the maximum distance between a leaf and the root is $\log n$, the while loop takes at most $\log n$ steps. Since the other steps in the algorithm (if statements and assignments) take constant time, it is clear that there exists a constant $c > 0$ such that for every input key p, executing the procedure BestBookRating(D, p) takes at most $c \cdot \log n$ time, therefore satisfying $\mathcal{O}(\log n)$.

Changes to Previous Implementation:

AddBook(D, x): In addition to inserting the book into T_{id}, the book is also inserted into T_{price}: First perform the AVL operation Search$(T_{price}, x.price)$, and store the result in a variable S.
If the S is not NIL, perform the max heap operation Insert$(S.books, x)$ and add x into the max heap. If x becomes the new root of $S.books$, compare $S.max\_left$ with $x.price$, and change max_left if $x.price$ is larger.

If S is NIL, create a new $node$ with key $x.price$. Create a node for x as the only element in the max heap, and assign $S.books$ to x, $S.max\_left$ to $x.rating$. Perform the AVL operation Insert$(T_{price}, node)$ to insert the $node$ into the tree. The $price\_ptr$ in the node in T_{id} should point to the node for x, which is in the max heap of the node in T_{price} with $price = x.price$, which is $S.books[i]$, where i is the index of the node x.

Traverse up the path from $node$ up to the root, and update all affected max_left values. Which, if $node.max\_left$ is larger than any max_left values of its parents, update the value. This operation takes $\log n$ time.

Re-balancing rotations affect the max_left values. When a left rotation is performed, simply compare the original right child of the rotation node with the rotation node, and change the max_left value of the child node if the max_left value of the rotation node is larger. When a right rotation is performed, compare the rotation node with its new left subtree, and change the max_left value of the child node to the larger value of the max_left of its new left subtree and the $rating$ of the root of its $books$ max heap. This operation takes constant time.

*Since the AVL Search, max heap Insert, AVL Insert, and max_left update operations used all have worst time complexity $\mathcal{O}(\log n)$, the time complexity of AddBook(D, x) remains unchanged.

c. Data Structure D:

D constructs of 3 AVL trees. The first one, T_{id}, is the same as the one from part b). Each node in T_{id} contains an additional field in this part: $rating\_ptr$, which points to the node that represents the same $book$ in T_{rating}.
The second one, T_{price}, is the same as the one from part b).

The third one is an AVL tree that uses the $rating$ of all $books$ as a key, denoted as T_{rating}. Each node in T_{rating} contains the $rating$ (the key), a pointer to its left child, $left$, a pointer to its right child, $right$, and a pointer to its parent, $parent$. There's also another field $books$, a pointer to a two directional linked list. This linked list contains all the $books$ with this $rating$. Each node in the linked list contains the $identifier$ and $rating$ of the book, and $next$, the pointer to the next element in the list, $previous$, the pointer to the previous element in the list.

Algorithm Description & Time Complexity:

This algorithm first uses BestBookRating(D, p) to find r. Next, perform the AVL operation Search(T_{rating}, r) to find the node r in T_{rating}, call it $target$. Return $target.books$, the first element of the linked list of the node with $rating$ r. If o $book$ has $price$ at most p, BestBookRating returns -1 which cannot be found in T_{rating}, thus Search should return NIL.

The worst-case time complexity of this algorithm is $\mathcal{O}(\log n)$. The algorithm only performs two operations: BestBookRating and the AVl Search, both taking at most $\log n$ steps. Since returning the first node of a linked list takes constant time, and the other steps in the algorithm also take constant time, it is clear that there exists a constant $c > 0$ such that for every input key p, executing the procedure AllBestBooks(D, p) takes at most $c \cdot 2 \log n$ time, therefore satisfying $\mathcal{O}(\log n)$.

Changes to Previous Implementation:

AddBook(D, x): In addition to inserting the book into T_{id} and T_{price}, the book is also inserted into T_{rating}: First perform the AVL operation Search$(T_{rating}, x.price)$, and store the result in a variable S.

Create a linked list node for x. The *rating_ptr* in the node in T_{id} should point to this node. If the S is not NIL, make this node the new head of the linked list *S.books*.
If S is NIL, create a new *node* with key *x.rating*. *node.books* should point to the newly created linked list node for x. Perform the AVL operation Insert$(T_{rating}, node)$ to insert the *node* into the tree.

*Since the AVL Search, and Insert operations used all have worst time complexity $\mathcal{O}(\log n)$, and adding a new node as the head of a linked list takes constant time, the time complexity of AddBook(D, x) remains unchanged.

Other operations are not affected.

d. Let *Base* be a single node with one field $Base.price = T_{price}.root.price$ (the price value of the current root of T_{price}) (if T_{price} is empty then let $Base.price = 0$).

Data Structure D Modifications:

1. Add *Base* to D as the forth data structure.
2. T_{id}: Delete the *price* field in each node.
3. T_{price}: For each node w: change *w.price* to $Base.price - w.price$.

IncreasePrice - Description:

Since only T_{price} contains the *price* field, IncreasePrice(D, p) is the same as IncreasePrice(T_{price}, p).
We just simply increase *Base.price* by p dollars.
Before IncreasePrice(D, p): the *price* of each *book* b is $Base.price - b.price$.
After IncreasePrice(D, p): the *price* of each *book* b is [*Base.price* before IncreasePrice(D, p)] $+ p - b.price$.
Therefore, IncreasePrice(D, p) successfully increases the *price* of every *book* in D by p dollars.
Since finding the *Base* node and increasing *Base.price* by p dollars takes constant time, it is now clear that there is a constant $c > 0$ such that for all $n > 0$: for every D with n books, IncreasePrice(D, p) takes at most c time. Therefore, the worst-case running-time of IncreasePrice(D, p) is $\mathcal{O}(1)$.

Changes to Previous Implementation:

1. AddBook(D, x):
   If T_{price} is empty, let $Base.price = x.price$. Let m be $Base.price - x.price$. Do the original AddBook(D, x) but:
   a) when inserting x to T_{id}: only store a 2-tuple $(identifier, rating)$ in T_{id}.
   b) when inserting x to T_{price}: Use m as the key and the *price* value.
   The above changes do not change the worst-case time complexity so AddBook(D, x) is still $\mathcal{O}(\log n)$.
2. SearchBook(D, id):
   Do the original SearchBook(D, id) to find the corresponding node k in T_{id}. If there is no such a node in T_{id} whose *identifier* is *id*, return NIL. Otherwise, use the *k.price_ptr* to find the

corresponding node h in T_{price}. Return ($Base.price - h.price$) and $h.rating$.
The original SearchBook(D, id) is $\mathcal{O}(\log n)$, using the $price\_ptr$ to find the corresponding node takes constant time. Thus SearchBook(D, id) is still $\mathcal{O}(\log n)$.

3. BestBookRating(D, p):
The $price$ of each book b now becomes $Base.price - b.price$.
The above change does not change the worst-case time complexity so BestBookRating(D, p) is still $\mathcal{O}(\log n)$.

e. Description & Time Complexity:

Call Delete(T_{id}, id) (of an AVL tree). If no such node exists, we're done. Otherwise, before deleting the corresponding node k from T_{id}:

1. Use $k.price\_ptr$ to find the corresponding node h in T_{price}. Let q be the node returned by Search($T_{price}, h.price$). Delete h from the max heap h belongs to is similar to what we did in assignment 2 1b): First increase $h.rating$ to $+\infty$, this will cause h to bubble up to the root of its max heap, then remove h using the ExtractMax operation on the max heap h belongs to.

   If h is the root of the max heap, after removing h: change $q.max\_left$ to $max\{q.left.max\_left$, $q.books.root.rating\}$. For y from q to the root: update $y.max\_left$ to $max\{y.books.root.rating$, $y.left.max\_left\}$, and stop when we find the changed $y.max\_left \leq y.books.root$.

   If h is the only node in q, we also need to delete q from T_{price}. Change $q.parent.max\_left$ to $max\{q.left.max\_left, q.parent.books.root.rating\}$. For g from $q.parent$ to the root: update $g.max\_left$ the same way as what we did on y in the previous part. Delete q from T_{price} using Delete($T_{price}, q.price$). Then if rotation is needed to re-balance T_{price}, it is quite similar to what we did in the re-balance part in (b).

2. Use $k.rating\_ptr$ to find the corresponding node j in T_{rating}. Since j is a node of a linked list, call Delete(j) (of a doubly linked list) on the doubly linked list j belongs to (we learned how to delete a node from a linked list in CSC148 and now we also have the reference to the node in front of j). If j is not the only node in the doubly linked list, we are done for this step. Otherwise, we also need to delete the node that j belongs to, whose key is $rating$. Call this node s. Use Delete($T_{rating}, j.rating$) (of an AVL tree) to delete s from T_{rating}.

Now we can delete k from T_{id} using Delete(T_{id}, id) (of an AVL tree).

In step 1: Find h in T_{price} takes constant time. Search($T_{price}, h.price$) is $\mathcal{O}(\log n)$. Delete h is $\mathcal{O}(\log n)$ (from a2 1(b) we know that delete on binomial max heap is $\mathcal{O}(\log n)$. Delete on max heap is also $\mathcal{O}(\log n)$ since they have the same max height log [U+2061]n). Update max_left is $\mathcal{O}(\log n)$ (since traverse up to the root is takes at most $\log n$ time and each update takes constant time. Delete($T_{price}, q.price$) is $\mathcal{O}(\log n)$. Rotation takes constant time.

In step 2: Find j in T_{rating} takes constant time. Delete(j) takes constant time (since j is in a doubly linked list). Delete($T_{rating}, j.rating$) is $\mathcal{O}(\log n)$. Delete k from T_{id} is $\mathcal{O}(\log n)$.

Thus it is now clear that there is a constant $c > 0$ such that for all $n > 0$: for every D with n books, DeleteBook(D, id) takes at most $c \cdot \log n$ time. Therefore, the worst-case running-time of DeleteBook(D, id) is $\mathcal{O}(\log n)$.

3 Problem 3

(Written by Xinyi Ji, read by Angela Zhu & Zhuozi Zou)

a. First, use hash function to insert every node of B into the hash table which size is of $\mathcal{O}(n)$. Second, for every node of A, try to search it in the hash table we've implemented before (we learned how to search the element of the given key in class). If the search operation returns an element, it means that B has the same element of A and we don't include this element in the result. If the search operation returns null, it means that B doesn't have this element and we can include this element in the result. Finally, we return the result, which is exactly $A - B$.

<u>pseudocode:</u>

```
Except($A, B$)
1   for $element1$ in $B$
2       Insert($element1$)
3   for $element2$ in $A$
4       if(Search($element2$)) == NULL)
5           $result = result + element2$
6   return $result$
```

b. For steps 1 and 2, the for loop has $length(B) = n$ iterations, and each iteration takes constant time since we learned in class that the Insert operation is of $\mathcal{O}(1)$. Hence steps 1 and 2 take $\mathcal{O}(1) \cdot n$ time, which is of $\mathcal{O}(n)$.
For steps 3,4,5, the for loop has $length(A) = n$ iterations and each iteration takes constant time since we learned in class that the Search operation is of $\mathcal{O}(1)$ under SUHA assumption(any key k is equally likely to hash into any of the m slots of T (independent of other keys)). Hence steps 3,4,5 take $\mathcal{O}(1) \cdot n$ time, which is of $\mathcal{O}(n)$.
Hence the whole algorithm's expected running time is of $\mathcal{O}(n) + \mathcal{O}(n)$, which is of $\mathcal{O}(n)$.

c. The worst-case running time of my algorithm is of $\Theta(n^2)$, the explanation is as the following.

First, the worst-case running time of my algorithm is of $\mathcal{O}(n^2)$:
Let A, B be two arbitrary lists of length n, for steps 1 and 2, it has $length(B) = n$ iterations, and each iteration takes constant time since we learned in class that the Insert operation is of $\mathcal{O}(1)$. Hence steps 1 and 2 take $\mathcal{O}(1) \cdot n$ time, which is of $\mathcal{O}(n)$. For steps 3,4,5, it has $length(A) = n$ iterations and each iteration takes $\mathcal{O}(n)$ time since we learned in class that the Search operation is of $\mathcal{O}(n)$. Hence steps 3,4,5 take $\mathcal{O}(n) \cdot n$ time, which is of $\mathcal{O}(n^2)$ time. Hence the whole algorithm's running time is of $\mathcal{O}(n) + \mathcal{O}(n^2)$ which is of $\mathcal{O}(n^2)$. Hence the worst-case running time of my algorithm is of $\mathcal{O}(n^2)$.

Second, the worst-case running time of my algorithm is of $\Omega(n^2)$:
Let A, B be two lists of length n whose elements all have the same hash function result. For steps 1 and 2, it has $length(B) = n$ iterations, and each iteration takes constant time (call it c), since we learned in class that insert operation is of $\mathcal{O}(1)$. Hence steps 1 and 2 take $c \cdot n$ time. For steps 3,4,5, it has $length(A) = n$ iterations and each iteration takes $n \cdot constant$ time (call it d). We learned in class that the time of the Search operation is proportional to length of linked list search which is $n \times constant$ time. And all elements of B has the same hash function result so they will be in the same linked list therefore having length n, and it takes constant time to find whether the element we are searching in this linked list is equal to the element of the iteration of A. Hence steps 3,4,5 take $n \cdot d \cdot n$ time, which is of $d \cdot n^2$ time. Hence the whole algorithm's running time is of $c \cdot n + d \cdot n^2$ which is of $\Omega(n^2)$. Hence the worst-case running time of my algorithm is of $\Omega(n^2)$.

Hence The worst-case running time of my algorithm is of $\Theta(n^2)$.