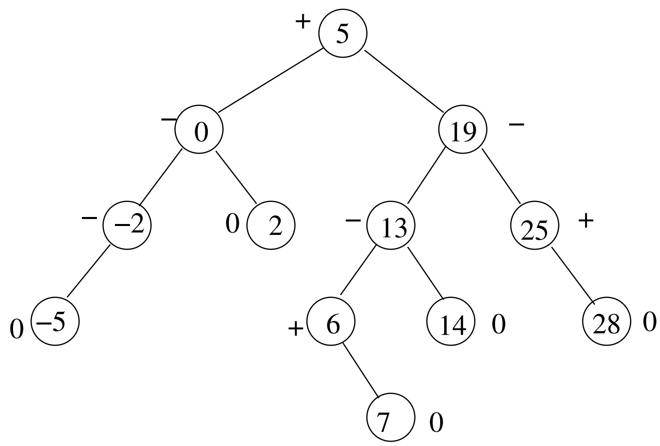
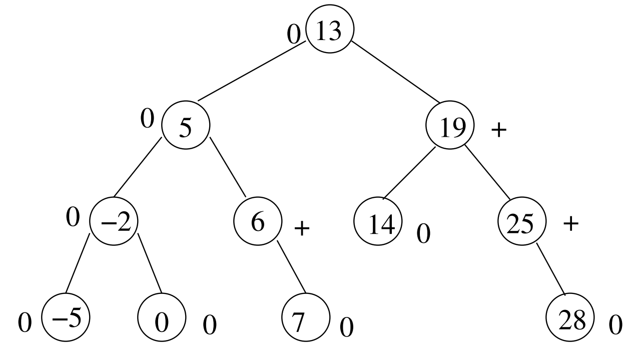
Solutions for Homework Assignment #3

Answer to Question 1. [Simple AVL]



After the twelve insertions



After the deletion of 2

Answer to Question 2.

a. Our data structure D is an AVL tree that we denote T_{ID}. Each node u of the AVL tree T_{ID} contains the following information of a ***book***. It contains fields *identifier*(u), giving unqiue *identifier* of the ***book***; *price*(u), giving the *price* of the ***book***; *rating*(u), giving the *rating* of the ***book***. Field *identifier* is used as the key of the node. In addition, each node u contains the usual fields of an AVL tree node: pointers to the left and right children as well as the parent, and the balance factor. The AddBook and SearchBook operations are implemented using the standard AVL insert and search operations. Consequently, the worst-case time complexity of each operation is $O(\log n)$.

b. To support the BestBookRating operation, we use an *augmented* AVL tree, denoted T_{PRICE}, in addition to the AVL tree T_{ID}. Each node u of the AVL tree T_{PRICE} contains the following information of a ***book***: *identifier*(u), giving the unique *identifier* of the ***book***; *price*(u), giving the *price* of the ***book***; *rating*(u), giving the *rating* of the ***book***. In addition, node u is *augmented* with the field *maxRating*(u) that contains the maximum *rating* of all the ***books*** in the subtree rooted at u (including u).

Note that the following identity holds at every node u of T_{PRICE}:

$$maxRating(u) = \max\big(rating(u), maxRating(lchild(u)), maxRating(rchild(u))\big) \qquad (*)$$

(where $maxRating(\text{NIL}) = -1$).

Thus, the value of the augmented field *maxRating*(u) at a node u can be computed "locally" from the *maxRating* of the children of u, and the *rating*(u). This is why updating the augmented fields (e.g., during an insertion or a rotation) can be done efficiently.

Field *price* is used as the key of each node u in T_{ID}. In addition to the above fields, each node u contains the usual fields (pointers and balance factor) of an AVL tree node.

Note that each ***book*** contributes to a separate node in T_{PRICE}, and since *prices* are not unique, T_{PRICE} can have multiple nodes with the same key (i.e., same price). We handle inserts the same way as we handle inserts in AVL trees where duplicate keys are allowed.

AddBook(D, x) is modified as follows: in addition to inserting the ***book*** $= (identifier, price, rating)$ pointed to by x into T_{ID}, we also insert the ***book*** into T_{PRICE}, using *price* as key to traverse the tree.

Whenever necessary, we update the *maxRating* field of a node u using the identity (*) described above. SEARCHBOOK operation remains the same.

Each insertion into T_{PRICE} involves a standard insert into an AVL tree and updates to the *maxRating* fields of the new node's ancestors, and the *maxRating* fields of the nodes involved in a rotation (a small, constant number of nodes for each rotation). Since there are $O(\log n)$ nodes for which the *maxRating* field must be updated, and each *maxRating* update requires $O(1)$ time, the additional time taken by ADDBOOK(D, x) is also $O(\log n)$. Thus, the worst-case time complexity of ADDBOOK(D, x) remains $O(\log n)$.

Let v be any node of T_{PRICE}. The procedure described below returns the maximum rating of all the books in the subtree of T_{PRICE} rooted at v.

MAXRATING(v, p)
    if $v =$ NIL **then**
        return -1
    else if $p < \mathit{price}(v)$ **then**
        return MAXRATING$(\mathit{lchild}(v), p)$
    else
        return $\max\big(\mathit{maxRating}(\mathit{lchild}(v)), \mathit{rating}(v), \text{MAXRATING}(\mathit{rchild}(v), p)\big)$
    end if

Then BESTBOOKRATING(D, p) returns MAXRATING$(\mathit{root}(T_{PRICE}), p)$, where $\mathit{root}(T_{PRICE})$ is the root of T_{PRICE}.

The worst-case time complexity of MAXRATING(v, p) is proportional to the height of the AVL subtree of T_{PRICE} rooted at v. This is because (i) each call at a node u in this subtree results in a *single* recursive call, at the left or the right subtree of u; and (ii) each call involves a constant amount of work other than the recursive call that it makes. If T_{PRICE} contains n ***books***, the height of any subtree of T_{PRICE} is $O(\log n)$, so the time complexity of MAXRATING(v, p) is also $O(\log n)$ in the worst-case. Consequently, the time complexity of BESTBOOKRATING(D, p) is also $O(\log n)$ in the worst-case.

c. To support the ALLBESTBOOKS operation, we use a third AVL tree, denoted T_{RATING}, in addition to the AVL trees T_{ID}, T_{PRICE}. Each node u of the T_{RATING} contains the following information. It contains fields: *rating*(u), giving a real number in range $[0, 5]$; *bookList*(u), giving a pointer to the head of a doubly linked list which contains all the ***books*** which have r as their rating. Field *rating* is used as the key of the node. Note that a node u with *rating* r exists in T_{RATING} if and only if r is the *rating* of some book in T_{ID}.

ADDBOOK(D, x) is modified so that, in addition to inserting the ***book*** $= (\mathit{identifier}, \mathit{price}, \mathit{rating})$ pointed to by x into T_{ID} and T_{PRICE}, we also insert the ***book*** into T_{RATING}. We do this by using *rating* of the ***book*** as the key to traverse T_{RATING}, and adding ***book*** to the front of the *bookList*(u) of the tree node u which has *rating* as its key (if there is no node in the tree which has *rating* as its key, we create a new node with u with *rating* as its key and *bookList*(u) containing only ***book***, and insert the node into T_{RATING}). SEARCHBOOK and BESTBOOKRATING operations remain the same.

Each insertion into T_{RATING} involves a standard insert into an AVL tree and an insert to the front of a linked list (which takes $O(1)$ time). Since the height of T_{RATING} is $O(\log n)$, the worst-case time complexity of ADDBOOK remains $O(\log n)$.

To implement ALLBESTBOOKS(D, p), we first call MAXRATING$(\mathit{root}(T_{PRICE}), p)$ from Part **b**. Let r be the result returned. If r is -1, we return NIL, else we do a standard search on the AVL tree T_{RATING} for the key r, and return *bookList*(u) of the node u whose key is r.

The operation ALLBESTBOOKS involves a single call to MAXRATING$(\mathit{root}(T_{PRICE}), p)$, which takes $O(\log n)$ in the worst-case, and a standard search on the T_{RATING} AVL tree, which also takes $O(\log n)$ in the worst-case. Hence, the time-complexity of ALLBESTBOOKS is $O(\log n)$ in the worst-case.

d. To support the INCREASEPRICE operation, we add a global `OFFSET` variable, intially set to 0, to keep track of the total amount of price increases so far. When INCREASEPRICE(D, p) is called, we simply increment `OFFSET` by p dollars (this clearly takes $O(1)$ time); we do not modify any of our trees. Hence the variable `OFFSET`, at any point in time, stores the cumulative price increase until that point. The basic idea is to maintain all the books in our AVL trees as if no price increase ever occurred, as we now explain.

For the SEARCHBOOK(D, id) operation, we search the tree T_{ID} for the ***book*** with *identifier* id. Say the search returns the pair $(price, rating)$. We instead return the pair $(price + \texttt{OFFSET}, rating)$ as the result of the SEARCHBOOK operation.

For the ADDBOOK operation, where we want to insert a ***book*** with *price* p into D, we instead insert the ***book*** with its price set to $p - \texttt{OFFSET}$ into D.

For the BESTBOOKRATING(D, p), we return MAXRATING$(root(T_{PRICE}), p - \texttt{OFFSET})$. A similar modification is made to the ALLBESTBOOKS operation.

Clearly, the time complexity of each of the above operations does not change.

e. Assume that the ***book*** with *identifier* id is present in D. To delete the ***book*** with *identifier* id from D:

1. We must delete the node with key id from T_{ID}.
2. Let p be the *price* of the ***book*** with *identifier* id. Then, we must delete from T_{PRICE} the node u whose key is p and the attribute $identifier(u)$ is id (recall that there is exactly one such node).
3. Let r be the *rating* of the ***book*** with *identifier* id and let u be the node in T_{RATING} with key r. Then, we must delete the linked list node ***book*** from $bookList(u)$. Additionally, if deleting the list node ***book*** makes $bookList(u)$ empty, then we have to delete the node u from T_{RATING}.

While it is straightforward to do Step 1 above efficiently, our implementation of D doesn't allow us to do Steps 2 and 3 efficiently, since *prices* and *ratings* are not unique. To solve this problem, we add three attributes to every node u of T_{ID} as follows. Suppose u stores the book $B = (identifier, price, rating)$, then u has the following additional attributes: (1) $priceTreeNode(u)$, giving a pointer to the node v in T_{PRICE} that stores book B, (2) $ratingTreeNode(u)$, giving a pointer to the node w in T_{RATING} whose linked list $bookList(w)$ contains book B, and (3) $ratingListNode(u)$ giving a pointer to the linked list node of $bookList(w)$ that contains book B.

When we insert a new ***book*** with *identifier* id into D, we update the attributes $priceTreeNode(u)$, $ratingTreeNode(u)$, and $ratingListNode(u)$ of the new node u (with key id) that we inserted into T_{ID}, so that they point to the corresponding nodes we inserted in T_{PRICE} and T_{RATING}. Clearly, the worst-case time complexity of ADDBOOK does not change.

To delete a ***book*** with *identifier* id from D, we first search for the node u with identifier id in T_{ID}. We then delete: (1) the node pointed to by $priceTreeNode(u)$ from T_{PRICE}, (2) the node pointed to by $ratingListNode(u)$ from the linked list which is part of some node in T_{RATING}; if deleting this linked list node made the list empty, we also delete the node pointed to by $ratingTreeNode(u)$ from T_{RATING}, and, finally, (3) delete the node u from T_{ID}.

We do at most three standard AVL deletes, each of which takes $O(\log n)$ in the worst-case, and a delete from a doubly linked list (given a pointer to the node to be deleted), that takes $O(1)$ in the worst-case. Hence the worst-case time complexity of DELETEBOOK is $O(\log n)$.

Answer to Question 3.

a. ***Algorithm***: Hash table with chaining.

We use a hash table T of size m and with a hashing function h (more about m and h later).

For each $i, 1 \leq i \leq n$: INSERT $B[i]$ in hash table T
For each $i, 1 \leq i \leq n$:
  SEARCH for $A[i]$ in hash table T
  if $A[i]$ is ***not*** found then output $A[i]$

---

b. ***Assumptions***:

1. SUHA (Simple Uniform Hashing Assumption): Every integer in $B[1..n]$ is equally likely to hash into any of the m slots of T (independent of where the other elements of B hash).

2. The size m of T is "proportional" to n, more precisely m is $\Theta(n)$ (actually $m \in \Omega(n)$ suffices).

3. Computing the hash function h on each integer of $B[1..n]$ takes $\Theta(1)$ time.

c. ***The expected running time of the algorithm is*** $O(n)$:

1. Each "INSERT $B[i]$ in hash table T" takes $\Theta(1)$ time. So inserting the n elements of $B[1..n]$ into T in the first For loop takes at most $O(n)$ time.

2. By Assumption 1, the expected length of each chain of T is $\alpha = n/m$.

3. By Assumption 2, $\alpha = \Theta(1)$.

   So the expected time for each "SEARCH for $A[i]$ in hash table T" is $O(1)$, and the expected time to do it for all i, $1 \leq i \leq n$, is at most $O(n)$.

d. ***The worst-case running time is*** $\Theta(n^2)$***:***

1. ***It is*** $O(n^2)$ ***because***:

   - As we explained above, inserting the n elements of $B[1..n]$ into T takes at most $O(n)$ time.
   - Since no chain in T can contain more than n elements, each "SEARCH for $A[i]$ in hash table T" takes at most $O(n)$ time.

     Since the algorithm does at most n such searches, this takes at most $O(n^2)$ time.

2. ***It is*** $\Omega(n^2)$ ***because***:

   The following "time-consuming" execution can occur:

   (i) *All the elements of B hash to the same slot $T[k]$ of T (they all "collide").* So they form a chain of length n starting at slot $T[k]$.

   (ii) *All the elements of A hash into the same slot $T[k]$, but B does not intersect with A.* So every "SEARCH for $A[i]$ in hash table T" traverses the entire list of n elements of B in slot $T[k]$ and "fails".

   Thus, the total time for all these n failed searches is $\Omega(n^2)$.