Solutions for Homework Assignment #3

**Answer to Question 1.**

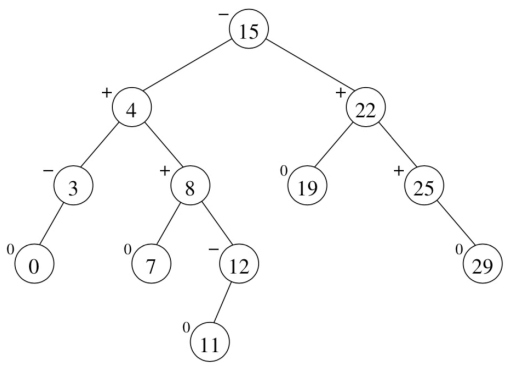


Figure 1: After all the insertions

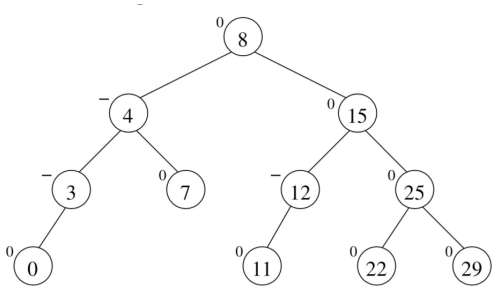


Figure 2: After deleting 19

**Answer to Question 2.**

**a.** The data stucture $D$ is represented as an augmented AVL tree. Each node $u$ of the augmented AVL tree contains information about a house listing. It contains fields *price*$(u)$, giving the price of the listing; *area*$(u)$, giving the floor area of the listing; and *maxarea*$(u)$ containing the maximum floor area of all the listings in the subtree rooted at $u$ (including $u$). Field *price* is used as the key of the node — thus, an in-order traversal of the tree visits all the listings *in non-decreasing price order*. In addition, each node $u$ contains the usual information of an AVL tree node: pointers to the left and right children as well as the parent, and the balance factor.

**b.** The operations are implemented as follows:

- INSERT$(D, x)$: *If $x$ is a pointer to a new listing, insert the listing pointed to by $x$ into $D$.*

  (1) Use the ordinary BST insertion algorithm to insert $x$ into the tree using the *price* field as the key. (Note that it does not matter if the key of $x$ already exists in the BST: if, while entering $x$, we encounter a record $y$ with the same key as $x$, by we can consider $x$ to be an immediate successor of $y$, and so we can continue the insertion of $x$ in the right subtree of $y$.) Node $x$ is now a leaf.

  (2) Traverse the path from the new leaf $x$ to the root of the tree. For each node $u$ along that path, update the *maxarea* field according to the following identity:

  $$\mathit{maxarea}(u) = \max\big(\mathit{area}(u), \mathit{maxarea}(\mathit{lchild}(u)), \mathit{maxarea}(\mathit{rchild}(u))\big)$$

  (where we assume that *maxarea*(NIL) $= -1$).

  (3) Traverse the path from the new leaf $x$ to the root of the tree, performing rotations and updating the balance factors as required by the ordinary AVL insertion algorithm. In addition, we update the *maxarea* field of every node $u$ involved in a rotation, according to the above identity.

  We stop rotations and updates to the balance factors as in the ordinary AVL insertion algorithm.

  Remark: The two traversals described above can be done in single pass where we first update the *maxarea* field and we then do the balancing part (which may also require a few *maxarea* field updates).

The time required by this algorithm is that required by the ordinary AVL insertion algorithm, i.e., $O(\log n)$, plus the time required to update the *maxarea* fields of the new node's ancestors, and the *maxarea* fields of the nodes involved in a rotation (a few nodes for each rotation). Since there are $O(\log n)$ such nodes and each *maxarea* update requires $O(1)$ time, the additional time is also $O(\log n)$. Thus, INSERT$(D, x)$ takes $O(\log n)$ time.

- DELETE$(D, x)$: *If $x$ is a pointer to a listing in $D$, remove that listing from $D$.*

  (1) Use the ordinary BST deletion algorithm to delete $x$ from the tree using the *price* field as the key. Since the tree is an AVL tree, the deletion of $x$ ultimately results in the removal of a leaf from the tree. Let $z$ be the parent of that leaf.

  (2) Traverse the path from $z$ to the root of the tree. For each node $u$ along that path, update the *maxarea* field according to the following identity:

  $$\mathit{maxarea}(u) = \max\big(\mathit{area}(u), \mathit{maxarea}(\mathit{lchild}(u)), \mathit{maxarea}(\mathit{rchild}(u))\big)$$

  (where we assume that $\mathit{maxarea}(\text{NIL}) = -1$).

  (3) Traverse the path from $z$ to the root of the tree, performing rotations and updating the balance factors as required by the ordinary AVL insertion algorithm. In addition, we update the *maxarea* field of every node $u$ involved in a rotation, according to the above identity.

  We stop rotations and updates to the balance factors as in the ordinary AVL insertion algorithm.

  Remark: The two traversals described above can be done in single pass where we first update the *maxarea* field and we then do the balancing part (which may also require a few *maxarea* field updates).

  The time required by this algorithm is that required by the ordinary AVL insertion algorithm, i.e., $O(\log n)$, plus the time required to update the *maxarea* fields of the removed leaf's ancestors, and the *maxarea* fields of the nodes involved in a rotation (a few nodes for each rotation). Since there are $O(\log n)$ such nodes and each *maxarea* update requires $O(1)$ time, the additional time is also $O(\log n)$. Thus, DELETE$(D, x)$ takes $O(\log n)$ time.

- MAXAREA$(r, p)$: *If $r$ is the root of a subtree of $D$, return the largest floor area, amongst all listings in this subtree whose price is $\leq p$; if there is no such listing, then return $-1$.*

```
MaxArea(r, p)
  if r = NIL then
      return −1
  else if p < price(r) then
      return MaxArea(lchild(r), p)
  else
      return max(maxarea(lchild(r)), area(r), MaxArea(rchild(r), p))
  end if
```

  Note that MAXAREA$(D, p)$ is simply MAXAREA$(r, p)$ where $r$ is the root of tree $D$.

  The time complexity of MAXAREA$(r, p)$ is proportional to the height of the AVL subtree of $D$ rooted at $r$. (This is because (i) each call at a node $u$ 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 $D$ contains $n$ listings, the height of any subtree of $D$ is $O(\log n)$, so the time complexity of MAXAREA$(r, p)$ is also $O(\log n)$.

**Answer to Question 3.**

We want to determine whether there is an $x_i \in X$ and a $y_j \in Y$ such that $x_i^2 + y_j^2 = d^2$, i.e., whether $\exists i, 1 \leq i \leq n$, and $\exists j, 1 \leq j \leq n$, such that $x_i^2 = d^2 - y_j^2$. Note that since every $x_i \in X$ and $y_j \in Y$ is an integer, $x_i^2 = d^2 - y_j^2$ can hold only if $d^2$ is also an integer.

**a. *Algorithm:***

To do so 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 $x_i^2$ in table $T$ (in front of the list starting at $T[h(x_i^2)]$) /* each INSERT takes $O(1)$ time */

For each $j, 1 \leq j \leq n$:

SEARCH for $(d^2 - y_j^2)$ in table $T$ (in the list starting at $T[h(d^2 - y_j^2)]$) /* this takes $O(\alpha)$ expected time */

If found then output "YES" and stop /* found an $x_i$ such that $x_i^2 = d^2 - y_j^2$ */
else output "NO"

---

**b. *Assumptions*:**

1. SUHA (Simple Hashing Uniform Assumption): For each $i$, $1 \leq i \leq n$, integer $x_i^2$ is equally likely to hash into any of the $m$ slots of $T$, independently from where $x_1^2, x_2^2, \ldots, x_{i-1}^2, x_{i+1}^2, \ldots, x_n^2$ hash into.
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 $h$ takes $\Theta(1)$ time.

**c. *The expected running time is* $\Theta(n)$*:***

- Each "INSERT $x_i^2$ in table $T$" takes $\Theta(1)$ time. So inserting $\{x_1^2, x_2^2, \ldots, x_n^2\}$ into $T$ takes $\Theta(n)$ time.
- By Assumption 1, the expected length of each chain of $T$ is $\alpha = n/m$. By Assumption 2, $\alpha = \Theta(1)$.
- So the expected running time for *each* "SEARCH for $(d^2 - y_j^2)$ in table $T$" is $\Theta(1)$, and the expected time to do this search for *all* $j$, $1 \leq j \leq n$, is $\Theta(n)$.

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

1. ***It is* $O(n^2)$ *because:***
   - As explained above, the first loop of the algorithm (which inserts $x_1^2, x_2^2, \ldots, x_n^2$) takes $O(n)$ time.
   - Now consider the second loop of the algorithm. Since no chain in $T$ can contain more than $n$ elements, each "SEARCH for $(d^2 - y_j^2)$ in table $T$" in this loop takes at most $O(n)$ time. Since the loop does at most $n$ such searches, the loop takes at most $O(n^2)$ time.
2. ***It is* $\Omega(n^2)$ *because:***

   The following "time-consuming" execution can occur.

   (i) *All the integers $x_1^2, x_2^2, \ldots, x_n^2$ hash into the same slot $T[k]$ of $T$* (i.e., they all "collide" into $T[k]$). So they form a chain of length $n$ starting in slot $T[k]$.

   (ii) *For every $j$, $1 \leq j \leq n$, $(d^2 - y_j^2)$ also hashes into slot $T[k]$, but there is no $i$ such that $x_i^2 = d^2 - y_j^2$.* So every "SEARCH for $(d^2 - y_j^2)$ in $T$" in the second loop of the algorithm "fails": each search traverses the whole chain of length $n$ at $T[k]$ and takes $\Omega(n)$ time. Thus the $n$ searches for $(d^2 - y_j^2)$ in $T$ take $\Omega(n^2)$ time.

*A similar algorithm that also works:*

---

For each $i, 1 \leq i \leq n$:

INSERT $d^2 - x_i^2$ in table $T$ (in front of the list starting at $T[h(d^2 - x_i^2)]$)

For each $j, 1 \leq j \leq n$:

SEARCH for $y_j^2$ in table $T$ (in the list starting at $T[h(y_j^2)]$) /* found a $y_j$ such that $y_j^2 = d^2 - x_i^2$ */

if $y_j^2$ is found then output "YES" and stop
else output "NO"

---

*Another algorithm that works:*

---

For each $i, 1 \leq i \leq n$:

INSERT $x_i$ in table $T$ (in front of the list starting at $T[h(x_i)]$)

For each $j, 1 \leq j \leq n$:

$x := \sqrt{d^2 - y_j^2}$

SEARCH for $x$ in table $T$ (in the list starting at $T[h(x)]$)

if $x$ is found then output "YES" and stop /* found an $x_i$ such that $x_i = \sqrt{d^2 - y_j^2}$ */
else output "NO"

---