Tutorial 4 Exercise Solutions

COMP526: Efficient Algorithms

28–29 October, 2024

Exercise 1. Starting from an empty binary search tree T, suppose the following elements are added in the specified order:

$$7, 4, 15, 11, 6, 17, 3, 9, 8, 12.$$

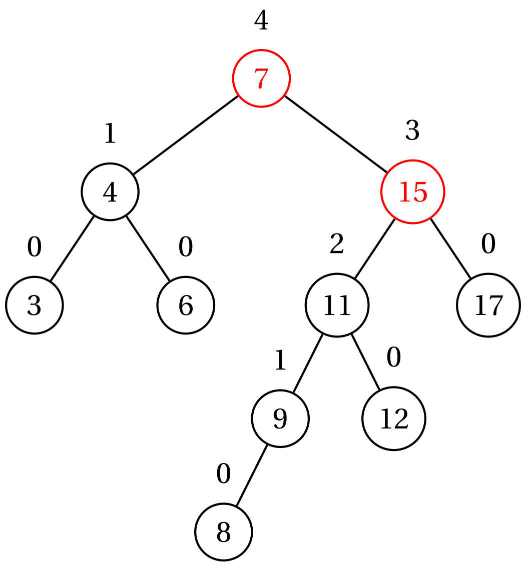
(a) Draw the T after all of the insertions have been completed.

(b) Indicate the height of every vertex in the tree.

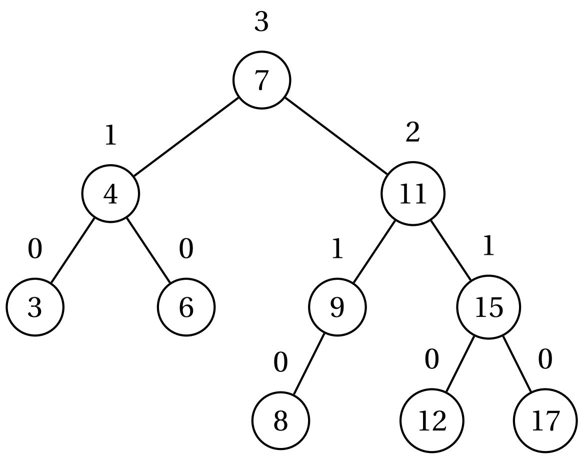
(c) Indicate on your picture all of the vertices that are *not* height balanced.

(d) Find a single rotation that can be performed to result in a height balanced tree, and draw the state of the tree after performing the rotation, along with the new heights of every vertex in the tree.

Solution. Here is the state of T after adding the elements. The heights are draw above each vertex, and the unbalanced vertices are colored red.



We can perform a single left rotation at vertex 15 to fix the imbalance. The resulting tree is shown below, with the new heights of each vertex labeled above them.



□

Exercise 2. Suppose we represent a binary (search) tree as the class BST, where each vertex is represented by a NODE class as follows:

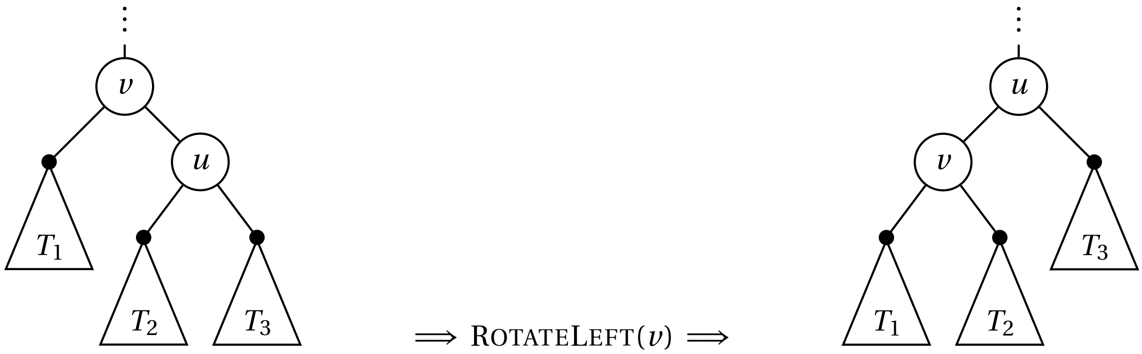
```
class NODE
    NODE PARENT
    NODE LEFTCHILD
    NODE RIGHTCHILD
    integer HEIGHT
    KEY
end class
class BST
    NODE root
    procedure FIND(x)    ▷ Return the NODE storing KEY x, or the NODE at which the search fails if there is no NODE with KEY = x.
        u, v ← root
        while v ≠ ⊥ and x ≠ KEY(v) do
            u ← v
            if x < KEY(v) then
                v ← LEFTCHILD(v)
            else
                v ← RIGHTCHILD(v)
            end if
        end while
        if v ≠ ⊥ then
            return v
        else
            return u
        end if
    end procedure
end class
```

Write pseudocode implementing the following functions:

(a) UPDATEHEIGHT(v) that updates the height of NODE v in the tree, assuming its children's heights are correct.

(b) INSERT(x) that inserts a new element with KEY = x if x is not already stored in the BST, and does nothing if x is already stored in the BST. Additionally, INSERT should update the heights of all vertices that changed as a result of inserting x in $O(h)$ time, where h is the height of the tree. (Hint: use the output of FIND so that you aren't reproducing the code there!)

(c) ROTATELEFT(v) that performs left rotation at vertex v (as depicted below). What is the running time of ROTATELEFT?



Solution. Here is pseudocode for UpdateHeight. Note that its running time is $O(1)$:

```
procedure UpdateHeight(v)
    h ← Height(LeftChild(v))
    h ← max{h, Height(RightChild(v))}
    Height(v) ← 1 + h
end procedure
```

For Insert(x), the following procedure uses the output of Find to determine if x is already stored in T. If not, it creates a new Node v to store x. To update the heights, it iterates over v's ancestors and updates their heights. Note that only ancestors of v may need to update their height, so the procedure iterates over at most h such ancestors.

```
procedure Insert(x)
    u ← Find(x)
    if thenKey(u) = x                    ▷ u will become x's parent if x is not in the BST
        return                           ▷ x is already in the BST
    end if
end procedure
v ← new Node
Key(v) ← x, Height(v) ← 0
Parent(v) ← u
if x < Key(u) then                       ▷ Determine if v should be right or left child of u
    LeftChild(u) ← v
else
    RightChild(u) ← u
end if
while u ≠⊥ and Height(u) < 1 + Height(v) do   ▷ Update heights fo v's ancestors
    UpdateHeight(u)
    v ← u
    u ← Parent(u)
end while
```

Finally, RotateLeft can be performed in $O(1)$ time:

```
w1 ← LeftChild(v)
w2 ← LeftChild(u)
w3 ← RightChild(u)
Parent(u) ← Parent(v)
Parent(v) ← u
Parent(w2) ← v
RightChild(v) ← w2
```

□

Exercise 3. An array a of length n storing integer values is called *bitonic* if there is an index b with $0 < b < n$ such that a is increasing for indices $0, 1 \ldots, b$ and decreasing for indices $b, b+1, \ldots, n-1$. That is, if $i < b$, we have $a[i] < a[i+1]$ and if $b \leq i < n-1$, then $a[i] > a[i+1]$. We say a is *tritonic* if there are indices b and c, with $0 < b < c < n-1$ such that a is (1) increasing between indices 0 and b, (2) decreasing between indices b and c, and (3) increasing between indices c and $n-1$.

(a) If a is bitonic of length n, explain how you can find b in time $O(\log n)$.

(b) (challenge) If a is tritonic, explain why finding b takes $\Omega(n)$ time in the worst case.

Solution. For part (a), we can use binary search. Given any index $i < n-1$, we can determine if $i < b$ by checking if $a[i] < a[i+1]$. The following variant of binary search will do the trick:

procedure BINARYSEARCH
  $i \leftarrow 0, k \leftarrow n-1$
  while $i < j$ **do**
    $j \leftarrow \lfloor (i+k)/2 \rfloor$
    if $a[j] > a[j+1]$ **then** ▷ This is precisely when $k \geq b$
      $k \leftarrow j$
    else
      $i \leftarrow j$
    end if
  end while
  return $i+1$
end procedure

For part (b), consider the following tritonic arrays. For $k = 0, 1, \ldots, n-2$, define the array a_k by

$$a_k[i] = \begin{cases} i & \text{if } k \neq i, i+1 \\ k+1 & \text{if } i = k \\ k & \text{if } i = k+1 \end{cases}$$

That is, each a_k consists of the the elements $0, 1, \ldots, n-1$ in sorted order, except values k and $k+1$ are swapped. Notice that any two arrays a_k and a_ℓ differ at at most four indices. Notice that a_k is tritonic with $b = k$ and $c = k+1$.

Now consider any algorithm A that finds the index $b = k$ for any tritonic array a. We argue by contradiction that A must read at least $\Omega(n)$ values of a in the worst case. To this end, suppose that A reads fewer than $n/2 - 2$ values of a for all input. Consider the process of constructing an array a in response to A's accesses to a as follows: whenever A accesses $a[i]$, we set the value $a[i] \leftarrow i$. Since A used fewer than $n/2-2$ values, there are still $n/2+2$ values that A never read before producing its output. Among these unread values, there are two distinct indices i and j such that A did not the values $a[i], a[i+1], a[j]$, or $a[j+1]$. Therefore, both a_i and a_j are consistent with the accesses made by A. If A outputs $b = i$, then complete $a \leftarrow a_j$ so that A produces the wrong output. On the other hand, if A outputs $b \neq i$, then set $a \leftarrow a_i$, so that A also produces the incorrect output. Therefore, if A uses fewer than $n/2-2$ accesses to a, it cannot correctly find b. Thus, A must use at least $n/2-1 = \Omega(n)$ accesses to a. □

Exercise 4. In lecture, we showed that building a binary heap containing n values can be performed in $O(n \log n)$ time by simply adding elements to the heap (represented as an array) using the BUBBLEUP procedure. Consider the following alternative HEAPIFY method that turns an arbitrary array into a heap:

procedure HEAPIFY(a, n) ▷ a is an array of size n
  $h \leftarrow \lceil \log_2 n \rceil$ ▷ h is the height of the tree representation of the heap
  for $\ell = h-1, h-2, \ldots, 0$ **do** ▷ Iterate over levels of the tree representation of the heap, from farthest from the root to closest to the root.
    for $i = 2^\ell - 1, 2^\ell, \ldots, 2^{\ell+1} - 2$ **do** ▷ Iterate over the vertices at level ℓ, i.e., the vertices at distance ℓ from the root
      TRICKLEDOWN(a, i)
    end for
  end for
end procedure

That is, HEAPIFY iterates over the heap elements from lowest level (farthest from the root) to highest level (ending at the root) and calls TRICKLEDOWN on each of the elements.

(a) Argue that after calling HEAPIFY(a), a is a binary heap (i.e., satisfies the heap property).

(b) Argue that the running time of HEAPIFY(a) is $\Theta(n)$.

You may assume that TRICKLEDOWN(a, i) obeys the following properties:

1. If TRICKLEDOWN(a, i) is called from an index i corresponding to level ℓ in the heap (i.e., i is at distance ℓ from the root), then it terminates after $c \cdot (h - \ell)$ operations.

2. If the the descendants of i's children satisfy the heap property, then after calling TRICKLEDOWN(a, i), i and its descendants satisfy the heap property as well.

Additionally, you may find the following equation useful: $\sum_{k=0}^{\infty} \frac{k}{2^k} = 2$.

Solution. For part (a), we argue by induction that after iteration ℓ of the outer loop, the heap property is satisfied for all values at levels $\ell' \geq \ell$. Before the first loop (i.e., $\ell = h$) this is satisfied because no vertex at level h has any children.

For the inductive step, we use property 2 of TRICKLEDOWN. Specifically, by the inductive hypothesis, all of indices at levels $\geq \ell$ satisfy the heap property. Therefore, after calling TRICKLEDOWN(a, i) from an index i at level $\ell - 1$, i and all of its descendants satisfy the heap property as well.

Applying the conclusion at $\ell = 0$, the entire array satisfies the heap property, as desired.

For part (b), we appeal to property 1 of TRICKLEDOWN above. First observe that for each level $\ell = 0, 1, \ldots, h$ there are at most 2^ℓ indices in level ℓ. Further, for each index

i at level ℓ, the running time $t(i)$ of TRICKLEDOWN at index i is at most $c \cdot (h-\ell)$. The total running time is $t = \sum_{i=1}^{n} t(i)$. We can break this sum up summing over the layers of the heap:

$$
\begin{aligned}
t &= \sum_{i=0}^{n-1} t(i) \\
&\le \sum_{\ell=0}^{h} 2^{\ell} \cdot c \cdot (h-\ell) && \text{(by property 1 of HEAPIFY)} \\
&= c2^h \sum_{k=0}^{h} \frac{k}{2^k} && \text{rewriting the sum} \\
&< c2^h \sum_{k=0}^{h} \frac{k}{2^k} \\
&\le 2c2^h = 2cn.
\end{aligned}
$$

Therefore, the running time is $2cn = O(n)$. Since the procedure reads all values of a, the running time is $\Omega(n)$ as well, so that the overall running time is $\Theta(n)$. □