Tutorial 4 Exercises

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:

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

(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.

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

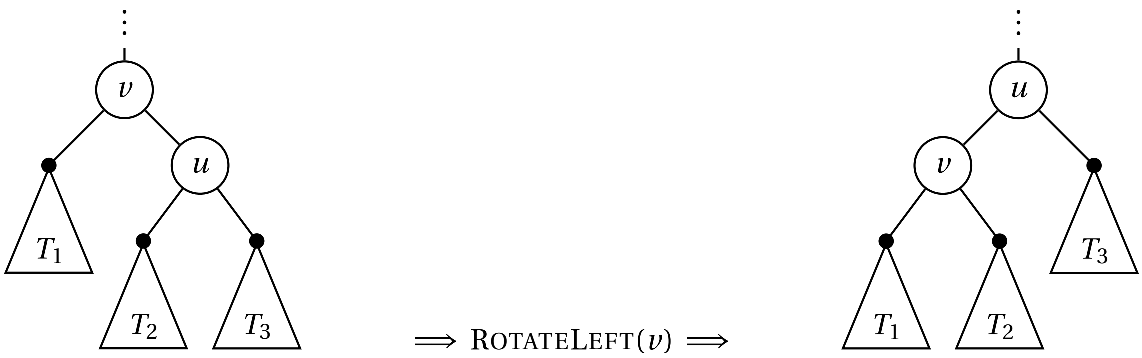
```
class Node
    Node Parent
    Node LeftChild
    Node RightChild
    integer Height
    Key
end class
class BST
    Node root
    procedure Find(x)        ▷ Return the Node storing Key x, or the Node at which the search fails if there is no Node with Key = x.
        u, v ← root
        while v ≠ ⊥ 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?



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.

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: $\sum_{k=0}^{\infty} \frac{k}{2^k} = 2$.