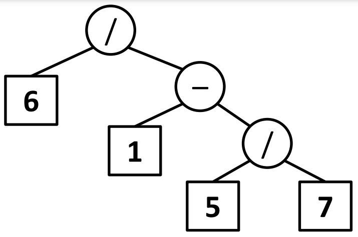
York University **EECS2011MZ: A3 Solutions** **Winter 2019**
Dept. of EECS **Instructor: Andy Mirzaian**

Problem 1. [15%] $\dfrac{6}{1-\frac{5}{7}} = 21.$



Problem 2. [35%]

Design Idea: We exploit the recursive structure of a binary tree T as root v, left subtree L, and right subtree R, and associate diameter of the tree with its root: $diameter(T) \equiv diameter(v)$. Suppose (p, q) is the farthest pair of nodes in T, i.e., $dist(p, q) = diameter(T)$.

Where could the two nodes p and q be in T? There are the following 4 cases . We take the maximum of the distances obtained from these cases.

1. $\boldsymbol{p \in L, q \in L}$**:** Then, $dist(p, q) = diameter(L)$. A recursive call on the left subtree will determine this value.
2. $\boldsymbol{p \in R, q \in R}$ **:** Then, $dist(p, q) = diameter(R)$. A recursive call on the right subtree will determine this value.
3. $\boldsymbol{p \in L,\ q \in R}$ **:** Then, $v = LCA(p, q)$, and p must be a deepest node in L, and q must be a deepest node in R. So,
$$dist(p, q) = dist(p, v) + dist(q, v) = \big(1 + height(L)\big) + \big(1 + height(R)\big).$$
4. $\boldsymbol{L = \emptyset}$ **or** $\boldsymbol{R = \emptyset}$ **:** If L is empty, we may take $p = v$. If R is empty, we may take $q = v$. The formula for $dist(p, q)$ in case (3) still applies if we define the height of an empty tree (i.e., a null external node) to be -1 (as base case). This also correctly tells us that a single-node tree has diameter 0 and height 0. For these to combine correctly with cases (1) and (2) above, we should define diameter of an empty tree to be -1 as base case.

In summery, we have the following recursive definitions for $height(v)$ and $diameter(v)$:

$$height(v) = \begin{cases} -1 & if\ v = null \\ 1 + \max\Big(\ height\big(left(v)\big), height\big(right(v)\big)\Big) & if\ v \neq null \end{cases}$$

$$diameter(v) = \begin{cases} -1 & if\ v = null \\ \max\left(\begin{array}{l} diameter\big(left(v)\big), \\ diameter\big(right(v)\big), \\ 2 + height\big(left(v)\big) + height\big(right(v)\big) \end{array} \right) & if\ v \neq null \end{cases}$$

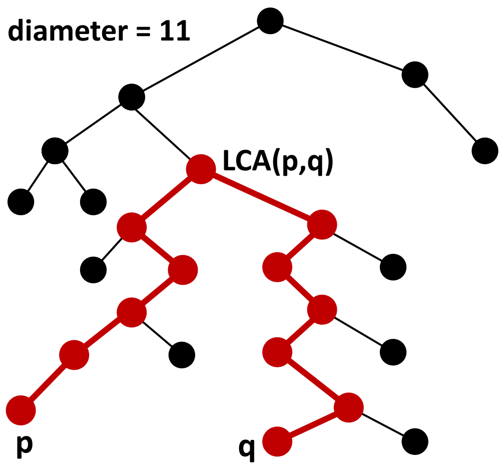
We can compute these by a post-order traversal of the binary tree with strengthened post-condition that returns the height as well as the diameter. So, our auxiliary recursive algorithm *diameter_Height(v)* will have a *Result* return type as a composite pair $\langle d, h \rangle$, where $d = diameter(v)$ and $h = height(v)$.
(Result can simply be an int array of length 2.)

The method *diameter(T)* calls the auxiliary method *diameter_Height(T.root())* and then returns the computed diameter (ignoring the computed height). The pseudo-code appears below.

```
Algorithm diameter ( T )          //  O(n) time
// Pre-Cond:  T is a binary tree.
// Post-Cond:  returns diameter of T.
1.   ⟨d , h⟩ ⟵ diameter_Height ( T.root( ) )
2.    return  d
end
```

```
Algorithm diameter_Height ( v )
// Pre-Cond:   v is root of a binary tree.
// Post-Cond:  returns ⟨ diameter(v) , height(v) ⟩ .
3.    if  v = null  then  return  ⟨-1 , -1⟩
4.    ⟨ dLeft , hLeft ⟩  ⟵  diameter_Height (  left(v)  )     // traverse  left  subtree
5.    ⟨dRight , hRight⟩  ⟵  diameter_Height ( right(v) )     // traverse right subtree
      //  Lines 6 - 8 :  "visit" node v:
6.   h ⟵  1 + max ( hLeft , hRight )
7.   d ⟵  max (  dLeft , dRight ,  2 + hLeft + hRight )
8.   return   ⟨d , h⟩
end
```

Time Analysis:
Since this is a post-order traversal algorithm, and visiting a node v (lines 3 and 6-8) takes O(1) time, the entire algorithm takes $O(n)$ time, where n is the number of nodes in the main tree.



Important Note: If we write two separate methods for height(v) and diameter(v), the former will take $O(n)$ time and will cause the latter to take $O(n^2)$ time (since the latter would have to call the former at each node based on the recurrence on the previous page). That is very inefficient! That's why we strengthened the post-condition by combining the two together into one method that returns both height and diameter as you see above.

Problem 3. [20%]

Design Idea: The algorithms appear below.

We form a max-heap out of the n given frequent flyers, where key is the frequent flyer millage. (See LS9, section 9.3.5 of [GTG] & PriorityQueue(Collection<? extends E> c) in the Java API.) We define the comparator comp so that its compare method works for max-heap, not min-heap (LS9, pp: 31-33), i.e., change the direction of the inequality in the heap-order invariant (LS9, p. 21). We place the data in a PQ heap, then call heapify() to build PQ as max-heap bottom-up in $O(n)$ time (LS9, p. 35).

We compute $k = \lfloor \log n \rfloor$ with $O(\log n)$ arithmetic operations by repeated halving starting from n.

We then do k removeMax() operations and collect the returned frequent flyers in topList to be returned. These are the k topmost frequent flyers (ties broken arbitrarily).

Each removeMax takes $O(\log n)$ time. So, the k removeMax ops take $O(k \log n) = O(\log^2 n)$ time. (Note: $\log^2 n$ stands for $(\log n)^2$.)

Therefore, The total running time is $O(n + \log n + \log^2 n) = O(n)$.

```
Algorithm intLog (int n )            //  O(log n) time
// Pre-Cond:  $n$ is a positive integer.
// Post-Cond: returns $\lfloor \log n \rfloor$.
1.  if $n \leq 1$ then return 0
2.  return 1 + intLog (n/2)          // integer division
end
```

```
Algorithm topFlyers ( A )            //  O(n) time
// Pre-Cond:  A[0..n-1] is an array of n frequent flyers, n = A.length.
// Post-Cond: returns the list of $\lfloor \log n \rfloor$ topmost flyers by millage flown.
3.  PQ ⟵ new HeapPriorityQueue(comp)
4.  PQ.heap ⟵ A                     // size = n = A.length
5.  PQ.heapify()                     // max-heapify in O(n) time
6.  k ⟵ intLog(A.length)            // O(log n) time
7.  topList ⟵ new ArrayList(k)      // Flyer element type, capacity k
8.  for i ⟵ 1 .. k do topList.add( PQ.removeMax() )   // $O(\log^2 n)$ time
9.  return topList
end
```

Problem 4. [30%]

Design Idea:

1. Partition the n elements of the collection into two groups: lowHalf and highHalf, with lowHalf containing the $\lceil n/2 \rceil$ smallest elements and highHalf contaning the remaining $\lfloor n/2 \rfloor$ elements. This forces the **balanced partition invariant:** lowHalf.size( ) − highHalf.size( ) ∈ $\{0,1\}$, and **key partition invariant:** max(lowHalf) ≤ min(highHalf).
2. The method getMed( ) requires access to the maximum element of lowHalf, and removeMed( ) should remove that accessed element. So, **lowHalf** should be a **max-heap.**
3. To insert a given element e, we first compare e.key with the current median. If e has a larger key we insert it in highHalf; otherwise, we insert it in lowHalf. If this invalidates the balanced partition invariant, we need to rebalance by transferring an element from one half to the other. If we need to move an element from lowHalf to highHalf, that element, based on the key partition invariant, must be maximum element of lowHalf. But if we need to move an element from highHalf to lowHalf, that element must be the minimum element of highHalf. So, we need to apply removeMax() on lowHalf or removeMin on highHalf. So, **highHalf** should be a **min-heap.**

```
Algorithm isEmpty( )            //  O(1) time
    return lowHalf.size() + highHalf.size() = 0
end

Algorithm getMed( )             //  O(1) time
    if isEmpty( ) then return null  else return lowHalf.max( )   // i.e., lowHalf[0]
end

Algorithm rebalance( )          //  O(log n) time    helper method for insert & removeMed
    if lowHalf.size( ) < highHalf.size( )      then  lowHalf.insert( highHalf.removeMin( ) )
    if lowHalf.size( ) > 1 + highHalf.size( )  then  highHalf.insert( lowHalf.removeMax( ) )
end

Algorithm insert( e )           //  O(log n) time
    if isEmpty() or e.key ≤ getMed( ).key then lowHalf.insert(e) else highHalf.insert(e)
    rebalance( )
end

Algorithm removeMed( )          //  O(log n) time
    if isEmpty( ) then return null
    e ⟵ lowHalf.removeMax( )
    rebalance( )
    return e
end
```