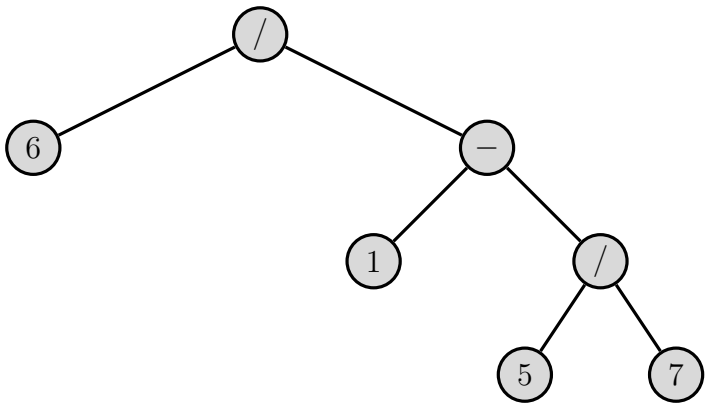
Assignment #3

Zhili Mai
Student Number: 215234842
Instructor: Andy Mirzaian

March 15th, 2019

1. Draw an arithmetic expression tree that has four external nodes, storing the numbers 1, 5, 6, and 7 (with each number stored in a distinct external node, but not necessarily in this order), and has three internal nodes, each storing an operator from the set $+, -, *, /$, so that the value of the root is 21. Any of these operators may be used more than once, and they are real (not integer) operators, i.e., they may operate on and return fractions.



2. Let T be a given binary tree with n nodes. The distance between two nodes p and q in T is the number of edges along the unique simple path between p and q, i.e., $d_p + d_q – 2d_a$, where a is the lowest common ancestor (LCA) of p and q, and d_x denotes the depth of node x in T. The diameter of T is the maximum distance between two nodes in T (i.e., the distance between the farthest pair of nodes in T). Give an efficient algorithm for computing the diameter of T and analyze its running time.

Algorithm 1: maxDist(root)

```
   /* Pre-Condition:  node is not empty                          */
   /* Post-Condition:  calculate the diameter of the tree       */
 1 if root is empty tree node then
 2 |  return 0
 3 end
 4 (left, right, dist) ← maxDistHelper(root)
 5 return left + right
```

Algorithm 2: maxDistHelper(root)

```
   /* Pre-Condition:  node root is not empty                    */
   /* Post-Condition:  return the max distance from left,
      right, and distan to LCA                                  */
   /* initial all variables to 0, l as left r as right, d as
      distance to current node                                  */
 1 l1, r1, d1, l2, r2, d2 ← 0
   /* search both left and right nodes if present, and
      increase distance with 1                                  */
 2 if hasLeft(root) then
 3 |   (l1, r1, d1) ← maxDistHelper(left(root))
 4 |   d1 ← d1 + 1
 5 end
 6 if hasRight(root) then
 7 |   (l2, r2, d2) ← maxDistHelper(right(root))
 8 |   d2 ← d2 + 1
 9 end
   /* calculate the max distance with current node from left
      to right                                                  */
10 maxDist ← max(l1, r1) + d1 + max(l2, r2) + d2
11 if maxDist > max(l1 + r1, l2 + r2) then
12 |   return (max(l1, r1) + d1, max(l2, r2) + d2, 0)
13 end
   /* retrun larger total length from left or right           */
14 if l1 + r1 > l2 + r2 then
15 |   return (l1, r1, d1)
16 end
17 if l1 + r1 < l2 + r2 then
18 |   return (l2, r2, d2)
19 end
20 return (max(l1, r1) + d1, max(l2, r2) + d2, 0)
```

In this question, we need to design an algorithmic to find the max distance in the tree. We have to do a scan the tree from bottom of tree until we find max distance. Otherwise, there is no way to know the longest path.

We need a function that keeps trace of longest path to current node that it is traveling. Then we need to keep trace of the longest distance with return value of function. We can construct a function that find the longest path from his left or right side. The last thing is that we can travel the tree in a recursive way. So this is straight forward. I construct a function $maxDist(root)$ (Algorithm

1) that returns the longest distance from the tree. This is not enough, we need to keep trace of three value, path to the left, right, and LCA. Then we can have a helper method $maxDistHelper(root)$ (Algorithm 2) that return the object that contains the three value.

The function, $maxDistHelper(root)$, search recursively. It should travel the tree in post-order which will start at the bottom. Then we can count the the distance from the bottom. This is an easier way to scan the tree.

To achieve $maxDistHelper$ its pre-condition is the tree is not empty and the post condition is that return the longest path. It maintains and keep trace of the max distance from the left and right node. Each step it travel to its left, and right (line 2 to 9). And compare the max distance path including the current node that it is traveling (line 10 to 13). If it is not longer than the left side or right side, it returns the side with longer distance (line 14 to 19). Then it updates the distance to next LCA (add one). If the length are the same, it returns the longest path from its left and right (line 20). The base case is the distance are all zero if the left and right node are not present.

The $max(a, b)$ is a helper function that return the number that are larger. It runs on $O(1)$.

The function runtime of $maxDist$ is base on $maxDistHelper$ since it has a single call the $maxDistHelper$. $maxDistHelper$ runs on $O(n)$ since it travel all the node in the tree. Each function call will check left and right node if it is present. There is no extra call.

Loop Invariant: The node (root) of a tree/subtree is not nil.
proof: at line 2 and line 6, the algorithm make sure that it never calls the function with an empty node.
Maintain: the value are the longest path from left to right.
proof: from line 10 to 20. It checks all case.

Case 1: $max(l1, r1) + d1 + max(l2, r2) + d2 > max(l1 + r1, l2 + r2)$
In this case, $max(l1, r1)$ is the longest path from left and $max(l2, r2)$ is the longest path from the right. $d1$ and $d2$ are the distance to current node. $max(l1, r1)+d1+max(l2, r2)+d2$ is the longest we can get in the tree/subtree. If it is larger than both side, it is going to be the longest that we can get. That is what we need to return. Also we reset the distance to LCA to 0 since it is connected to new LCA.

Case 2: $l1 + r1 > l2 + r2$
In this case, left side has a lager subtree then the right side. Then we return the left side.

Case 3: $l1 + r1 < l2 + r2$
In this case, right side has a lager subtree then the left side. Then we return the right side.

Case 4: default
In this case, we return the longest path from the left and right and update the path link to current node. They are $max(l1, r1) + d1$ and $max(l2, r2) + d2$. Also we reset the distance to LCA to 0 since it is connected to new LCA.

The case that $max(l1, r1) + d1 + max(l2, r2) + d2 < max(l1 + r1, l2 + r2)$ means that we have a larger subtree the needs to return. Then it checks for the larger side.

If they are the same, that means we have both tree are large. That way we need to update LCA. That gives might a longer path. It is basically all the case we need to check.

Each call will then add 1 to distance since the parent node are the one calling their child nodes. This is because the distance to their LCA are increased.

One boundary case is that the root is null in the beginning which is handle by the $maxDist(root)$ that check if the root is empty and reduce the return object to an integer that indicate the length.

Progress: Each call is a post-order travel. Therefore, it is making progress with their child nodes. Travel through each child node.

Termination: The function terminate when there are no more node to travel. Since it is binary tree that travel both side if present. So it won 't run forever because of the number of node are finite.

3. Tamarindo Airlines wants to give a first-class upgrade coupon to the top $log(n)$ of their frequent flyers, based on the number of miles accumulated, where n is the number of the airlines' frequent flyers. The algorithm they currently use, which runs in $O(nlog(n))$ time, sorts the flyers by the number of miles flown and then scans the sorted list to pick the top $log(n)$ flyers. They have hired you as their chief software engineer.Give an algorithm that identifies the top $log(n)$ flyers in $O(n)$ time.

   $O(nlog(n))$ is basically an insertion into binary tree list for a n items, and each insertion takes $O(log(n))$. Then remove This is too longer than $O(n)$. We need to use different way to construct it in different way.

   There are different way to construct tree. It is Bottom-up Heap Construction (Algorithm 3). It construct heap in linear time which is $O(n)$. Then we

removes $log(n)$ people from the heap that takes $O(log(n) * log(n))$ since each remove takes $O(log(n))$. So the total run time would be $n + log(n)log(n)$ and it is $O(n)$.

Algorithm 3: construct(tree)

```
   /* Pre-Condition:  tree is a n item array                    */
   /* Post-Condition:  constructed heap implemented with a n
      item tree array                                          */
1  for i ← len(tree) − 1 to 0 do
   |  /* down heap for the node at i of heap array            */
2  |  downheap(tree, i)
3  end
```

The algorithmic can be designed as topLogN(Flyer flyers[]) (Algorithm 4) that construct a Heap with array of flyers using $O(n)$ and return an array with size of $log(n)$ in $O(log(n) * log(n))$.

Algorithm 4: topLogN(tree)

```
   /* Pre-Condition:  tree is a n item array                    */
   /* Post-Condition:  list of top log n in heap               */
1  construct(tree)
   /* initialize array                                         */
2  list ← array with size of log(len(tree))
3  for i ← 0 to log(len(tree)) do
4  |  list[i] ← remove(tree)
5  end
6  return list
```

4. Dynamic Median Finder (DMF)
   DMF ADT that need $insert()$ and $removeMed()$ are $O(log(n))$ and $getMed()$ returns the median in $O(1)$. The best ADT that meets the requirement is Binary Search Tree. Then keep the median at the root. Left and right subtree keeps the number smaller and larger than median. That prefer insert the number at the right if new number cause tree to become unbalance.

   However, we we need to construct the with two heap. Also, in the collection of n elements, the median is $n/2$. Then we keep two heap the one is *minheap*, the other is *maxheap*. However, there are no heap implementation in java, so I use *PriorityQueue* as the heap since it is an array implementation binary tree. For *minheap*, it will sort numbers in descending order. It keeps the biggest number at the top. For *maxheap*, it will sort numbers in ascending order. It keeps the lowest number at the top. However, we still need to keep the tree balance,

which means we need to balance if the other tree have more element then the other tree. It will take the extra element and put the top one into other tree, so the different in size will be at most 1. Therefor, we can get midpoint very fast since we sort and keeps the median at the top of trees (only one if n is odd, on both tree if n is even).

For $insert(e)$, we need to insert the element base on median. If the value is large then current median, then it add e into *maxheap*. Otherwise, it will add e into *minheap*. Then we need to balance the tree if the tree is not balanced. This operate in $O(log(n))$ since the worst case is two inserts and one poll that makes $3log(n)$.

For $balance()$ it compare if the tree is too large and poll it from the large tree with $O(log(n))$. Then insert it in to smaller tree with $O(log(n))$.

For $getMed()$ since it is return at the top so it would be $O(1)$. This function does primary operation to know which tree contains the median. The median is at the tree with larger size. If the size are the same, it will be arbitrary at the left tree aka *minheap*.

For $removeMed()$ it removes the current median from the tree. The removal action in tree operate in $O(log(n))$. We use the same rule as *getMed* to get an arbitrary median so we can easily remove the median.

```
1  import java.util.Comparator;
2  import java.util.PriorityQueue;
3
4  public class DMF<E> {
5      private PriorityQueue<E> maxheap;
6      private PriorityQueue<E> minheap;
7      private Comparator<? super E> comparator;
8
9      public DMF(Comparator<? super E> comparatorMin, Comparator<? super
       E> comparatorMax) {
10         this.maxheap = new PriorityQueue<>(comparatorMin);
11         this.minheap = new PriorityQueue<>(comparatorMax);
12         this.comparator = comparatorMin;
13     }
14
15     public void balance() {
16         if (Math.abs(maxheap.size() - minheap.size()) > 1) {
17             if (maxheap.size() > minheap.size()) {
18                 minheap.add(maxheap.poll());
19             } else {
20                 maxheap.add(minheap.poll());
21             }
22         }
```

```
23      }
24
25      public void insert(E e) {
26          if (isEmpty()) {
27              minheap.add(e);
28          } else if (comparator.compare(getMed(), e) < 0) {
29              maxheap.add(e);
30          } else {
31              minheap.add(e);
32          }
33          balance();
34      }
35
36      public E getMed() {
37          if (minheap.size() >= maxheap.size()) {
38              return minheap.peek();
39          }
40          return maxheap.peek();
41      }
42
43      public E removeMed() {
44          if (minheap.size() >= maxheap.size()) {
45              return minheap.poll();
46          }
47          return maxheap.poll();
48      }
49
50      public int size() {
51          return maxheap.size() + minheap.size();
52      }
53
54      public boolean isEmpty() {
55          return maxheap.isEmpty() && minheap.isEmpty();
56      }
57  }
```