EECS2011A: Fundamentals of Data Structures, Fall 2017

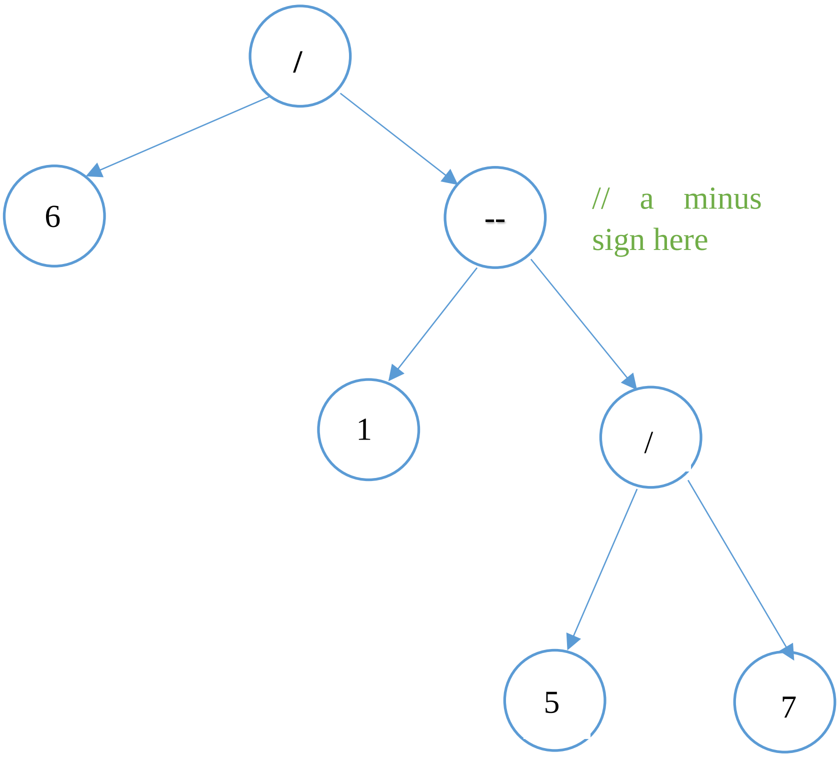
Assignment 3

Student Full Name: Chenxing Zheng

Student EECS account: cxing95

Student ID number: 214634901

Problem 1:



Result = 6 / (1 – 5/7)
= 6 / (2/7)
= 21

Problem 2:

The diameter of a binary tree T should be one of these cases:

1. The diameter of the left subtree. (so recursion)
2. The diameter of the right subtree.
3. The height of left subtree + height of right subtree + 2 (2 more edges to go through the root)

Pseudo-code:

```
/*
* Use this result class to keep track of the height, so we    don't need to call a
* separate getHeight() method whose time complexity would be O(n)
*/
class Result{
    int height = 0;
}
......
int diameter(TreeNode p, Result result){
    Result leftResult = new Result();
    Result rightResult = new Result();

     //base case
    if( isExternal(p) || p == null){
        result.height = 0;
        return 0;
    }
    else{// internal node, call the recursive method
    int leftDiameter = diameter(p.leftChild, leftResult);
     int rightDiameter = diameter(p.rightChild, rightResult);

     // update the height of current node p
    result.height = max(leftResult.height, rightResult.height) + 1;

     // return the diameter
    return max( max(leftDiameter, rightDiameter),
                    leftResult.height + rightResult.height + 2);
    }
}
......
main method(){
    // ...construct the tree
    // initialize the first instance of Result
    Result result = new Result();
    int diameterOfTree = diameter( root, result)
}
```

Time complexity:

For diameter(p, result), this method traverse all the nodes. In each node, there are computations taking constant time. So O(n).

Or by induction:

$$
\begin{aligned}
T(n) &= 2T(n/2) + c \\
&= 4T(n/4) + 3c \\
&= 8T(n/8) + 7c \\
&= nT(1) + (n-1)c \\
&= O(n)
\end{aligned}
$$

Problem 3:

1. Compute logn by normal arithmetic operations.
2. Build a maxHeap(as defined in Problem4) by bottom-up heap construction, to store these flyers.
3. Search and return the top logn flyers.

Pseudo code:

```
class Flyer{
    int miles; // the key of each flyers
    int id;
}
......
int num; // Global varible = # of flyers to find = logn
......
/* n is the number of frequent flyers
* Select top logn flyers from them
*/
ArrayList<Flyer> topFlyers( int n, ArrayList<Flyer> frequentFlyers){
    // normal arithmetic operations to get logn
    int logn = 0;
    for(; n>1; n/=2){ // O(logn)
    logn++;
```

```
        }
        num = logn;
        ArrayList<Flyer> result = new ArrayList<Flyer> [logn];

        MaxHeap maxHeap = bottomUpConstruction( frequentFlyers); // O(n)

        searchTopFlyers(maxHeap.max(), result);

        return result;

}
......
// Pre-order traversal of a heap
void searchTopFlyers( Entry maxHeapTop , ArrayList<Flyer> result){
        if(result.size() == logn) return;

        result.add(maxHeapTop);
        num++;

        searchTopFlyers(maxHeap.max().leftChild(), result);
        searchTopFlyers(maxHeap.max().rightChild(), result);

}
```

Time complexity:

1. operations to get logn:

    $T(n)= T(n/2) + c$
    ...
    $= T(1) + c\log n$
    $= O(\log n)$

2. bottomUpConstruction():

    To construc a heap, we first construct its left and right subheap (already heapified), then merge them with the root(which takes O(logn) time).

    The last but one level has 2^(h-1) node, they take O(1) time to downheap();
    The last but two level has 2^(h-2) node, they take O(2) time to downheap();
    ......
    The top(root) take O(logn) time to downheap().

    $T(n) = 2^{(h-1)} \times 1 + 2^{(h-2)} \times 2 + \ldots\ldots +1 \times (h-1)$
    $2T(n) = 2^{(h)} \times 1 + 2^{(h-1)} \times 2 + \ldots\ldots +2 \times (h)$
    So $2T-T = T$
    $= 2^{(h)} + 2^{(h-1)} + 2^{(h-2)} + \ldots\ldots + 2 + h$
    $= 2^{(h+1)} - 2 + h$ // h = logn

$= 2n - 2 + \log n$
$= O(n)$

3. searchTopFlyers():
   $T(n) = 2T(n/2) + c$
   $= O(n)$ // as shown in Problem2.

Problem 4:

This DMF ADT use two heaps.

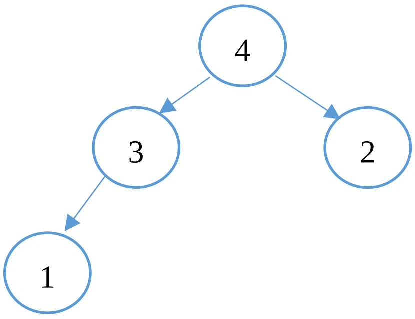
1. minHeap: key(v) >= key(parent(v)), the root is the minimum. (This is the heap shown in class.)
2. maxHeap: key(v) <= key(parent(v)), the root is the maximum. (Easy to implement as we have learned minHeap. Just bring the greater element up.)

What I want is:
The smaller half of elements are stored in the maxHeap. Their maximum is the root.
The greatrer half of elements are stored in the minHeap. Their minimum is the root.

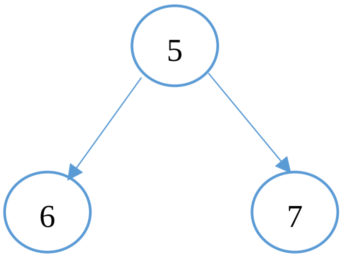
So, the median of all the elements is always the root of maxHeap as shown below.

Elements[] = {5, 4, 2, 1, 3, 6, 7}

maxHeap:



minHeap:



Result: median = root of maxHeap = 4

The maxHeap.size() should always equal to minHeap.size() or minHeap.size() +1.

So if both of these two heaps are empty, I insert the first element into maxHeap.
……
Then if the element is greater than the root of maxHeap, insert it into minHeap, otherwise insert it into maxHeap.
After insertion, if maxHeap.size() > minxHeap.size() + 1, removeMax() from maxHeap and insert this element into minHeap. if minHeap.size() > maxHeap.size(), removeMin() from minHeap and insert this element into maxHeap.

Pseudo-code:

```
class DMF{
    MinHeap minHeap = new MinHeap();
    MaxHeap maxHeap = new MaxHeap();

    void insert( element e){
        if( maxHeap.isEmpty() && minHeap.isEmpty()) maxHeap.insert(e);
        // For simplification, I use > and < instead of compareTo().
        if( e<= maxHeap.max()) maxHeap.insert(e);
        else minHeap.insert(e);

        if( maxHeap.size() > minHeap.size() + 1){
            element temp = maxHeap.removeMax();
            minHeap.insert( temp);
        }

        if( minHeap.size() > maxHeap.size()){
            element temp = minHeap.removeMin();
            maxHeap.insert( temp);
        }
    }

    element getMed(){
        return maxHeap.max();
    }

    element removeMed(){
        return maxHeap.removeMax();
    }
}
```

Time Complexity:

1. insert(e): At most call one remove() method and one insert() method of heap. That's 2O(logn), so O(n).
2. getMed(): Return the root of maxHeap. That takes O(1) time.
3. removeMed(): It's the removeMax() of maxHeap, which takes O(logn) time.