

EECS2011A: Fundamentals of Data Structures, Fall 2017

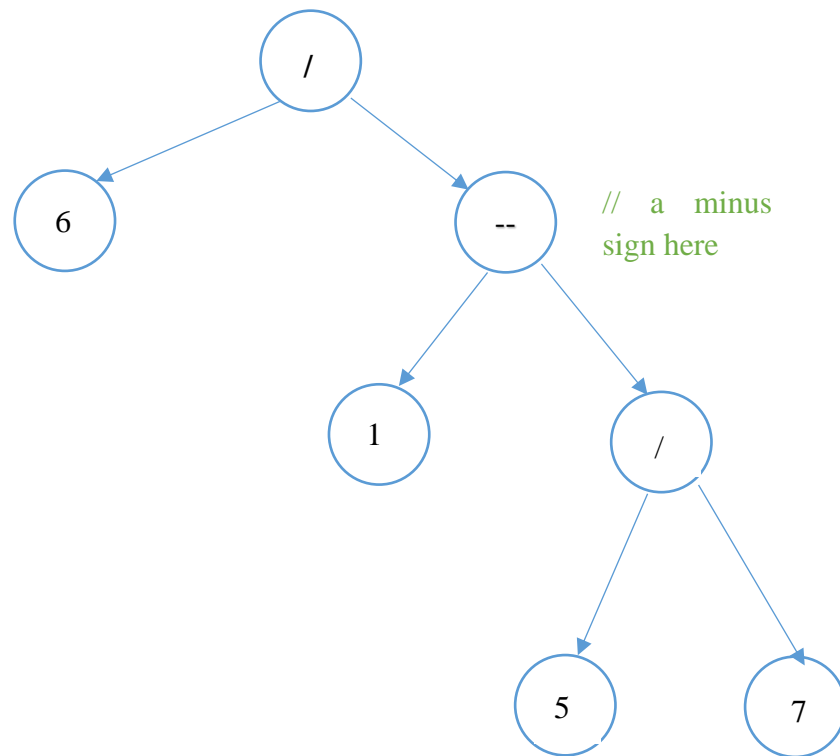
Assignment 3

Student Full Name: **Chenxing Zheng**

Student EECS account: **cxing95**

Student ID number: **214634901**

Problem 1:



$$\begin{aligned}\text{Result} &= 6 / (1 - 5/7) \\ &= 6 / (2/7) \\ &= 21\end{aligned}$$

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 $O(n)$.

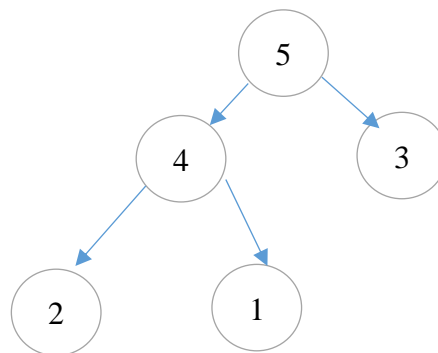
Or by induction:

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

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.
4. To return the top logn elements in a heap(a binary tree), we need to do a level-order traversal. I construct a queue called FlyerQueue to help.
The searchTopFlyers() get the first flyer in the queue, add it into the result, offer the left child and right child to the queue, then call itself.

For example, if the heap has {5,4,3,2,1} as its element.



First, add 5 into the queue.

Queue= [5]

Take the first element in the queue, add it into the result list, and add 5's children into the queue.

Queue= [4, 3]

Take 4, add 4's children.

Queue= [3, 2, 1]

Repeat until the queue is empty.

Pseudo code:

```
class Flyer{
    int miles; // the key of each flyers
    int id;
}
.....
int num; // Global variable = # 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)
FlyerQueue queue = new FlyerQueue( frequentFlyers.size()); // O(n)

Queue.enqueue(maxHeap.max());
searchTopFlyers(queue, result);

return result;

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

    Flyer temp = queue.dequeue();
    result.add( temp);
    num++;
    queue.enqueue( temp.leftChild());
    queue.enqueue( temp.rightChild());

    searchTopFlyers(queue, 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 construct a heap, we first construct its left and right subheap (already heapified), then merge them with the root(which takes $O(\log n)$ time).

The last but one level has $2^{(h-1)}$ nodes, they take $O(1)$ time to downheap();

The last but two level has $2^{(h-2)}$ nodes, they take $O(2)$ time to downheap();

.....

The top(root) take $O(\log n)$ time to downheap().

$$T(n) = 2^{(h-1)} \times 1 + 2^{(h-2)} \times 2 + \dots + 1 \times (h-1)$$

$$2T(n) = 2^h \times 1 + 2^{(h-1)} \times 2 + \dots + 2 \times (h)$$

$$\text{So } 2T - T = T$$

$$= 2^h + 2^{(h-1)} + 2^{(h-2)} + \dots + 2 + h$$

$$= 2^{(h+1)} - 2 + h \quad // \text{ } h = \log n$$

$$= 2n - 2 + \log n$$

$$= O(n)$$

3. searchTopFlyers():

$$T(n) = 2T(n/2) + c$$

$$= O(n) \text{ // as shown in Problem2.}$$

So time complexity of topFlyers() = $\max(1, 2, 3) = O(n)$

Problem 4:

This DMF ADT use two heaps.

1. minHeap: $\text{key}(v) \geq \text{key}(\text{parent}(v))$, the root is the minimum. (This is the heap shown in class.)
2. maxHeap: $\text{key}(v) \leq \text{key}(\text{parent}(v))$, the root is the maximum. (Easy to implement as we have learned minHeap. Just bring the greater element up.)

What I want is:

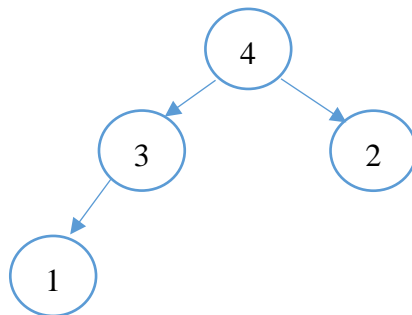
The smaller half of elements are stored in the maxHeap. Their maximum is the root.

The greater half of elements are stored in the minHeap. Their minimum is the root.

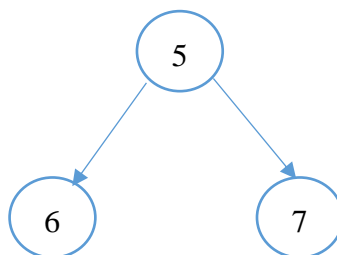
So, the median of all the elements is always the root of maxHeap as shown below.

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() > minHeap.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 $2O(\log n)$, so $O(\log n)$.
2. `getMed()`: Return the root of `maxHeap`. That takes $O(1)$ time.
3. `removeMed()`: It's the `removeMax()` of `maxHeap`, which takes $O(\log n)$ time.