CSE 373: Section 3

Analysis, Traversals and AVL

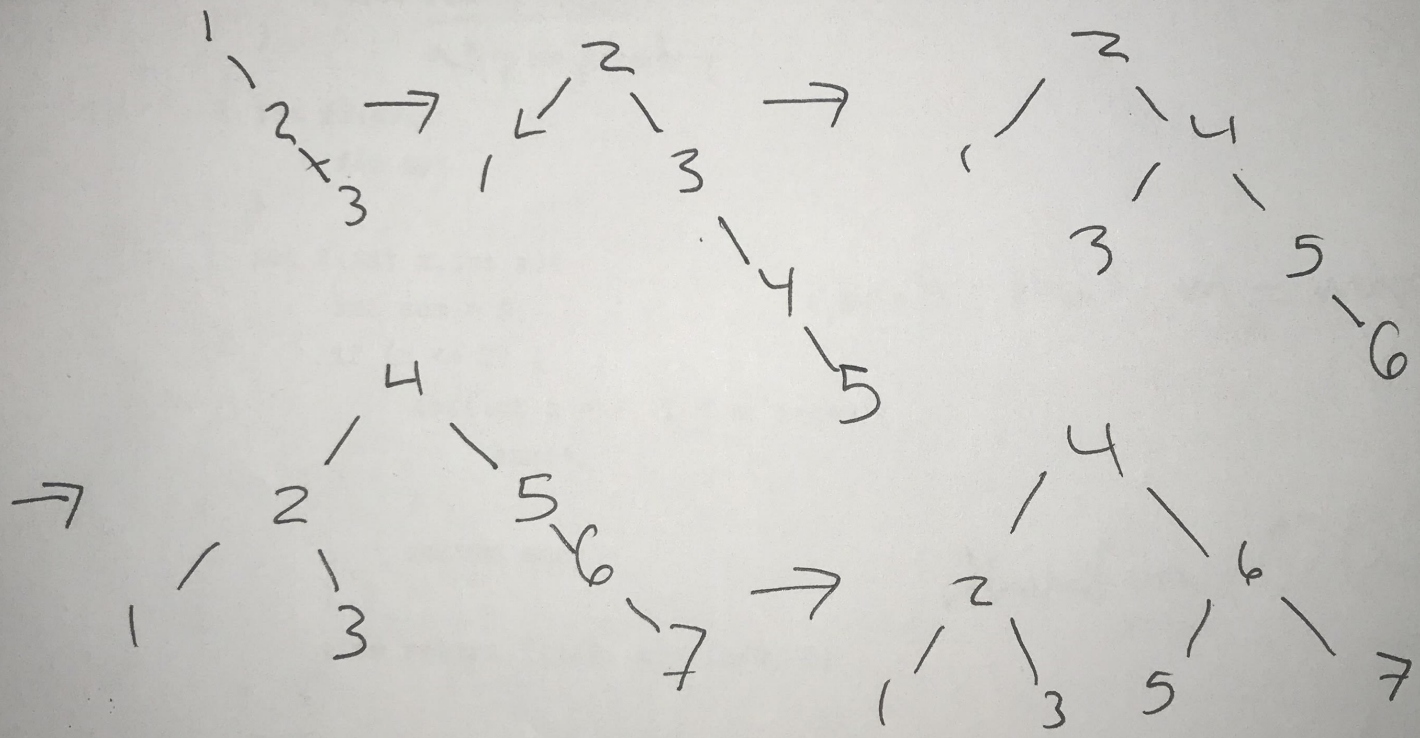
October 12th, 2017

AVL insertions

Show an AVL Tree as each of the following keys are added (in the order given). You may ignore their corresponding values.

$$\{1, 2, 3, 4, 5, 6, 7\}$$

Show the tree at each step. Observe how rotations occur at different levels of the tree



Produce the BFS traversal ordering as well as pre-order, in-order and post-order traversals of the tree

BFS: 4261357

post ~~pre~~ : 1325764

pre ~~post~~: 4213657

in: 1234567

Asymptotic Analysis

For the following methods, determine asymptotic runtime in terms of n

1.
```
void f1(int n){
    for(int i = n; i>0;i--){
        System.out.println("!");
    }
}
```

$O(n)$

2.
```
int f2(int n){
    if (n < 10) return n;
    else if(n < 1000) return f2(n-2);
    else return f2(n/2);
}
```

asymptotic

$O(\log n)$

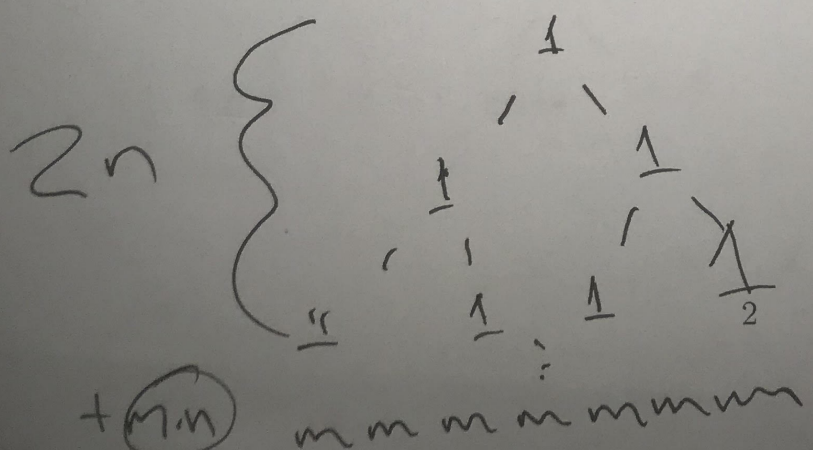
3.
```
int f3(n){
    f(n,n);
}
int f(int n,int m){
    int sum = 0;
    if (n <= 2) {
        for(int i = 2; i < m; i=i*2){
            sum++;
        }
        return sum;
    }
    else return f(n/2, m)+f(n/2, m);
}
```

recall that $m = \max(n)$

therefore $O(n^2)$

$T(1) = m$

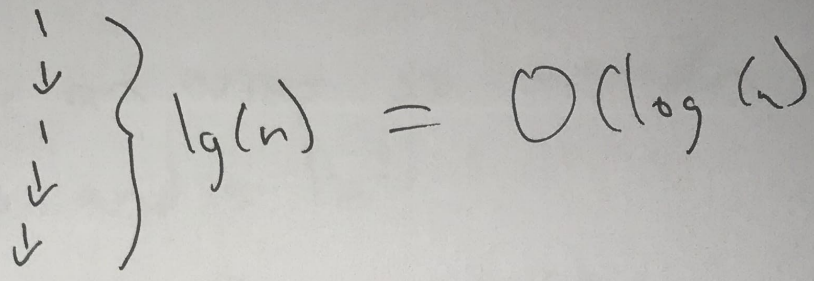
$T(n) = 1 + 2T(n/2)$



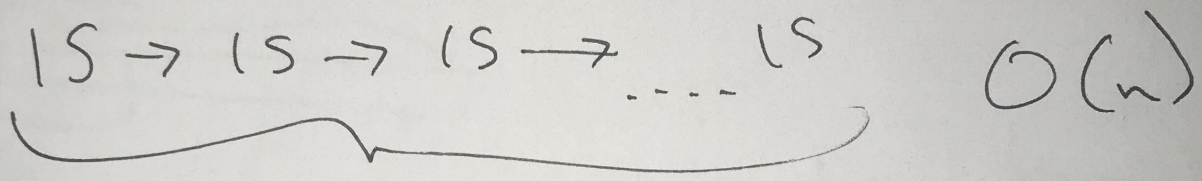
Recurrences

Given the following recurrences, use any of the methods provided in class (rolling out the recurrence, drawing a recurrence tree, master theorem) to find the tight bigO bound for the function in terms of n. You may assume that the base case runs in constant time for all functions.

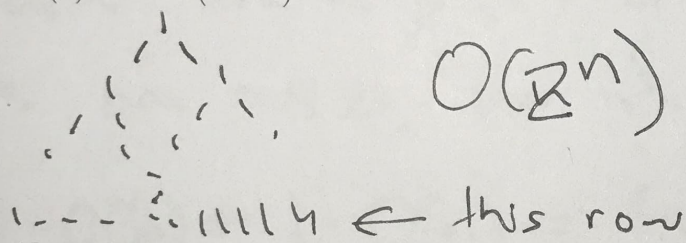
1. $T(n) = 1 + T(n/2)$



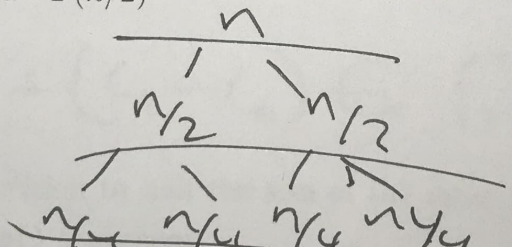
2. $T(n) = 15 + T(n-1)$



3. $T(n) = O(1) + 2 * T(n-1)$

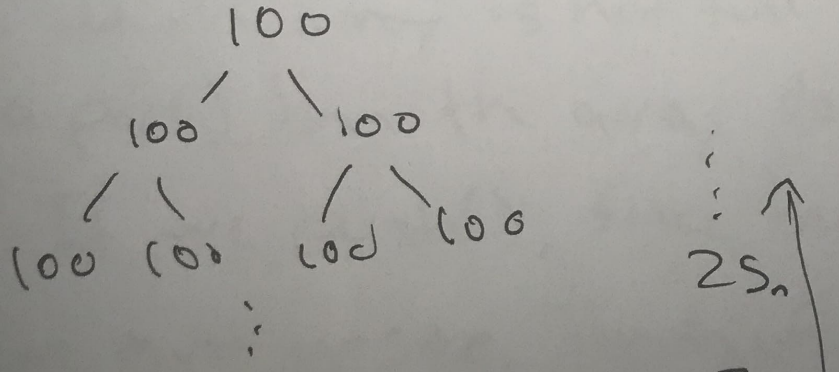


4. $T(n) = \underbrace{12n + \log n}_{O(n)} + 2 * T(n/2)$



n each rows
n there are log n
n rows ∴ O(n log n)

5. $T(n) = 100 + 2 * T(n/2)$



100n ≤ Σ 1/2^i ≤ 200n

O(n)

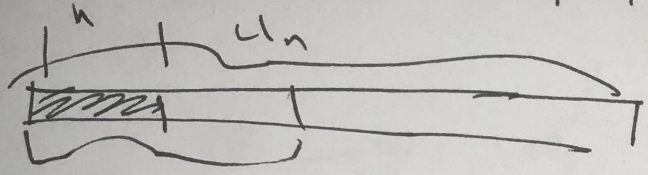
Amortized Analysis

Imagine an array implementation for a stack. Normally, resizes occur to make more room if the array is full. If the array becomes too empty, we often resize to avoid wasting memory.

- What is the amortized runtime of `pop()` if the array downsizes to half its size if the array is $\frac{1}{4}$ full?

Suppose an array has $4n$ possible elements and is half full.

Perform n pop() operations:



after the n pop() operations there is one resize, for all n operations

$$(n-1)\cdot O(1) + 1\cdot O(n) = c_0 n - c_0 + c_1 n + c_0$$

$$(c_1 + c_0)n + (c_1 - c_0) \in O(n)$$

- Why might it be a bad decision to half the size of the array when it is $\frac{1}{2}$ full? Consider alternating `push()` and `pop()` sequences.

If the array is half full and we perform a pop(), then the array downsizes.

If we push(), since the array is now full, we must upsize. Each of these operations are $O(n)$. Even amortized over n operations, we have $O(n)$ runtime