CS 61A CSM
Fall 2017

The Structure and Interpretation of Computer Programs
Kartik Kapur

Objects and Orders of Growth

1 Introduction

In this worksheet, we will be discussing Orders of Growth, Trees, and Objects.

2 Trees

To start off, we will think about current ways that we can store data. Well all we have right now are lists, and while that's great sometimes we want to sort our data structure in one that is more organized.This is the motivation behind having a Tree data structure.

2.1 Tree Structure

We will define the tree to be one that has a root and a set of children. But the question is, how do we make this? Well using our list data structure and some abstraction, we can set up a nifty tree.

```
1 def tree(label, branches[]):
2     return [label]+branches
3 def label(tree):
4     return tree[0]
5 def branches(tree):
6     return tree[1:]
```

Now this may seem a bit strange, a bunch of brackets are flying around with random tree parts as the variables, so let's break it down a bit further. We will define the tree to be some arbitrary value, a number, character, ice cream bar etc. We will then define a list of branches that will serve as that particular root's children.
Both of these items will be placed inside of a list, with the first item referring to the label and the second item referring

to the branches.
To get an idea of how exactly a tree works, let's walk through an example.

```
1 tree(1, [2,[3], [4]],[5,[6]])
```

The above example would look like the following:

```
1 [1, [2,[3], [4]],[5,[6]]]
```

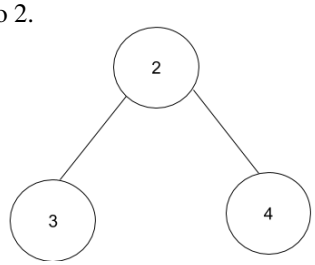
It may be a bit confusing to see how we get from this list to a tree so let's walk through it step by step. We will start by looking at the first element of the list [1, [2,[3], [4]],[5,[6]]]. We see that 1 is our first element, so we make that the root and now have the following tree:

Now that we have analyzed the 0th index of this list, let's move onto the children, [2,[3], [4]],[5,[6]]. We can see that there are 2 lists that contain all the other lists, [2,[3], [4]] and [5,[6]]. We progress by looking at the former. 2 is the first element of this list, so that means 2 will be the root of this tree.

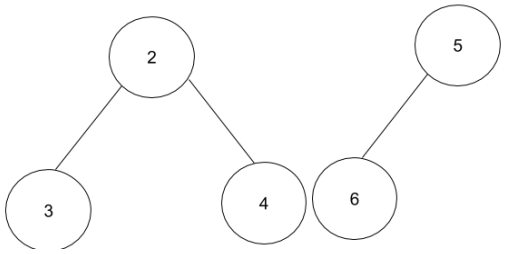
Now we will look at the next sublist, which is just composed of [[3],[4]]. Both of these will be a tree with the root of

themselves and no children, so they can be represented as follows

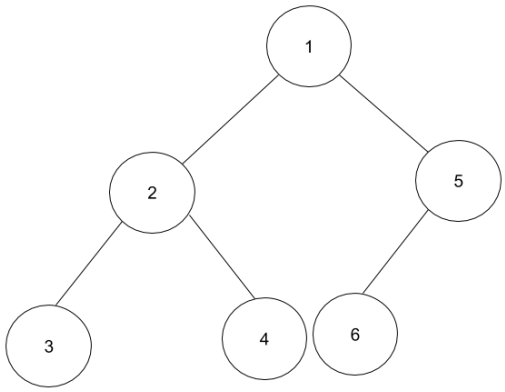
We have no other subtrees to visualize so let's go back up. Both 3 and 4 are children of 2, so we will "connect" them to 2.



We can do a similar process for [5,[6]] and get 2 subtrees that look as follows



We will connect them to our initial root, 1, and then we have finished constructing our tree!



The process of constructing trees can be a bit tedious; however, if one breaks it down systematically, it is relatively straightforward. One very important thing to remember is that a subtree of a tree is a tree itself. That is, a child of a tree is always a tree and will have a root and children, even if the children are null!

2.2 Utilizing Trees

We noted that our process of constructing a tree seems to be recursive in nature. As a result, we can expect that our traversal of these structures will follow a similar process. Using our knowledge of for loops and recursion, we can come up with a skeleton that works when we want to traverse all vertices in our tree.

```
1 ...
2 return [func_name(child) for child in tree.branches]
```

Basically what the above for loop does is return a new list that has the return values for all the nodes is a tree. This process tends to be used when we are constructing a new tree from our current values and simply want to do something to them (or if we want to loop over our tree for whatever reason).

3 Growth

When writing programs, it is vital that we keep track of how fast our program runs. Working with slow, inefficient code can make your life very hard- to avoid that, we will start by learning how to analyze code.
First, we need to go over some basic rules for find the runtime of a function.

- **Ignore Lower Order Terms:** When we are looking at asymptotics, we care about the highest term. This is because, as a function approaches infinity, it will be dominated by the highest order term. The proof of this can be done using some basic limit calculus so it will not be included in this document.
- **Ignore Constants:** Given two functions of the same order, we know that they will grow at the same rate. The constant will never affect how a function grows.

Now we will discuss the major asymptotic functions that we will discuss in the scope of this course.

- $\Theta(1)$ constant time
- $\Theta(log(n))$ logarithmic time
- $\Theta(n)$ linear time
- $\Theta(nlog(n))$ linearithmic time
- $\Theta(n^k)$ polynomial, where k is any constant number
- $\Theta(k^n)$ exponential, where k is any constant number.

We will briefly walk through how we can find the order of growth of some function.

```
1 def func(n):
2         while(n > 0):
3             n-=1
4             print('hello')
5         return 1
```

During each iteration of the loop we do a constant amount of work, since subtracting and print are both constant time operations. However, this loop runs a total of n times, which means that we run a total of $\Theta(n*1)$ times so $\Theta(n)$ times.

4 Questions

4.1 FunRuntime

Find the runtime of the following function in terms of n.

```
def fun(n):
    count = n
    while(n > 0):
        n-=1
    return fun(count-1)
```

Solution: This runs in $\Theta(n^2)$. We do a total of n recursive calls, and we are upper bounded by n work per call. You may be wondering how we're getting n^2 as it intuitively just seems like less. Well we do n work in the first call, n -1 in the second... and 1 in the last. So if we rewrite this we can get n+n-1+...1 = 1+2+...n-1+n = n^2 by identity.

4.2 Liebonacci

The basic fibonacci code is written as follows:

```
def fib(n):
    if(n <= 2):
        return n
    return fib(n-1)+fib(n-2)
```

However, this code is very inefficient, it runs in $\Theta(2^n)$ time. Write a better version of fibonacci that can run in linear time. You can use any exterior variables,arguments, or methods.

Solution:

```
cache = {}
def fib(n):
    if(n <2):
        return n
    if(n-1 not in cache):
        cache[n-1] = fib(n-1)
    if(n-2 not in cache):
        cache[n-2] = fib(n-2)
    return cache[n-1]+ cache[n-2]
```

This solution uses a technique called caching. Instead of throwing away the data you've used, you store it. This way, you have a significantly less number of calls to your function.