# Week 9

CSCA48 Winter 2020

## Height of a Tree

The height of a tree is the number of **edges** along the **longest path** from the root node to a leaf node

The height of an empty tree is -1

The height of a tree with one node is 0

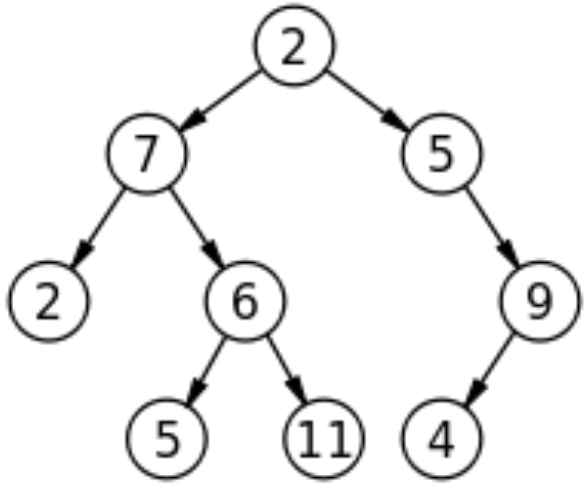
## Height of a Tree

The height of this tree is 3

A longest path is

$2 \rightarrow 5 \rightarrow 9 \rightarrow 4$

There are 3 $\rightarrow$ ‘s (edges)



## Depth of a Node

The depth of a node is the number of **edges** along the **path** from the root node to the **specified node**

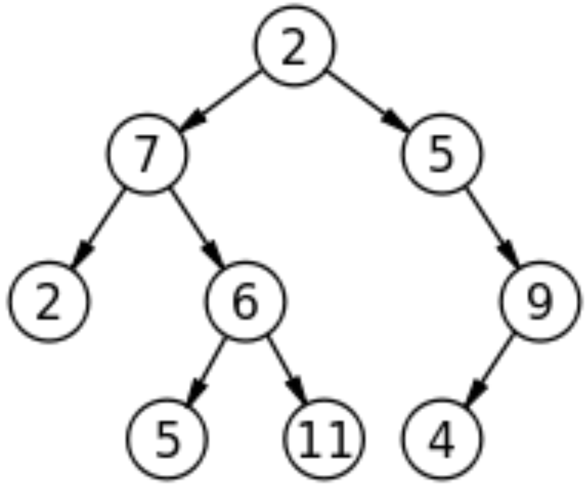
# Depth of a Node

The depth of the node 6 is 2

The path from the root to the node 6 is

2 → 7 → 6

There are 2 → ‘s (edges)



## Goal of this Tutorial

To teach you how to create a recursive solution for a problem

## What is recursion?

- U-sub, By-Parts, Partial fraction decomposition are all techniques to solve integrals

- Recursion is a technique used to solve problems

- Recursion is the process in which a function calls itself

## How does recursion work?

- Suppose I had a **measure()** function that uses a ruler of size 30cm (standard ruler) to measure the length of sticks!

- Now I have a very long stick that I want to measure, so I call the **measure()** function to measure it

- How can you do that?!? What if the stick is longer than 30cm?!? We only have a 30 cm ruler!!!

## How does recursion work?

We can measure the stick with the following technique:

1. If the stick can be measured, then good! Measure it! We are done, skip the rest of the steps.
2. The stick is too long! Break the stick into smaller pieces and measure the smaller sticks!
   **length1 = measure(smaller stick 1)** and **length2 = measure(smaller stick 2)**
3. Combine the solution of the smaller problems to solve the bigger problem!
   **fullLength = length1 + length2**

## How does recursion work?

Who does all the work? (breaking down problems and combining solutions)

A recursive function is responsible for all for the following

1. Being able to solve simple problems
2. Being able to break complex problems into smaller problems and solving the smaller problems (by **calling itself**)
3. Being able to combine the solutions to the smaller problems to formulate the solution for the larger complex problem

## How does recursion work?

We can measure a stick of any length with just a 30cm ruler! Amazing!

The result of recursion is the ability to solve any problem of any size from just knowing how to solve a simple small problem!

# Recursive Example (Very Very Very Easy)

The fibonacci sequence is a sequence of numbers that can be defined in the following way:

$$\mathrm{fib(n)} = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ fib(n-1) + fib(n-2), & n > 1 \end{cases}$$

## Recursive Example (Very Very Very Easy)

Here is a recursive function **int fib(int n)** that returns the n-th fibonacci number

```
int fib(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

$$\text{fib(n)} = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ fib(n-1) + fib(n-2), & n > 1 \end{cases}$$

# Two basic types of recursion

The difference between the following types is the way in which they break the problems down

**Divide and Conquer** - reduces the size of the problem by a factor of input size n
e.g. Cut problem of size n in half: $[n] \rightarrow [n/2] + [n/2]$

**"n-1" recursion** - reduces the size of the problem by a constant factor
e.g. Cut problem of size n by a constant factor 1: $[n] \rightarrow [1] + [n-1]$

## Creating a recursive algorithm

We create a recursive algorithm by following the “Recursion Recipe” created by Professor Nick Cheng at the University of Toronto

## Nick's Recursion Recipe (Do not Memorize)

1. Determine when the input is so **small** that the solution is easy to find

## Nick’s Recursion Recipe (Do not Memorize)

1. Determine when the input is so **small** that the solution is easy to find
2. Describe the solution to the problem when the input is **small**

## Nick’s Recursion Recipe (Do not Memorize)

1. Determine when the input is so **small** that the solution is easy to find
2. Describe the solution to the problem when the input is **small**
3. Take a *leap of faith*. **BELIEVE!!!**
   **If you aren’t able to solve the problem then you aren’t believing hard enough!**

## Nick’s Recursion Recipe (Do not Memorize)

1. Determine when the input is so **small** that the solution is easy to find
2. Describe the solution to the problem when the input is **small**
3. Take a *leap of faith*. **BELIEVE!!!**
   Suppose that for any input that is not **small**, you already had the solution to the same problem with any smaller input

# Nick's Recursion Recipe (Do not Memorize)

1. Determine when the input is so **small** that the solution is easy to find
2. Describe the solution to the problem when the input is **small**
3. Take a *leap of faith*. **BELIEVE!!!**
   Suppose that for any input that is not **small**, you already had the solution to the same problem with any smaller input
4. For any input that is not **small**, use the solution(s) to the same problem with smaller inputs to construct the solution to the original problem

## Nick's Recursion Recipe (Do not Memorize)

1. Determine when the input is so **small** that the solution is easy to find
2. Describe the solution to the problem when the input is **small**
3. Take a *leap of faith*. **BELIEVE!!!**
   Suppose that for any input that is not **small**, you already had the solution to the same problem with any smaller input
4. For any input that is not **small**, use the solution(s) to the same problem with smaller inputs to construct the solution to the original problem
   → Combining the smaller solutions to form a solution to the big problem is sometimes tricky. It may require some creativity.

## Too Long Didn’t Read the Recipe

1. Find all base cases
2. Figure out the inductive step (how do I get from problem n to a smaller problem of size n - 1)
3. Believe that your code works! Verify it by testing it.

## General Format of recursive problems

```
if (<condition>) {              ← Base Case
        // Code
} else if (<condition>) {       ← Base Case (Sometimes you need another one)
        // Code
...                             ← As many base cases as you need

} else {                        ← Recursive Case/Inductive Step
        // Code
}
```

## Example (Easy) - Linked List Insert at Tail

Write a function **insertTail** that inserts a new node into a linked list at the tail position.

The function will be provided a pointer to the head of the linked list and another pointer to the new node that needs to be inserted.

The function should return the head of the linked list after insertion.

## Example (Easy) - Linked List Insert at Tail

What is our input?

# Example (Easy) - Linked List Insert at Tail

**What is our input?**

A linked list

## Example (Easy) - Linked List Insert at Tail

**What is our input?**
A linked list

**What defines the size of our input n?**

## Example (Easy) - Linked List Insert at Tail

**What is our input?**

A linked list

**What defines the size of our input n?**

The size of the input n is the number of nodes in our linked list

## Example (Easy) - Linked List Insert at Tail

Base Case:

## Example (Easy) - Linked List Insert at Tail

**Base Case: n = 0**
Given the head of an empty linked list and a node to insert, we want to return the node itself

**Are there any more base cases?**

# Example (Easy) - Linked List Insert at Tail

**Base Case: n = 0**

Given the head of an empty linked list and a node to insert, we want to return the node itself

**Are there any more base cases?**

No

**Why don’t we have a base case for n = 1? n = 2? n = 3? n = 4? …**

## Example (Easy) - Linked List Insert at Tail

**Base Case: n = 0**
Given the head of an empty linked list and a node to insert, we want to return the node itself

**Are there any more base cases?**
No

**Why don’t we have a base case for n = 1? n = 2? n = 3? n = 4? ...**
We can most definitely have those bases cases, but we want to code as few bases cases as possible, but still maintain the ability to solve small problems.

More base cases won’t do harm, it just means extra coding for no reason.
**Why do more when you can do less?**

## Example (Easy) - Linked List Insert at Tail

**Inductive Hypothesis:**

Suppose that for any linked list of size (n - 1) or less, a call to **insertTail** will insert a given node into the linked list at the tail and return the head of the new linked list

## Example (Easy) - Linked List Insert at Tail

**Inductive Step: n > 0**

Given the head of a linked list with size n > 0 and a node to insert

- Call the **insertTail** function to insert the node into the **smaller** (size n - 1) linked list that starts with the next node (The head of the smaller linked list is head→next)

- The **return value** from this call should be the **new head** of the smaller linked list

- Therefore, replace the value of **head→next** with the return value of the function call

## Example (Easy) - Linked List Insert at Tail

```
struct node *insertTail(struct node *head, struct node *newNode) {
    // Base case (n = 0) empty list
    if (head == NULL) {
        newNode→next = NULL;
        return newNode;
    // Inductive step (n > 0)
    } else {
        head→next = insertTail(head→next, newNode);
        return head;
    }
}
```

## Example (Hard) - Depth First Search

See FloodFill.c

## Example (Hard) - Height Balanced Trees

See pdf notes