

# CS 310: Trees (Part I)

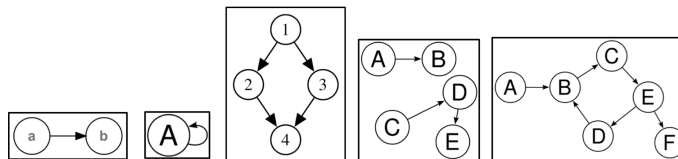
Connor Baker

February 19, 2019

## (Rooted) Trees

- A set of nodes and edges with no cycles
  - Edges point from parent to child
- One special node serves as the root
  - There is exactly one incoming edge per node per root
  - There is a unique path which traverses from the root to each node

## Examples

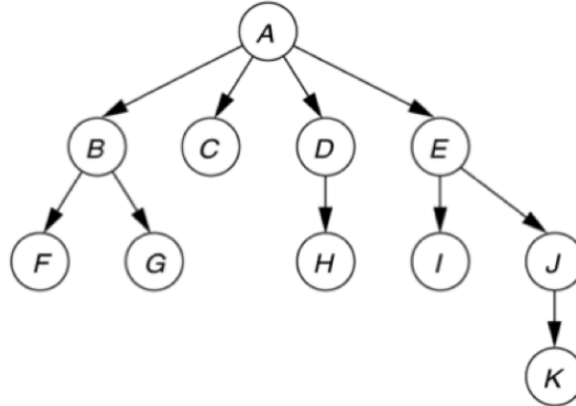


- From left to right:
  - Tree
  - Not a tree (it's cyclic)
  - Not a tree (more than one parent)
  - Not a tree (disjoint), or two trees, depending on how you look at it
  - Not a tree (more than one parent)

## Tree Definitions

- Node relationship:
  - The descendants of a node  $x$  are all the nodes that we can reach by following the paths starting from  $x$  (and sometimes includes the node  $x$  itself)
  - The ancestors of a node  $x$  are the nodes on the path from the root to  $x$  (and sometimes includes the node  $x$  itself)
  - Nodes are *siblings* if they have the same parent
- Special nodes:
  - The *root node* is a node that has no parent – there is only one root node in a rooted tree
  - A *leaf node* has no children

## Examples

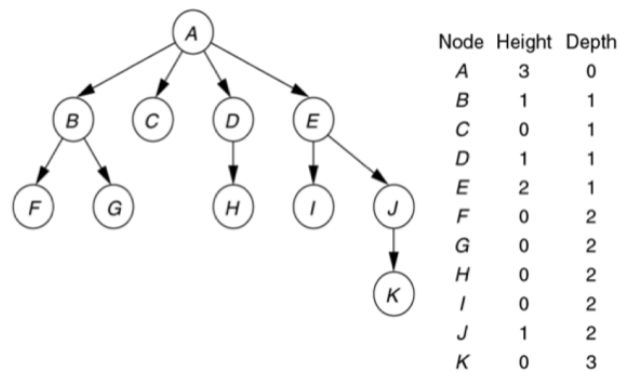


- What is the root node?
  - $A$
- What are the leaf nodes?
  - $F, G, C, H, I$  and  $K$
- What is the parent of  $H$ ?
  - $D$
- What are the children of  $B$ ?
  - $F$  and  $G$
- What are the ancestors of  $G$ ?
  - $B$  and  $A$
- What are the descendants of  $E$ ?
  - $I, J$ , and  $K$
- What are the siblings of  $C$ ?
  - $B, D$ , and  $E$

## Tree Properties

- Size (the number of nodes)
- Node height (the number of levels)
  - The *node height* is the length of the path (number of edges) from the node to the deepest leaf
  - The *tree height* is the height of the root
- Node depth (distance from the root)
  - The *depth* of a node is the length of the path from the root to the node

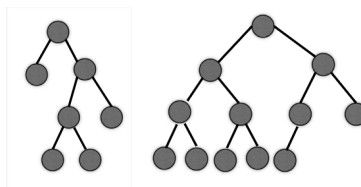
## Example



## Tree Features

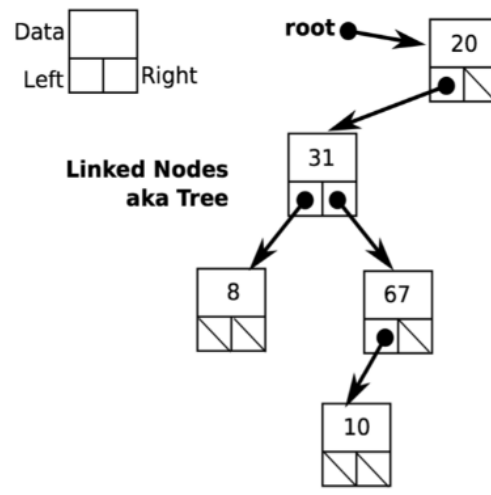
- Balanced tree
  - For a binary tree, the height of the left and right sub-trees of every node differ by 1 or 0 (as used by an AVL tree)
- Full tree
  - Every node other than the leaves has the maximum number of children
- Perfect tree
  - A full tree in which all leaves have the same depth
- Complete tree (an almost perfect tree)
  - A tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right and has no missing nodes

## Examples (Binary Trees)



- Left: a full tree
- Right: a balanced and (nearly) complete tree

## Trees Implemented with Linked Nodes

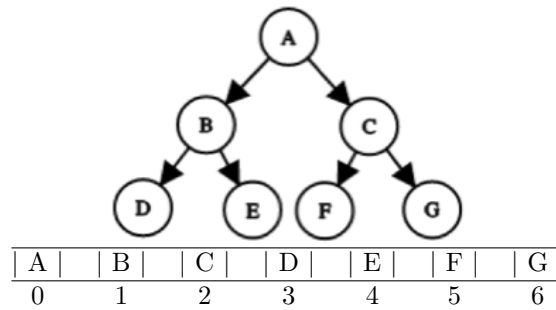


- Node structures are typically used for linked lists (owing to the singly-linked `next`/`data` idea)
  - Trees also use nodes
    - Data, pointers to children, possibly pointers to parents
    - Binary trees have left and right children:
- ```
class Node<T> {  
    T data;  
    Node<T> left, right;  
}
```
- Trees must have a finite number of children – however, the maximum number of children is arbitrary

## Trees Implemented with Arrays

- Store nodes in an array in level order
  - Start with the root
  - Left to right for each level
  - Reserve space for missing nodes
- Used to represent  $k$ -ary tree
  - Each parent can have at most  $k$  children
  - Binary tree when  $k = 2$
- Works the best with trees of a regular structures
  - Trees that are perfect or complete don't have (or have small) gaps
  - There's less wasted memory then

## Binary Tree with an Array



- Root is at index 0
- Parent at index  $p$ :
  - Left child:  $2p + 1$
  - Right child:  $2p + 2$
- Child at index  $c$ :
  - Parent at index  $\lfloor (c - 1) / 2 \rfloor$

## Binary Tree with an Array

- Complete binary trees
  - The array is filled left to right
- Arbitrary binary trees
  - Some elements of the array can be **null**

## Tree Implementations

- Arrays
  - Fast memory access
  - Must have a fixed branching factor
  - Inefficient use of memory
  - Common for regular and stable structures like complete binary trees
- Linked nodes
  - Easy to move around
  - Easier to deal with arbitrary trees

## Tree Applications

- Model hierarchical structures
  - File systems, class inheritance
- Store sorted data and support efficient search and insertion
  - Search trees or ordered trees in  $O(\log(n))$  time
  - Sorted lists and hash tables also come into play

- Many others
  - Expression trees
  - Huffman coding

## Common Tree Operations

- Searching for an item
- Adding items
- Deleting items
- Balancing
- Iterating through the tree
  - Selecting part of the tree

## Recursion

- *Recursion*: something defined in terms of itself
- *Function recursion*: when a function invokes itself
  - Two types of function recursion:
    - \* Direct: a function invokes itself
    - \* Indirect: `foo` invokes `bar`, and `bar` invokes `foo`

## Example: Factorial

- Mathematically:

$$n! = \begin{cases} 1 & n \leq 1 \\ (n-1)! & n > 1 \end{cases}$$

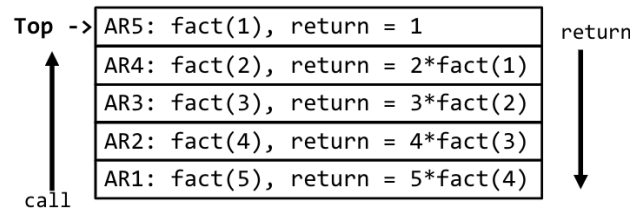
- Programatically:

```
public static int factorial(int n) {
    return (n <= 1) ? 1 : n * factorial(n - 1);
}
```

## Recursion Basics

- Think of your task in terms of a “base case” and a “recursive case”
- The *base case* is the answer which is directly calculated with no recursion
- The *recursive case* is the answer to be calculated by repeated application
  - It is crucial that repeated application of the definition make progress towards the base case

## Recursive Function Execution



- Each recursive call is distinct
  - There is a separate frame or activation record on the runtime stack for each call
  - It's as if they're separate functions with the same implementation

## Common Issues with Recursion

- The base case is unreachable
  - If the recursive definition never triggers the base case, it will never stop
  - Don't put the base case after the recursive call – it'll never be reached
- The recursive case doesn't make progress towards the base case
- Space or time efficiency matters

## Example

- Consider the following definition of the Fibonacci numbers

```
public static int fib(int n) {  
    return (n <= 2) ? 1 : fib(n - 1) + fib(n - 2);  
}
```

- For a call to `fib(5)`, how many additional calls are made?
  - Nine calls
  - It's  $O(2^n)$