CS 310: Recursion and Tree Traversals

Chris Kauffman

Week 10-1

Exams Back

Stat	R2017	R2016
Count	39	
Average	78.97	75.86
Median	80.00	78.33
Max	98.33	96.67
Stdev	11.77	

Range	Count
90 - 100	7
80 - 89	16
70 - 79	6
60 - 69	8
50 - 59	2

Logistics

HW 2

- Milestones Due Tonight
- Final due 1 week
- Discuss multiple ordered lists
- Questions

Reading

- Ch 6.7-9 Maps & Sets
- Ch. 7 Recursion
- Ch 18 General Trees
- Ch 19 BSTs

Today

- Midterm Exams Back
- Finish Maps/Sets
- Tree Traversals
- Recursive traversals
- Recursion practice for tree properties

Ordering

List property

There is a well defined ordering of first, next, last objects in the data structure,

- Wide ranging uses
- Supported in List data structure (`LinkedList`, `ArrayList`)
- Supported structurally in Lists
- *A property of the Data Structure*

Sorting property

There is a well defined ordering relation over all possible data of a type

- "bigger than" "less than" "equal to" are well defined
- A property of the *Data*
- A data structure can try to mirror the data ordering structurally
- Useful for searching, walking through stored data in order

Sorted Lists

Definition is straight-forward

- "Smallest" things are structurally "first", "Biggest" last
- Ordering on elements (`Comparable/Comparator`)
- `add/insert` put elements in proper place

Question: For a sorted `List L`, what is the complexity of `L.insert(x)` which preserves sorting?

`L` is an `ArrayList`

How long to

- find insertion location?
- complete insertion?
- traverse elements in order (e.g. for printing)?

`L` is a `LinkedList`

How long to

- find insertion location?
- complete insertion?
- traverse elements in order (e.g. for printing)?

Alternatives to the Linear Data Structures

Hash Tables

- Abandon list property
- Abandon sorting property
- $O(1)$ insertion/retrieval
- $O(N)$ traversal, not ordered

Trees

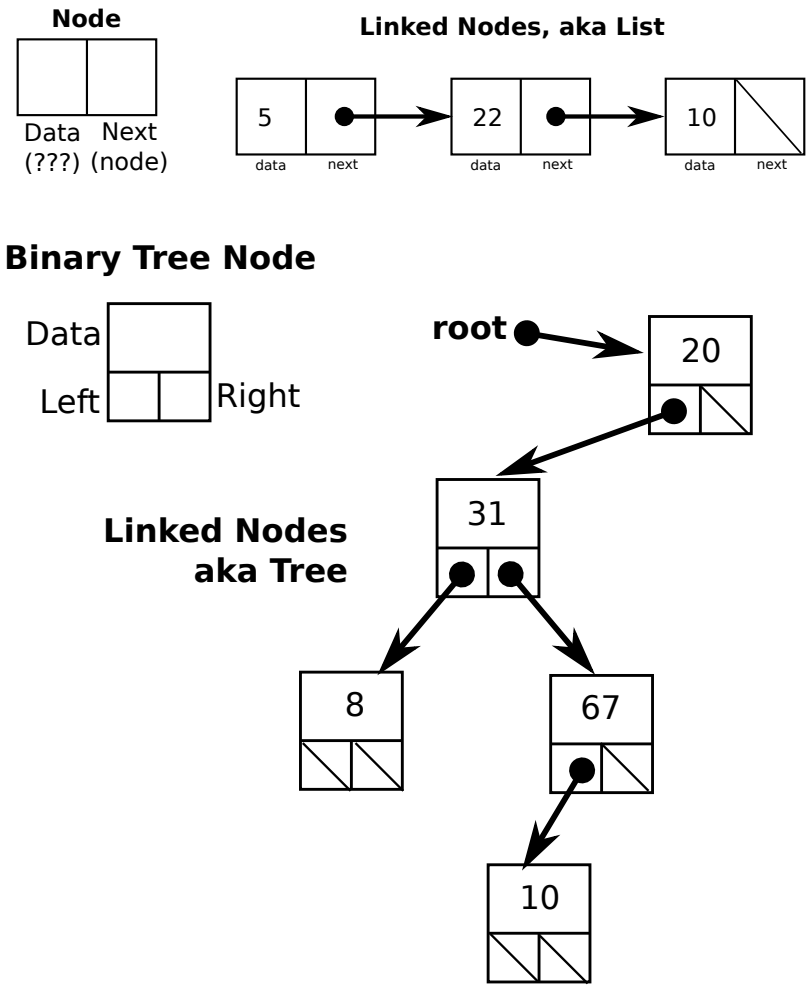
- Abandon list property
- Preserve sorting property
- $O(\log N)$ insertion/retrieval
- $O(N)$ traversal, ordered
- Commonly Binary Trees
- Other variants

Roots

Source

- Next few sessions we'll talk about roots
- For simplicity, we'll call them trees

Mutated Nodes



`Node` structures should be familiar for linked lists

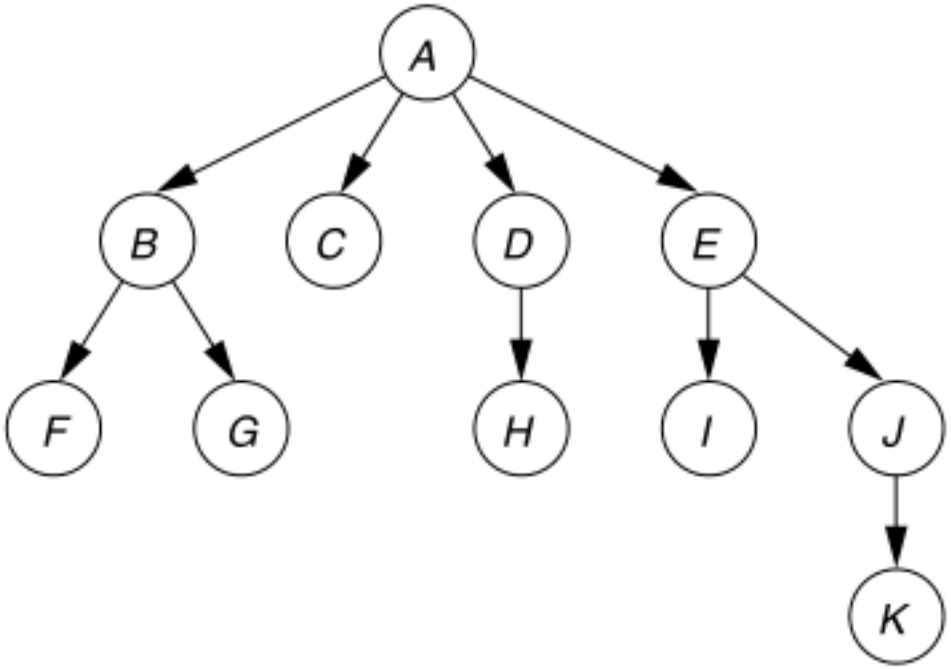
- Singly linked: `next/data`
- Doubly linked: `next/previous/data`

Trees use `Nodes` as well

- `children`, `data`, possibly `parent`
- Arbitrary Trees: `List<Node>` of `children`
- Binary Trees: `left` and `right` children

Tree Properties of Interest

- Root of tree
- Leaves
- Data at nodes
- Size (number of nodes)
- Height of tree
- Depth of a node



Node	Height	Depth
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3

An Apropos Quote

You spend years writing code without recursion and then one day you have to write functions that operate on trees and realize recursion is amazing.
–Kevin DeRonne

Recursion Warm-Up

Write two versions of Singly Linked List `length()` function.

`int length(Node n)`

1. Iterative
2. Recursive

Compare and contrast runtime and memory complexity

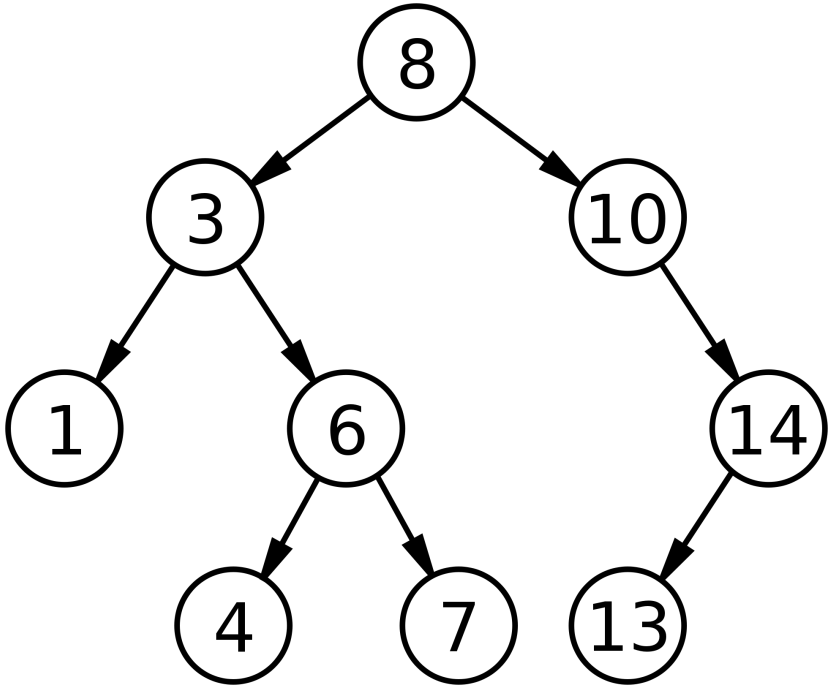
```
class Node<T> {
  T data; Node<T> next;
  public Node(T d, Node<T> n){
    this.data=d; this.next=n;
  }
}

// Singly linked
// No header/auxiliary/dummy nodes
class SimpleList<T>{
  Node<T> head;  // null When empty
  public int length(){
    return length(this.head);
  }
  public static <T>
  int length(Node<T> n){
    // Iterative version?
    // Recursive version?
  }
}
```

Binary Tree

Binary Tree Nodes

```
class Node<T>{
  T data;
  Node<T> left, right;
}
void main(){
  Node root = new Node();
  root.data = 8;
  root.left = new Node();
  root.right= new Node();
  root.left.data = 3;
  root.right.data= 10;
  root.left.left = new Node();
  ...
```

Structure



Recursive Example: Binary Tree Size Method

Tree Nodes

```
class Node<T>{
  T data;
  Node<T> left, right;
}
```

Usage

```
Tree<Integer> myTree = new Tree();
// add some stuff to myTree
int s = myTree.size();
```

Exercise

- Define a recursive `t.height()`
- `t.height()` is the longest path from root to leaf
- Empty tree has height=0

`int size(Node<T> t)`

Number of nodes in tree `t`

```
public Tree<T>{
  Node<T> root;

  // Entry point
  public int size(){
    return size(this.root);
  }
  // Recursive helper
  public static <T>
    int size( Node<T> t ){
    if(t == null){
      return 0;
    }
    int sL = size(t.left);
    int sR = size(t.right);
    return 1 + sL + sR;
  }
}
```

Recursive Implementation of `height()`

Slight difference of definitions from textbook

- Empty tree has `size=0` and `height=0`
- 1-node tree has `size=1` and `height=1`

Show the function call stack as the computation of `size()` and `height()` methods proceeds.

```
// Depth of deepest node
public Tree<T>{
  Node<T> root;
  public int height(){
    return height(this.root);
  }

  public static <T>
    int height( Node<T> t ){
    if(t == null){
      return 0;
    }
    int hL = height(t.left);
    int hR = height(t.right);
    int bigger = Math.max(hL,hR);
    return 1+bigger;
  }
}
```

Recall Tree Basics

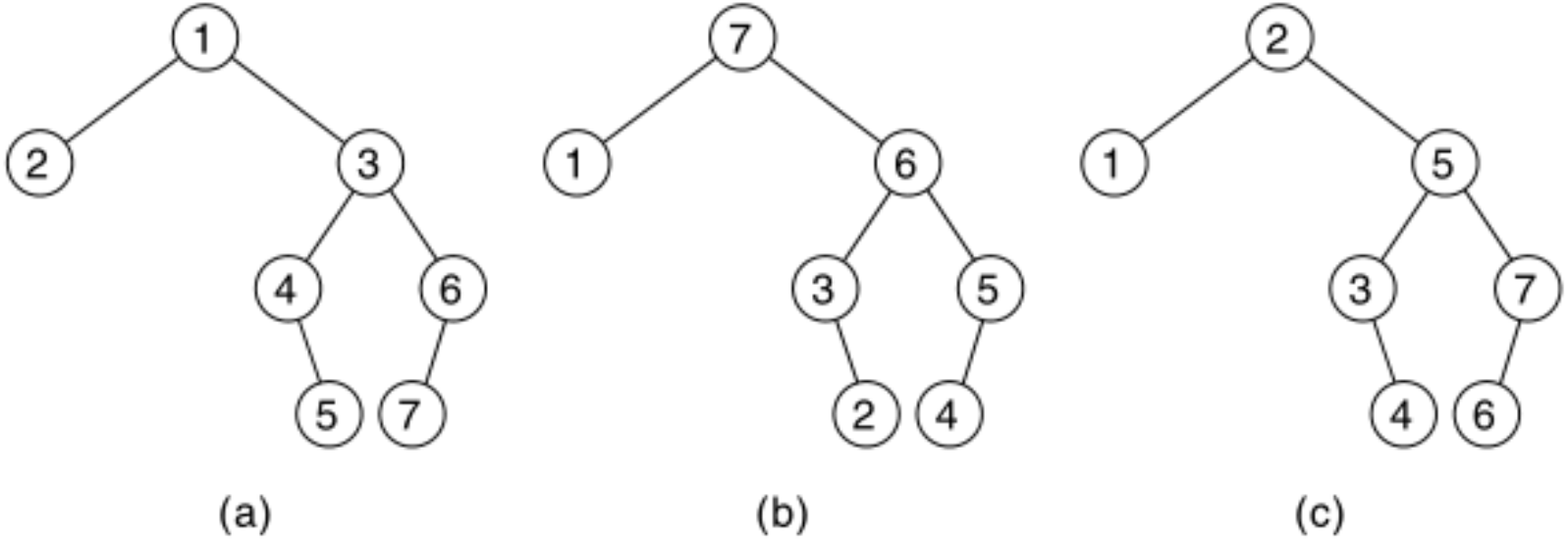
- What distinguishes a tree from a linked list? What gets stored at each tree `Node`
- What technique becomes useful for implementing operations on trees? Why?
- Why are many of the tree methods we write broken down as follows:

```
public class Tree<T> {
  ...
  public type method(){
    return method(this.root);
  }

  private type method(Node<T> n){
    ...
  }
}
```

The Many Ways to Walk

No list property: several orders to traverse tree

- (a) Pre-order traversal (parent, left, right)
- (b) Post-order traversal (left, right, parent)
- (c) In-order traversal (left, parent, right)



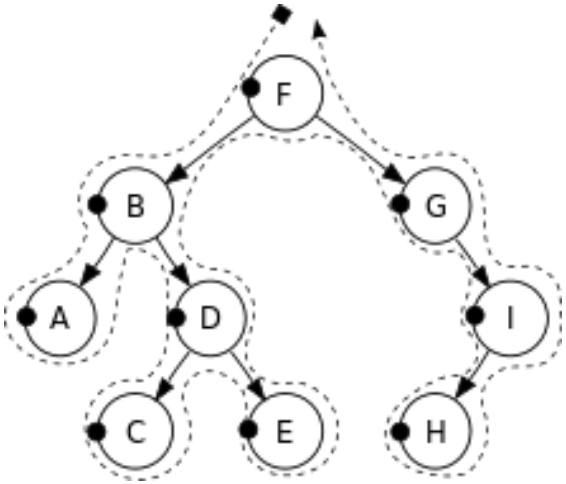
Picture shows the order nodes will be visited in each type of traversal

The Many Ways to Walk

No list property: several orders to traverse tree

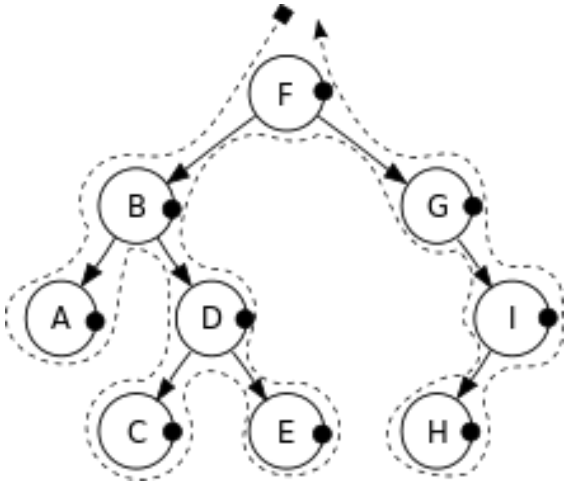
Pre-order traversal

parent, left, right



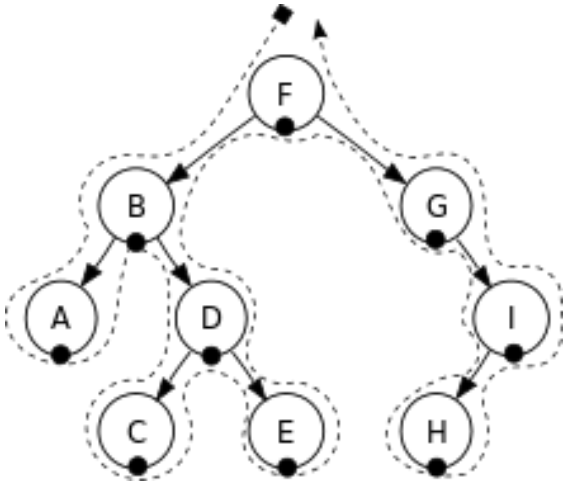
Post-order traversal

left, right, parent

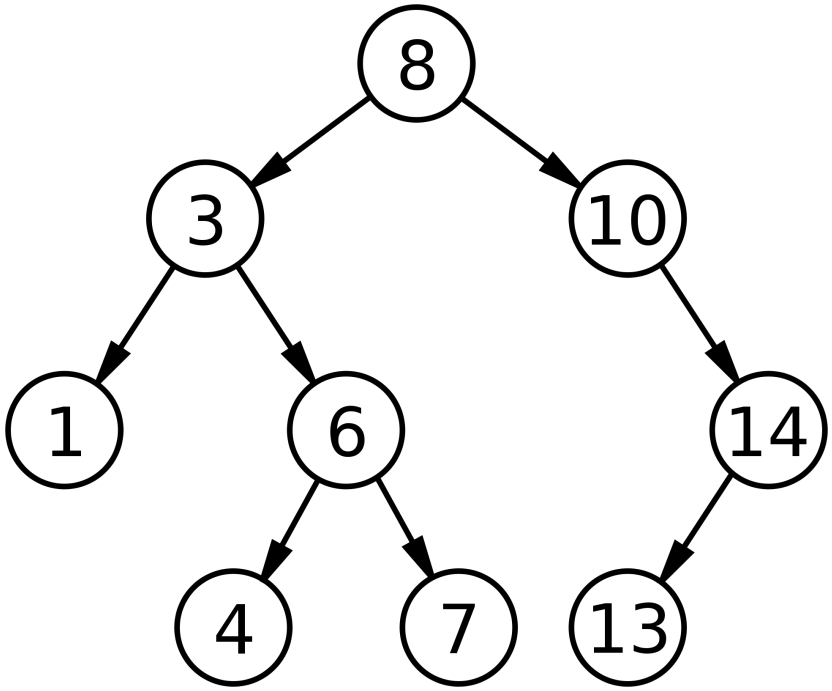


In-order traversal

left, parent, right



Walk This Tree



Show

- (a) Pre-order traversal (parent, left, right)
- (b) Post-order traversal (left, right, parent)
- (c) In-order traversal (left, parent, right)

Which one "sorts" the numbers?

Implementing Traversals for Binary Trees

```
class Tree<T>{
  private Node<T> root;

  public void printPreOrder(){
    preOrder(this.root);
  }
  private static void
  preOrder(Node<T> t){
    ... print(t.data) ...
  }

  public void printInOrder(){ }
  private static void
  inOrder(Node<T> t){ }

  public void printPostOrder(){ }
  private static void
  postOrder(Node<T> t){ }
}
```

```
class Node<T> {
  T data;
  Node<T> left, right;
}
```

Implement Print Traversals

- `preOrder(this.root)`
- `postOrder(this.root)`
- `inOrder(this.root)`

2 Ways

- Recursively (first)
- Iteratively (good luck...)

Recursive Implementation of Traversals

```
inOrder(Node t){
  if(t != null){
    inOrder(t.left);
    print(t.data);
    inOrder(t.right);
  }
}
```

```
preOrder(Node t){
  if(t != null){
    print(t.data);
    preOrder(t.left);
    preOrder(t.right);
  }
}
```

```
postOrder(Node t){
  if(t != null){
    postOrder(t.left);
    postOrder(t.right);
    print(t.data);
  }
}
```

Evaluate

- Correct?
- Time complexity?
- Space complexity?
- What makes this so easy?

Distribution Code

Today's code distribution contains demos of recursive methods

`SimpleList.java`

Demos recursive version of list length

`Tree.java`

Contains a very simple tree example that demos

- `size()`
- `height()`
- Traversals: Pre-order, In-order, Post-order

Iterative Implementation?

Compare to Iterative Implementation of Traversals

```
// Pseudo-code for post order print
void postOrder(root){
  Stack s = new Stack();
  s.push( {root, DOLEFT });
  while(!s.empty()){
    {tree, action} = s.popTop();
    if(tree == null){
      // do nothing;
    }
    else if(action == DOLEFT){
      s.push({tree, DORIGHT});
      s.push({tree.left, DOLEFT});
    }
    else if(action == DORIGHT){
      s.push({tree, DOTHIS});
      s.push({tree.right, DOLEFT});
    }
    else if(action == DOTHIS){
      print(tree.data);
    }
    else{
      throw new YouScrewedUpException();
    }
  }
}
```

- No call stack
- Use an explicit stack
- Auxilliary data `action`

`DOLEFT`	work on left subtree
`DORIGHT`	work on right subtree
`DOTHIS`	process data for current

Evaluate

- Correct?
- Time complexity?
- Space complexity?

Weiss's Traversals

Implemented as `iterators`

- See `TestTreeIterators.java`
- Uses `BinaryTree.java` and `BinaryNode.java`
- Must preserve state accross `advance()` calls

```
BinaryTree<Integer> t = new BinaryTree<Integer>( );
... // fill tree

TreeIterator<AnyType> itr = new PreOrder<Integer>( t );
for( itr.first( ); itr.isValid( ); itr.advance( ) ){
  System.out.print( " " + itr.retrieve( ) );
}
```

- Much more complex to understand but good for you
- Play with some of these in a debugger if you want more practice

General Notes

Iterative Traversal Implementation Notes

- Can augment tree nodes to have a `parent` pointer

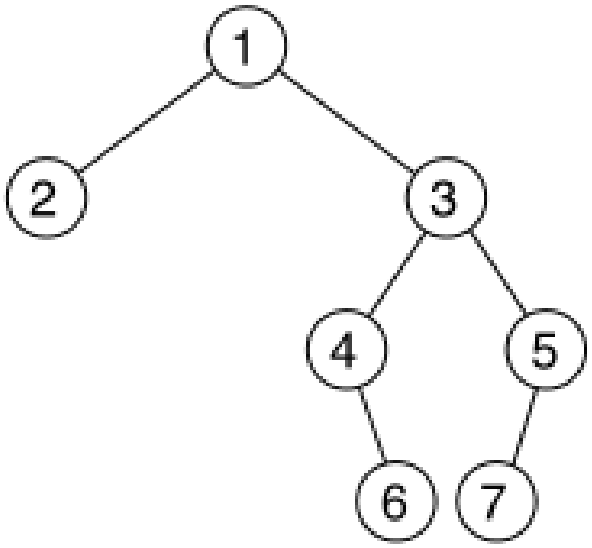
```
class Node<T>{
  T data; Node left, right, parent;
}
```

- Enables stackless, iterative traversals with great cleverness

Iterative vs Recursive Tree Methods

- Multiple types of traversals of `T`
- Other Tree methods: `T.find(x)`, `T.add(x)`, `T.remove(x)`
- Recursive implementations are simpler to code but will cost more memory
- Iterative methods are possible and save memory at the expense of tricky code

Level-order Traversal



Level Order Traversal: `1 2 3 4 5 6 7`

- Top level first (depth 1: `1`)
- Then next level (depth 2: `2 3`)
- etc.

This is a bit trickier

- Need an auxilliary data structure: Queue
- Does recursion help?