CS 600.226: Data Structures

Michael Schatz

Oct 1 2018

Lecture 14. Trees

Agenda

1. ***Review HW3***
2. ***Questions on HW4***
3. ***Recap on Lists***
4. ***Trees***

Assignment 3: Due Friday Sept 28 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment03/README.md

Assignment 3: Assorted Complexities

Out on: September 21, 2018
Due by: September 28, 2018 before 10:00 pm
Collaboration: None
Grading:

Functionality 60% (where applicable)
Solution Design and README 10% (where applicable)
Style 10% (where applicable)
Testing 10% (where applicable)

Overview
The third assignment is mostly about sorting and how fast things go. You will also write yet another implementation of the Array interface to help you analyze how many array operations various sorting algorithms perform.

Note: The grading criteria now include 10% for unit testing. This refers to JUnit 4 test drivers, not some custom test program you hacked. The problems (on this and future assignments) will state whether you are expected to produce/improve test drivers or not.

Agenda

1. ***Review HW3***
2. ***Questions on HW4***
3. ***Recap on Lists***
4. ***Trees***

Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

Assignment 4: Stacking Queues

Out on: September 28, 2018
Due by: October 5, 2018 before 10:00 pm
Collaboration: None
Grading:

Packaging 10%,
Style 10% (where applicable),
Testing 10% (where applicable),
Performance 10% (where applicable),
Functionality 60% (where applicable)

Overview
The fourth assignment is mostly about stacks and dequeues. For the former you'll build a simple calculator application, for the latter you'll implement the data structure in a way that satisfies certain performance characteristics (in addition to the usual correctness properties).

Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

Problem 1: Calculating Stacks (50%)

Your first task is to implement a basic RPN calculator that supports integer operands like 1, 64738, and -42 as well as the (binary) integer operators +, -, *, /, and %. Your program should be called `Calc` and work as follows:

- You create an empty `Stack` to hold intermediate results and then repeatedly accept input from the user. It doesn't matter whether you use the `ArrayStack` or the `ListStack` we provide, what does matter is that those specific types appear only once in your program.
- If the user enters a ***valid integer***, you ***push*** that integer onto the stack.
- If the user enters a ***valid operator***, you ***pop*** two integers off the stack, ***perform*** the requested operation, and ***push*** the result back onto the stack.
- If the user enters the symbol `?` (that's a question mark), you ***print*** the current state of the stack using its `toString` method followed by a new line.
- If the user enters the symbol `.` (that's a dot or full-stop), you ***pop*** the top element off the stack and ***print*** it (only the top element, not the entire stack) followed by a new line.
- If the user enters the symbol `!` (that's an exclamation mark or bang), you exit the program.

Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

```
$ java Calc
?
[]
10
?
[10]
20 30
?
[30, 20, 10]
*
?
[600, 10]
+
?
[610]
.
610
!
$
```

```
$ java Calc
? 10 ? 20 30 ? *
? + ? . !
[]
[10]
[30, 20, 10]
[600, 10]
[610]
610
$
```

Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

Problem 2: Hacking Growable Dequeues (50%)

Your second task is to implement a generic `ArrayDequeue` class as outlined in lecture. As is to be expected, `ArrayDequeue` must implement the `Dequeue` interface we provided on github.

- Your implementation must be done in terms of an array that grows by doubling as needed. It's up to you whether you want to use a basic Java array or the SimpleArray class you know and love; just in case you prefer the latter, we've once again included it on the github directory for this assignment. Your initial array must have a length of one slot only! (Trust us, that's going to make debugging the "doubling" part a lot easier.)
- Your implementation must support all `Dequeue` operations except insertion in (worst-case) constant time; insertion can take longer every now and then (when you need to grow the array), but overall all insertion operations must be constant amortized time as discussed in lecture.
- You should provide a `toString` method in addition to the methods required by the `Dequeue` interface. A new dequeue into which 1, 2, and 3 were inserted using `insertBack()` should print as [1, 2, 3] while an empty dequeue should print as []

Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

Bonus Problem (5 pts)

Develop an **algebraic specification** for the **abstract data type Queue**. Use `new`, `empty`, `enqueue`, `dequeue`, and `front` (with the meaning of each as discussed in lecture) as your set of operations. Consider unbounded queues only.

The difficulty is going to be modelling the FIFO (first-in-first-out) behavior accurately. You'll probably need at least one axiom with a case distinction using an if expression; the syntax for this in the Array specification for example.

Doing this problem without resorting to Google may be rather helpful for the upcoming midterm. There's no need to submit the problem, but you can submit it if you wish; just include it at the end of your README file.

Agenda

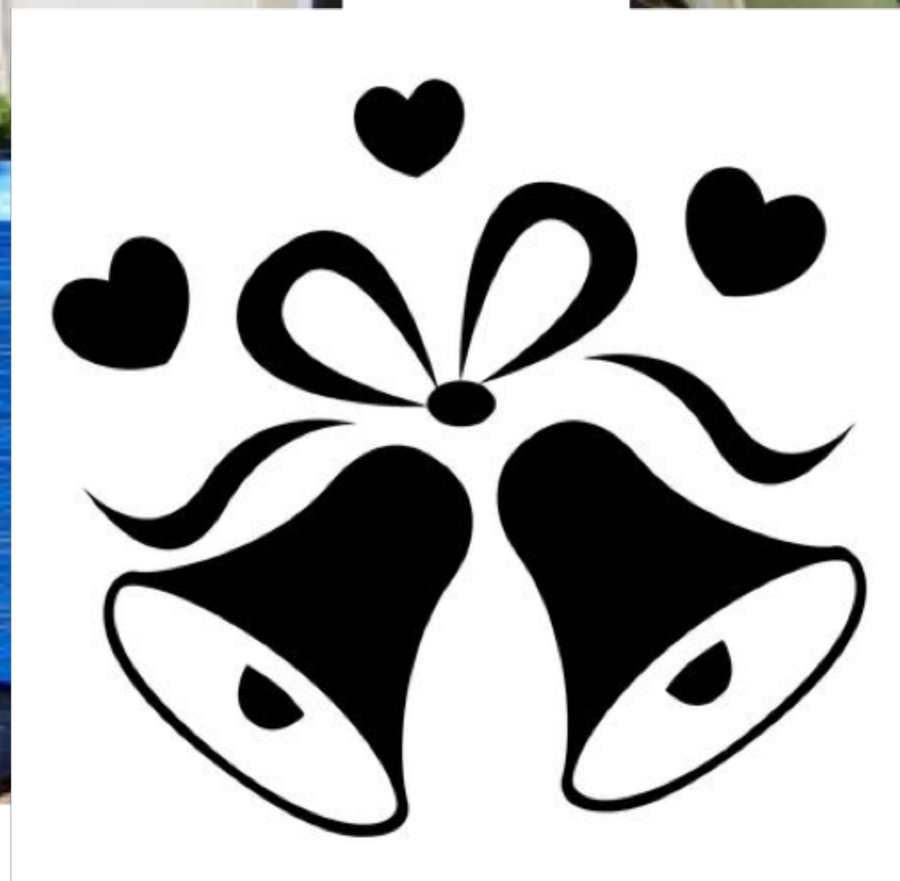
1. ***Review HW3***
2. ***Questions on HW4***
3. ***Recap on Lists***
4. ***Trees***

Stacks versus Queues

LIFO: Last-In-First-Out
Add to top +
Remove from top

FIFO: First-In-First-Out
Add to back +
Remove from front

Stacks versus Queues



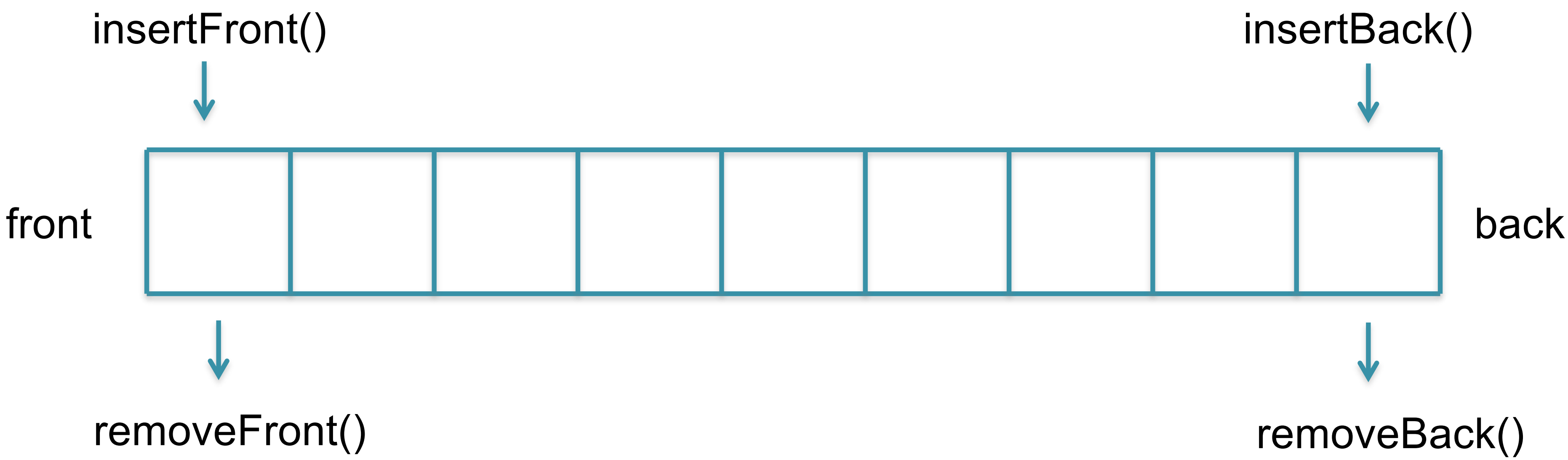
LIFO: Last-

Add to top +
Remove from top

st-In-First-Out

Add to back +
Remove from front

Dequeues

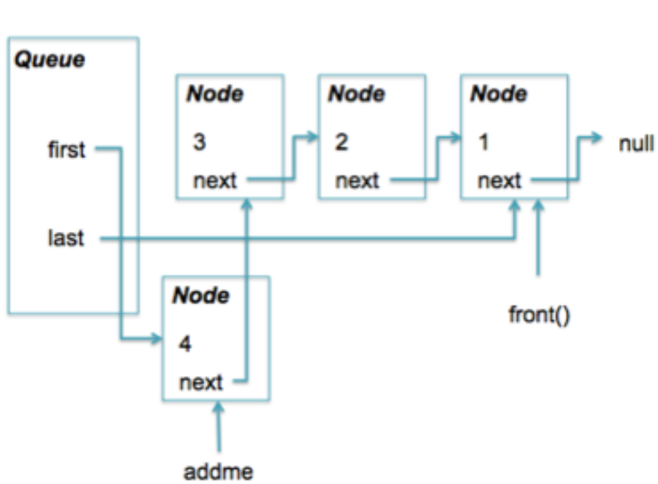


Dynamic Data Structure used for storing sequences of data

- Insert/Remove at either end in O(1)
- If you exclusively add/remove at one end, then ***it becomes a stack***
- If you exclusive add to one end and remove from other, then ***it becomes a queue***
- Many other applications:
  - browser history: deque of last 100 webpages visited

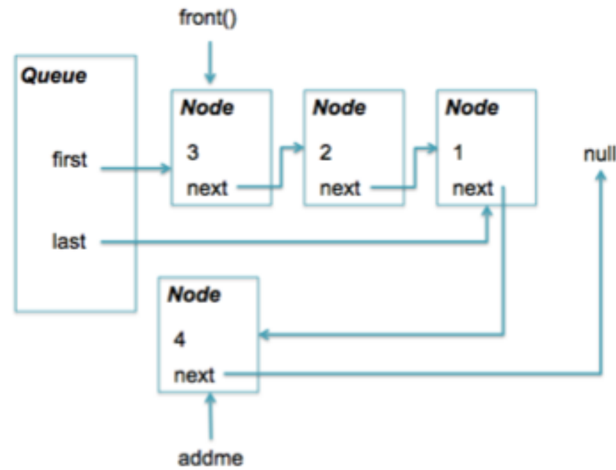
Singly Linked Lists

insertFront



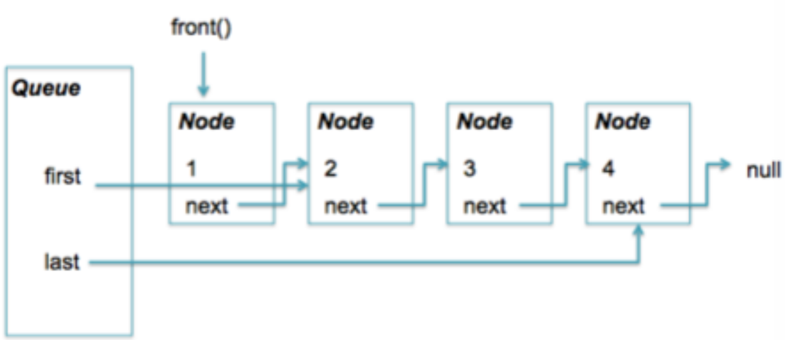
addme.next = first; first = addme;

insertBack



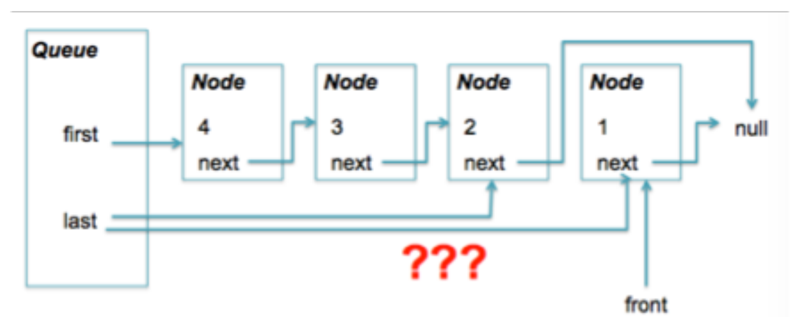
last.next = addme; addme.next = null

removeFront



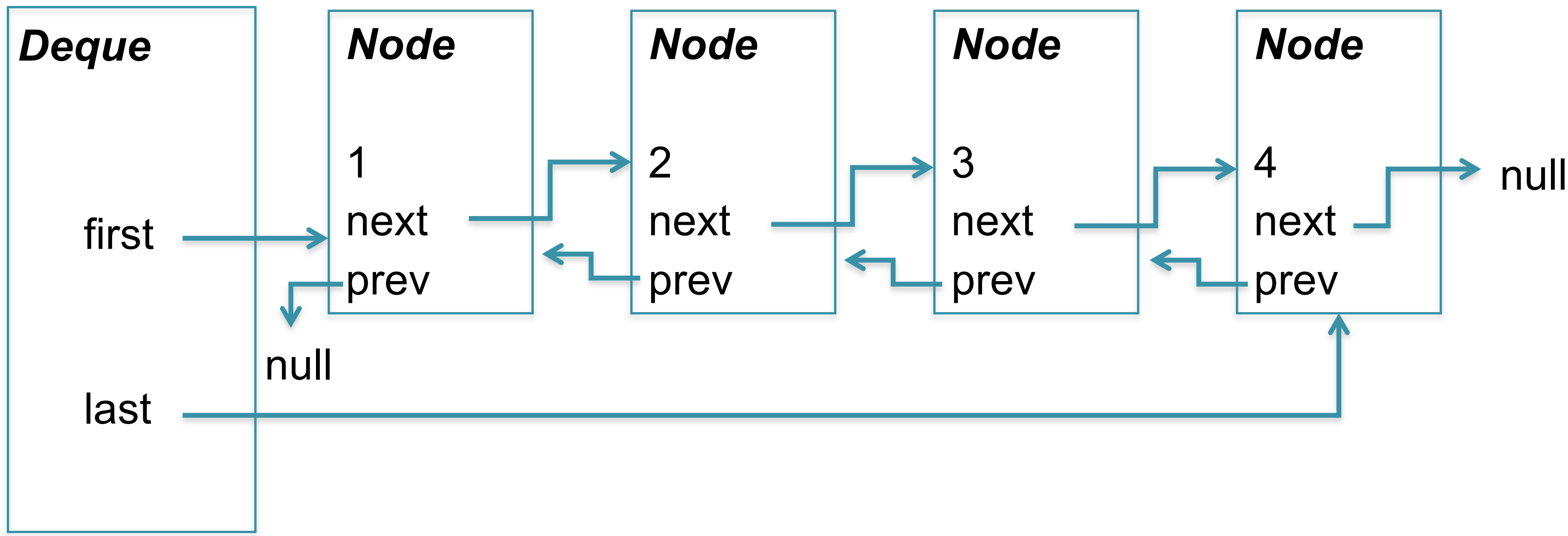
first = first.next;

removeBack



???

Doubly Linked List



Very similar to a singly linked list, except each node has a reference to both the next and previous node in the list

A little more overhead, but significantly increased flexibility: supports insertFront(), insertBack(), removeFront(), removeBack(), insertBefore(), removeMiddle()

List v4

```
public interface Node<T> {
    void setValue(T t);
    T getValue();

    void setNext(Node<T> n);
    void setPrev(Node<T> n);

    void getNext(Node<T> n);
    void getPrev(Node<T> n);
}

public interface List<T> {
    boolean empty();
    int length();

    Node<T> front();
    Node<T> back();

    void insertFront(Node<T> t);
    void insertBack(Node<T> t);

    void removeFront();
    void removeBack();
}
```

List v4

```
public interface Node<T> {
    void setValue(T t);
    T getValue();

    void setNext(Node<T> n);
    void setPrev(Node<T> n);

    void getNext(Node<T> n);
    void getPrev(Node<T> n);
}

public interface List<T> {
    boolean empty();
    int length();

    Node<T> front();
    Node<T> back();

    void insertFront(Node<T> t);
    void insertBack(Node<T> t);

    void removeFront();
    void removeBack();
}
```

```
public interface Position<T> {
    // empty on purpose
}

public interface List<T> {
    // simplified interface
    int length();

    Position<T> insertFront(T t);
    Position<T> insertBack(T t);
    void insertBefore(Position<T> t);
    void insertAfter(Position<T> t);

    void removeAt(Position<T> p);
}
```

List v4

"I am a position and while you can hold on to me, you can't do anything else with me!"

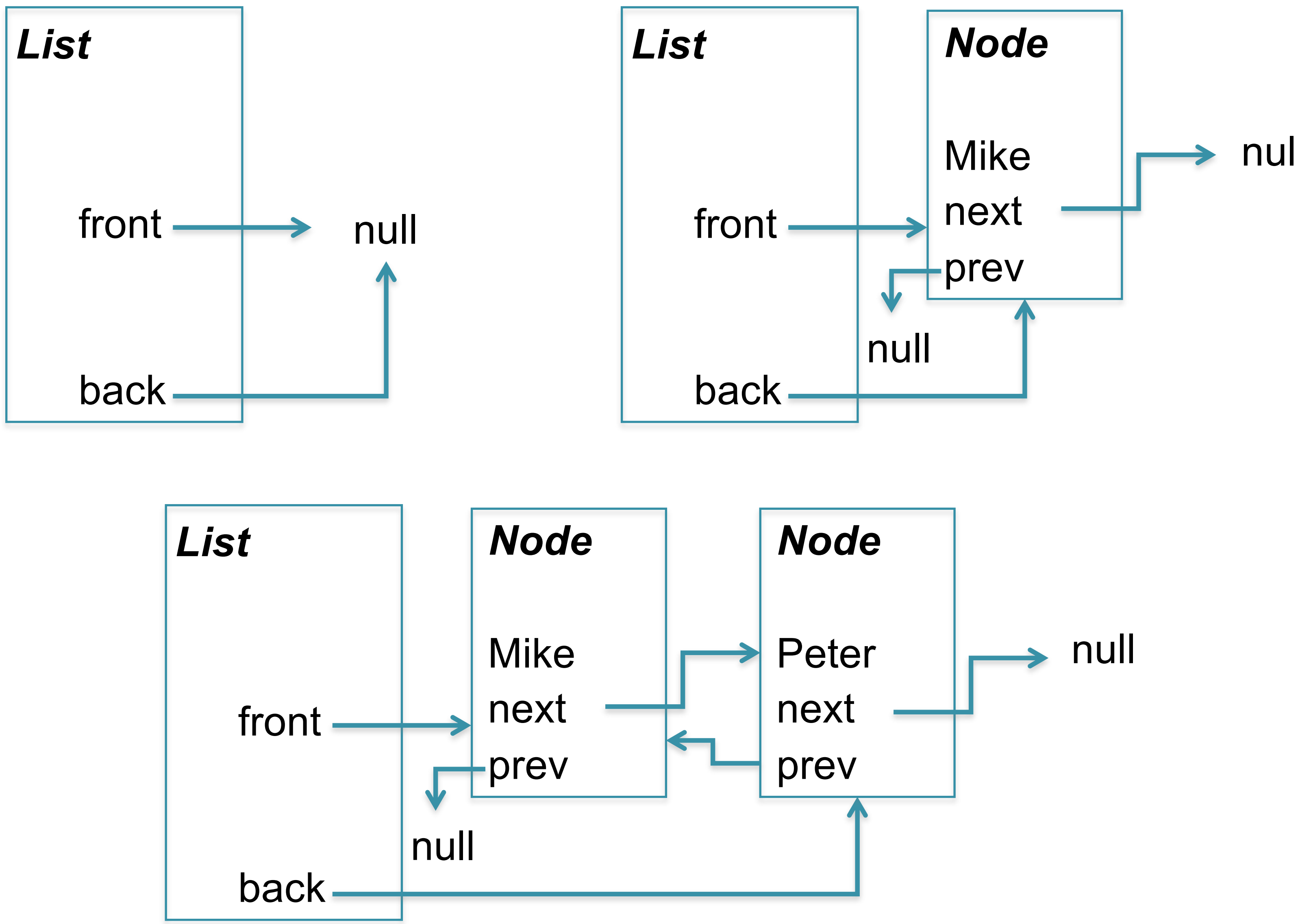
Inserting at front or back creates the Position objects.

If you want, you could keep references to the Position objects even in the middle of the list

Pass in a Position, and it will remove it from the list

```
public interface Position<T> {
    // empty on purpose
}

public interface List<T> {
     // simplified interface
     int length();

     Position<T> insertFront(T t);
     Position<T> insertBack(T t);
     void insertBefore(Position<T> t);
     void insertAfter(Position<T> t);

     void removeAt(Position<T> p);
}
```

Living in a null world



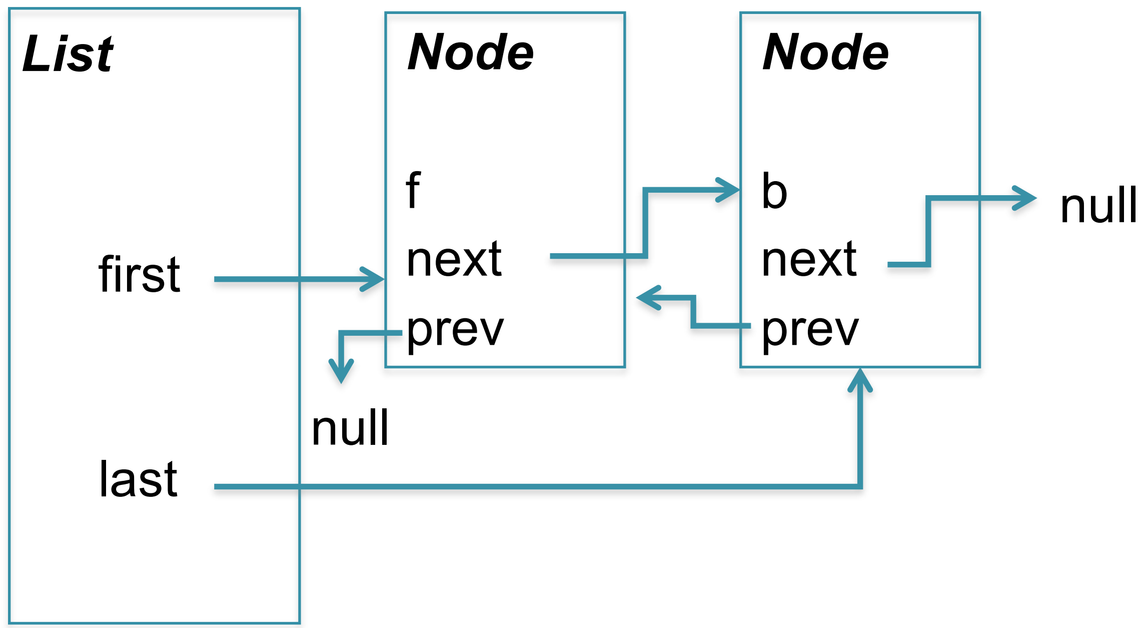
Living in a null world

```
public Position <T> insertBack(T t) {
    ...
    if (this.back != null) {
        this.back.next = n;
    }
    if (this.front == null) {
        this.front = n;
    }
    ...
}
```

```
public Position <T> insertFront(T t ) {
    ...
    if (this.front != null) {
        this.front.prev =n;
    }
    if (this.back==null) {
        this.back = n;
    }
    ...
}
```

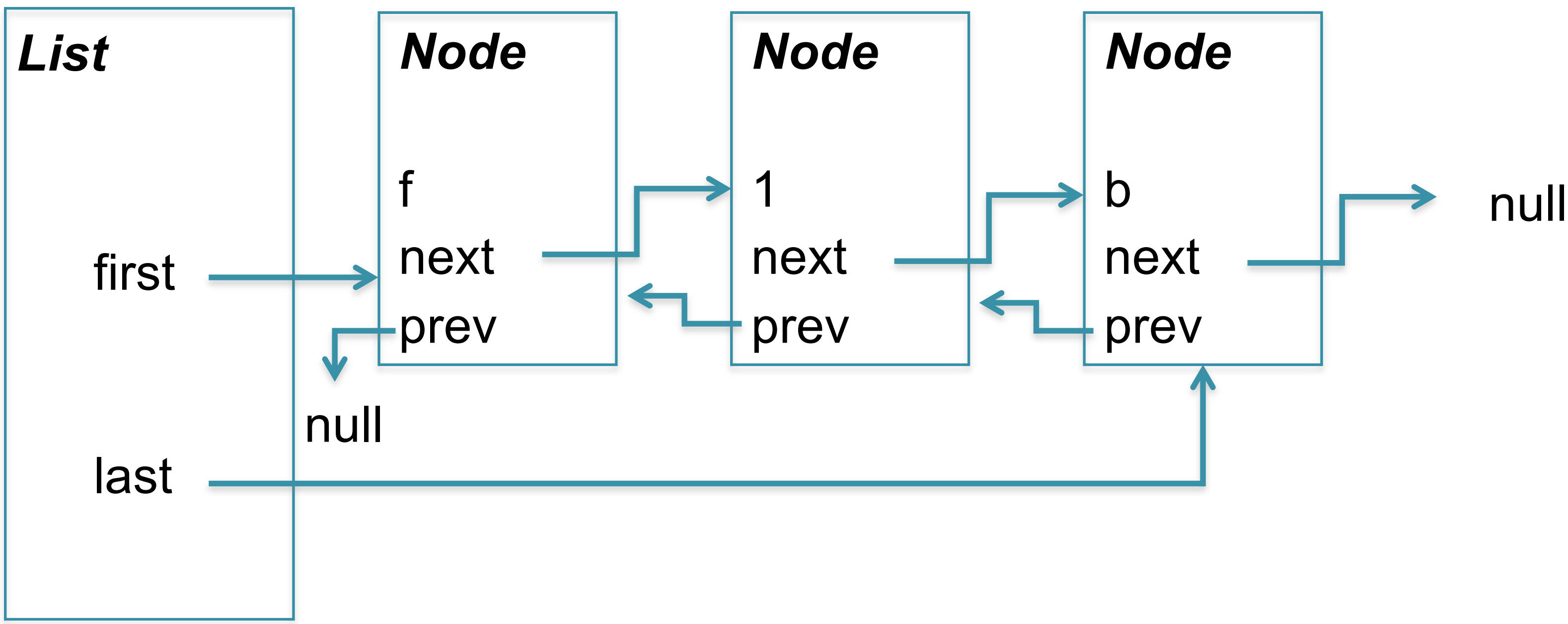
```
public void removeAt(Position<T> p) {
    ...
    if (n.next != null)  { n.next.prev = n.prev; }
    if (n.prev != null)  { n.prev.next = n.next; }
    ...
}
```

Doubly Linked List with Sentinels



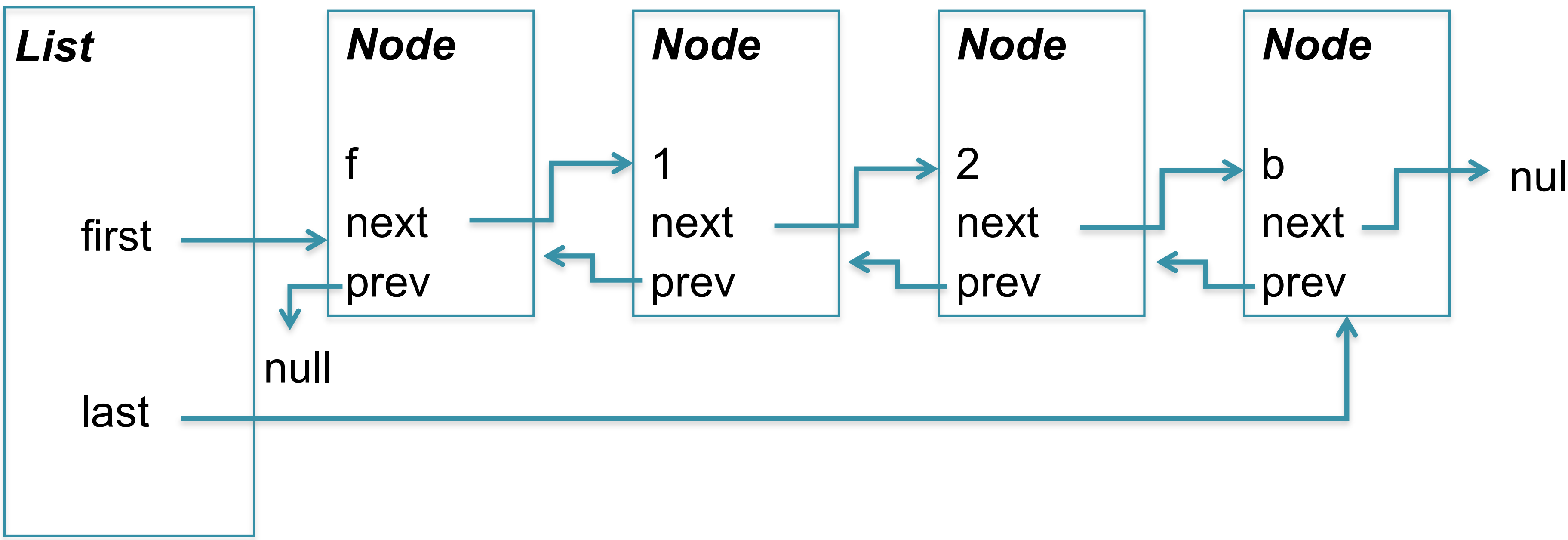
An “empty” list is initialized with the special front and back sentinels

Doubly Linked List with Sentinels



The user data is placed in between sentinels

Doubly Linked List with Sentinels

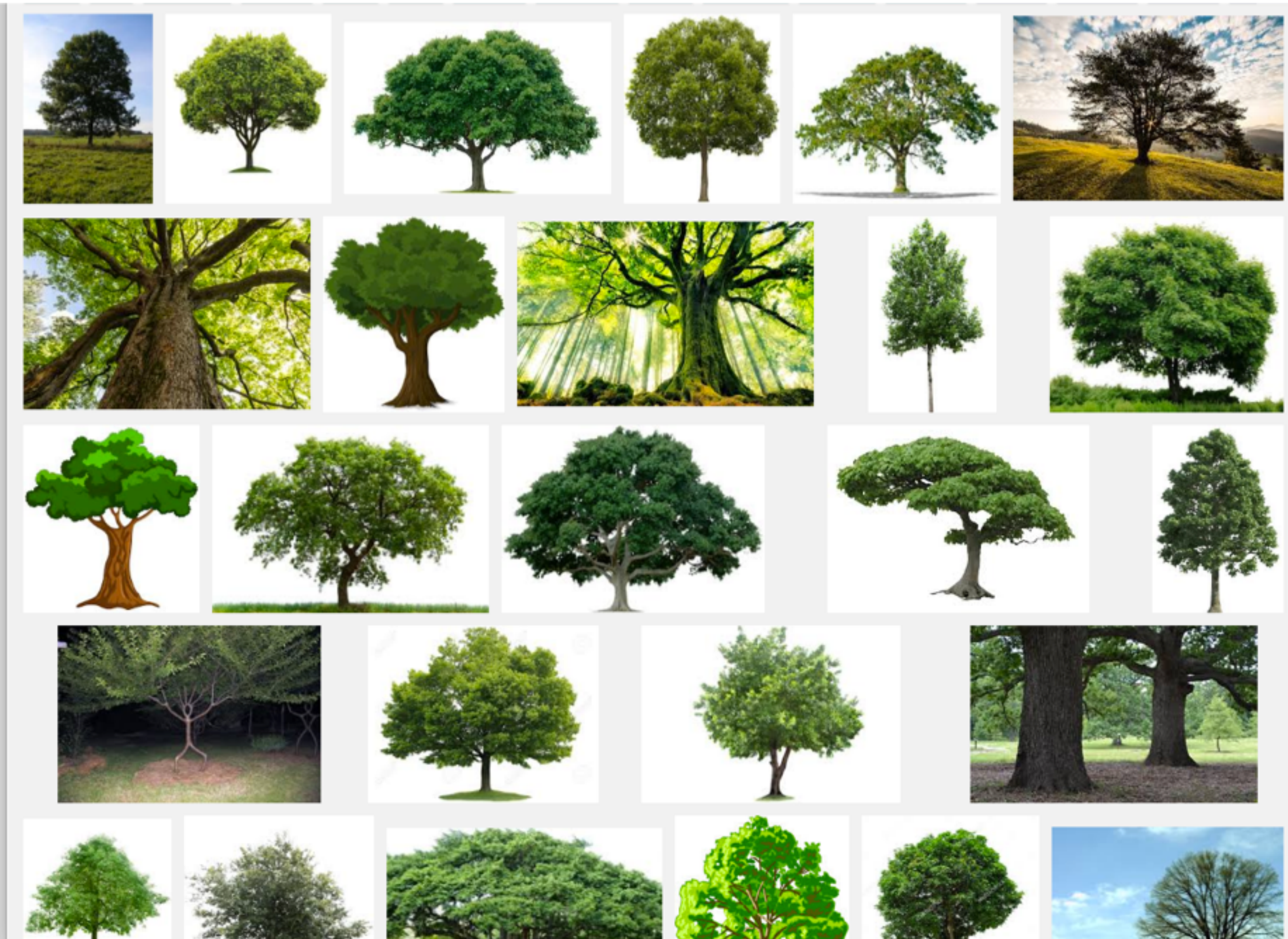


For the cost of a tiny bit of extra memory, the code gets significantly simpler!

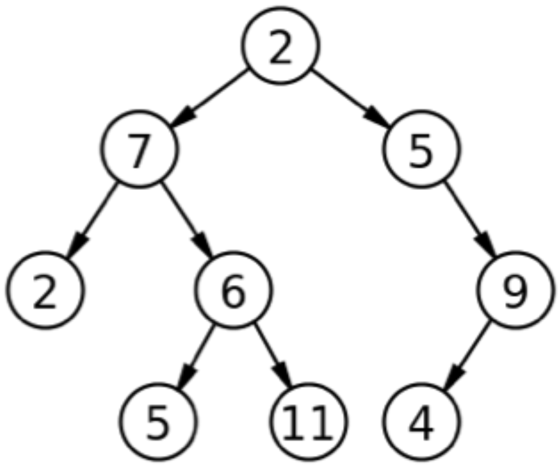
Get ready for HW5 ☺

Trees

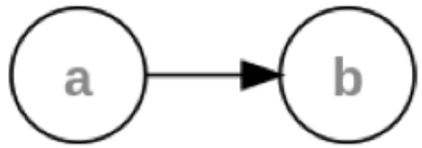
Trees are all around us ☺



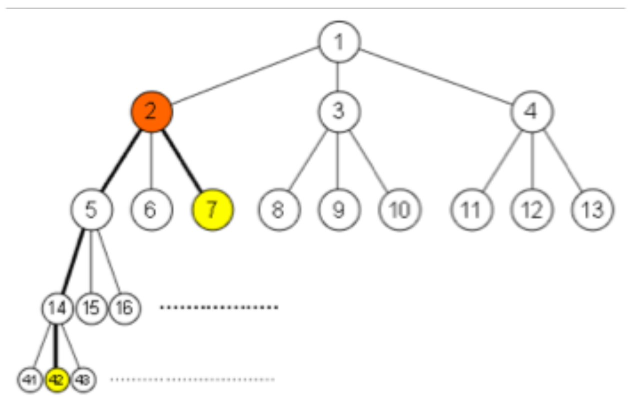
Types of Trees



Unordered
Binary tree



Linear
List

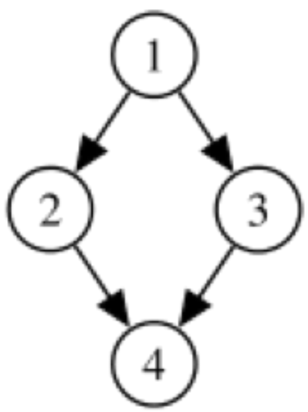


3-ary Tree
(k-ary tree has k children)

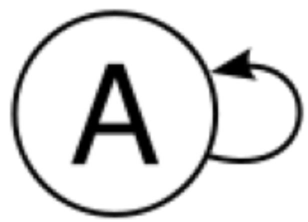
Single root node (no parent)
Each ***non-root*** node has at most 1 parent
Node may have 0 or more children

Internal node: has children; includes root unless tree is just root
Leaf node (aka external node): no children

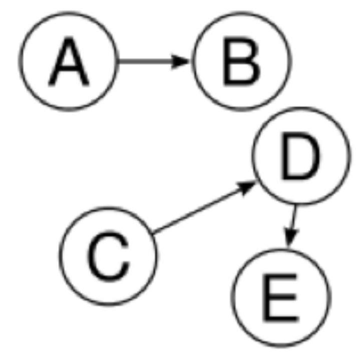
Not Trees



Node 4 has 2 parents



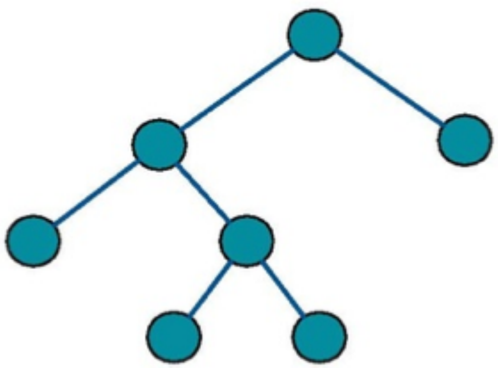
Forms a (self) cycle



2 root nodes: Forest

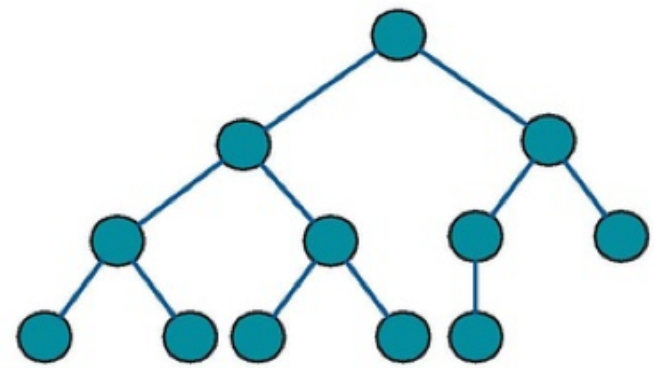
Single root node (no parent)
Each internal node has at most 1 parent
Node may have 0 or more children

Special Trees



Height of root
= 0

Total Height
= 3

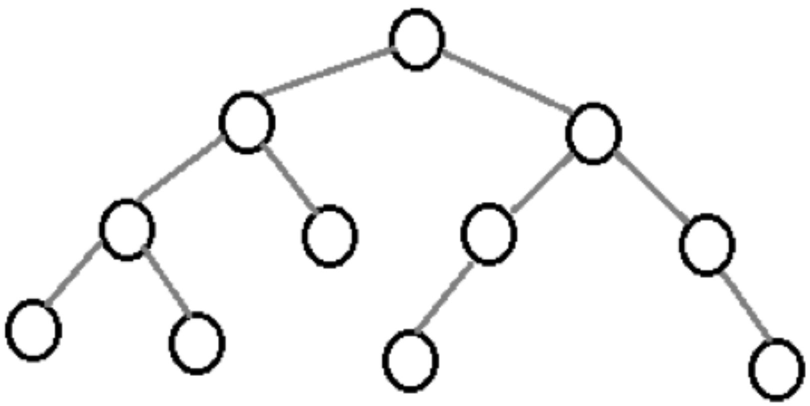


Full Binary Tree
Every node has
0 or 2 children

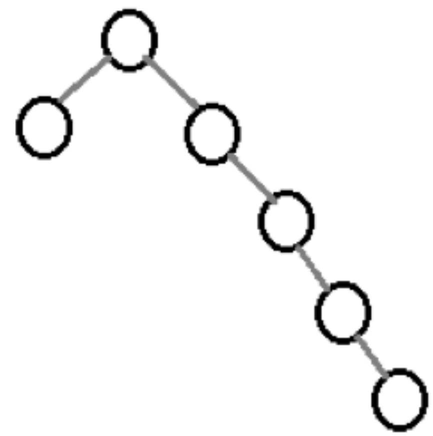
Complete Binary Tree
Every level full, except
potentially the bottom level

What is the maximum number of leaf nodes in a complete binary tree?	2^h
What is the maximum number of nodes in a complete binary tree?	$2^{h+1}-1$
What fraction of the nodes in a complete binary tree are leaves?	about half

Balancing Trees

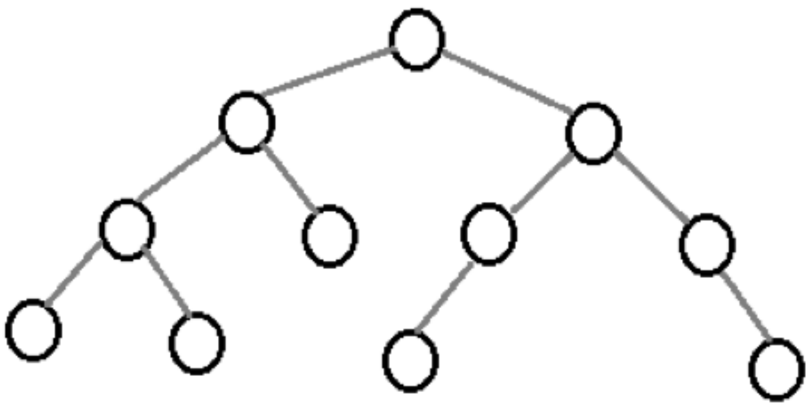


Balanced Binary Tree
Minimum possible height

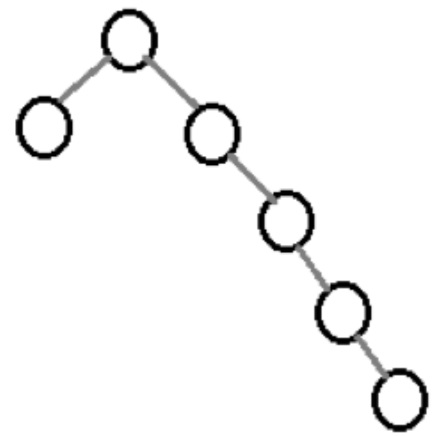


Unbalanced Tree
Non-minimum height

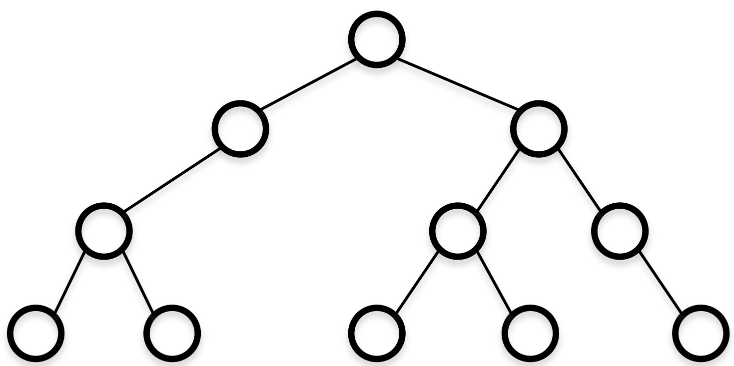
Balancing Trees



Balanced Binary Tree
Minimum possible height

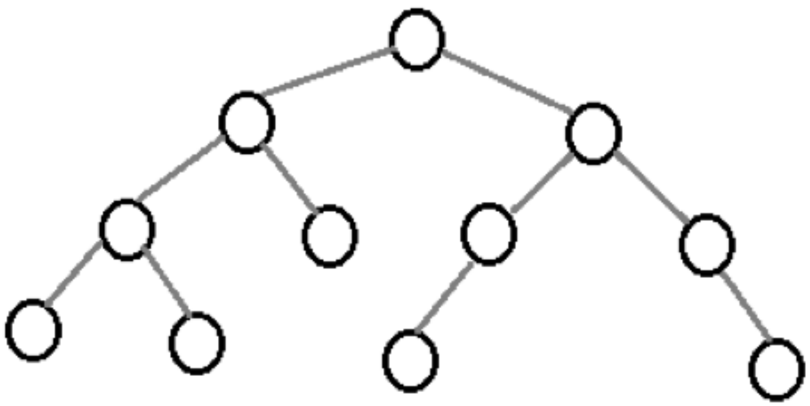


Unbalanced Tree
Non-minimum height

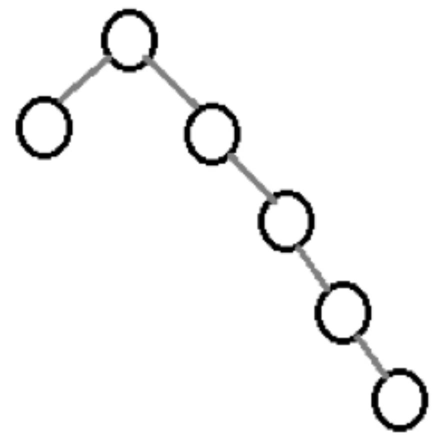


?

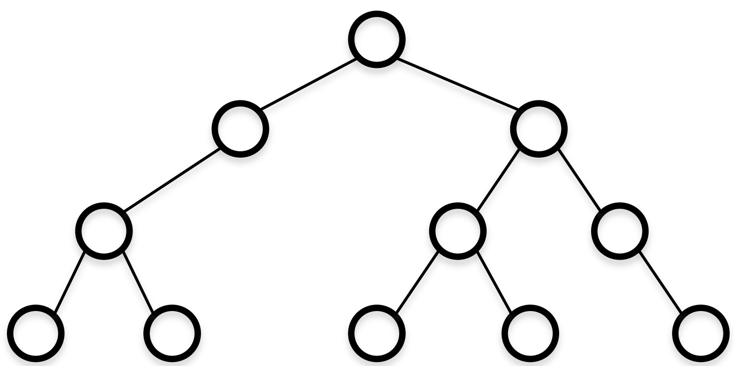
Balancing Trees



Balanced Binary Tree
Minimum possible height

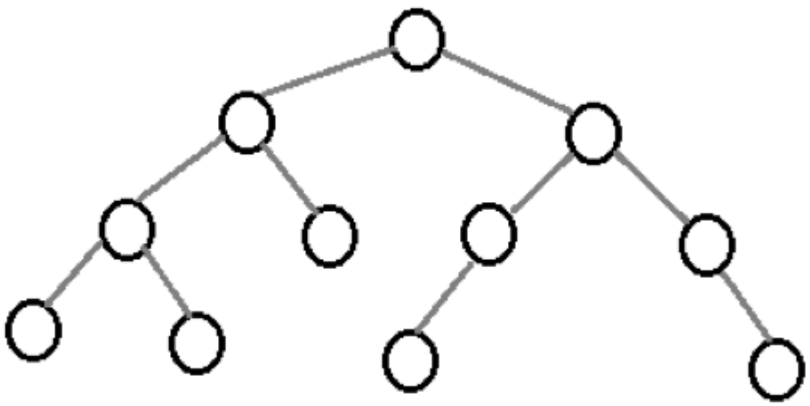


Unbalanced Tree
Non-minimum height

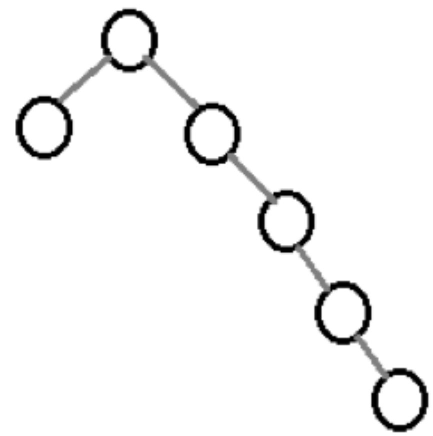


Balanced but not complete!

Balancing Trees



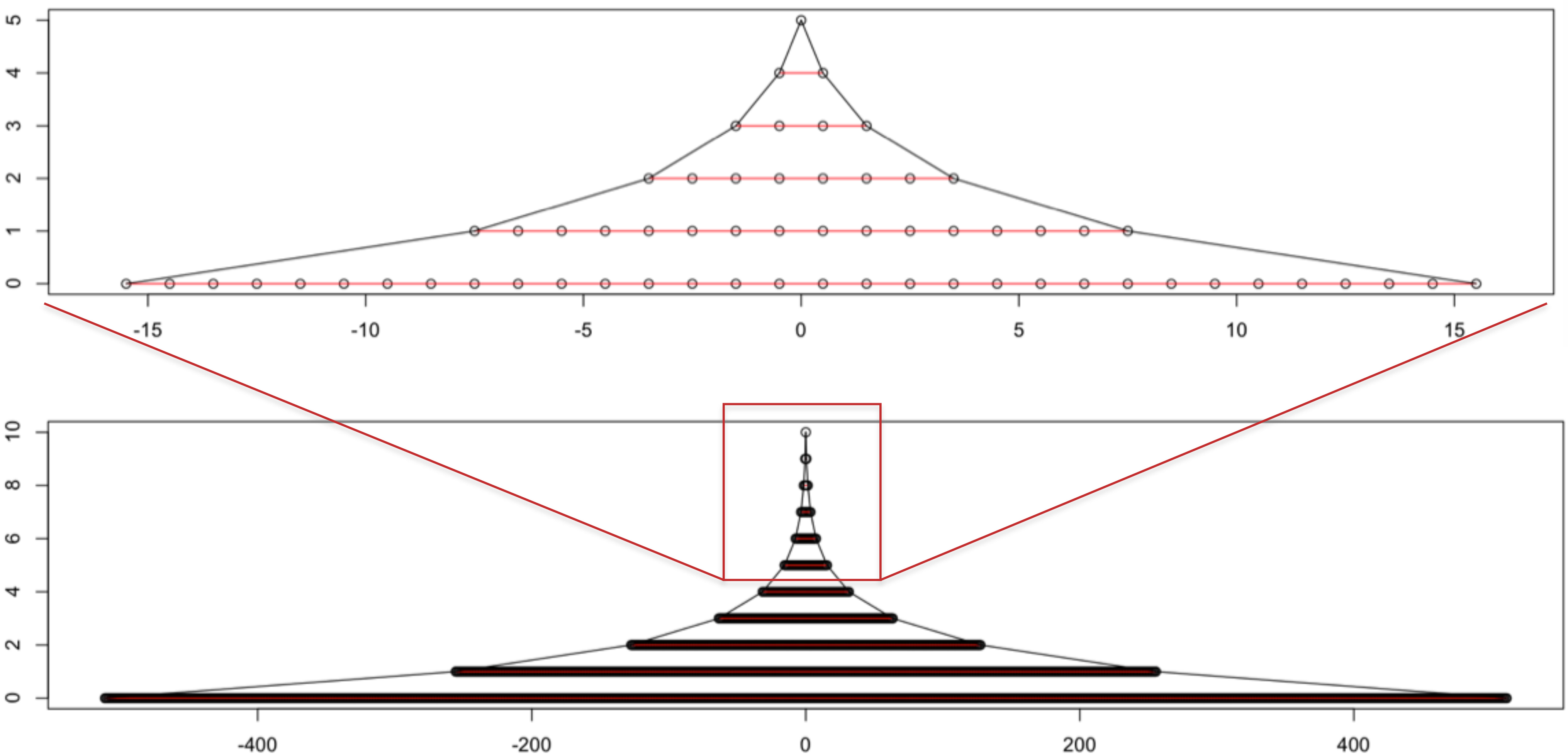
Balanced Binary Tree
Minimum possible height



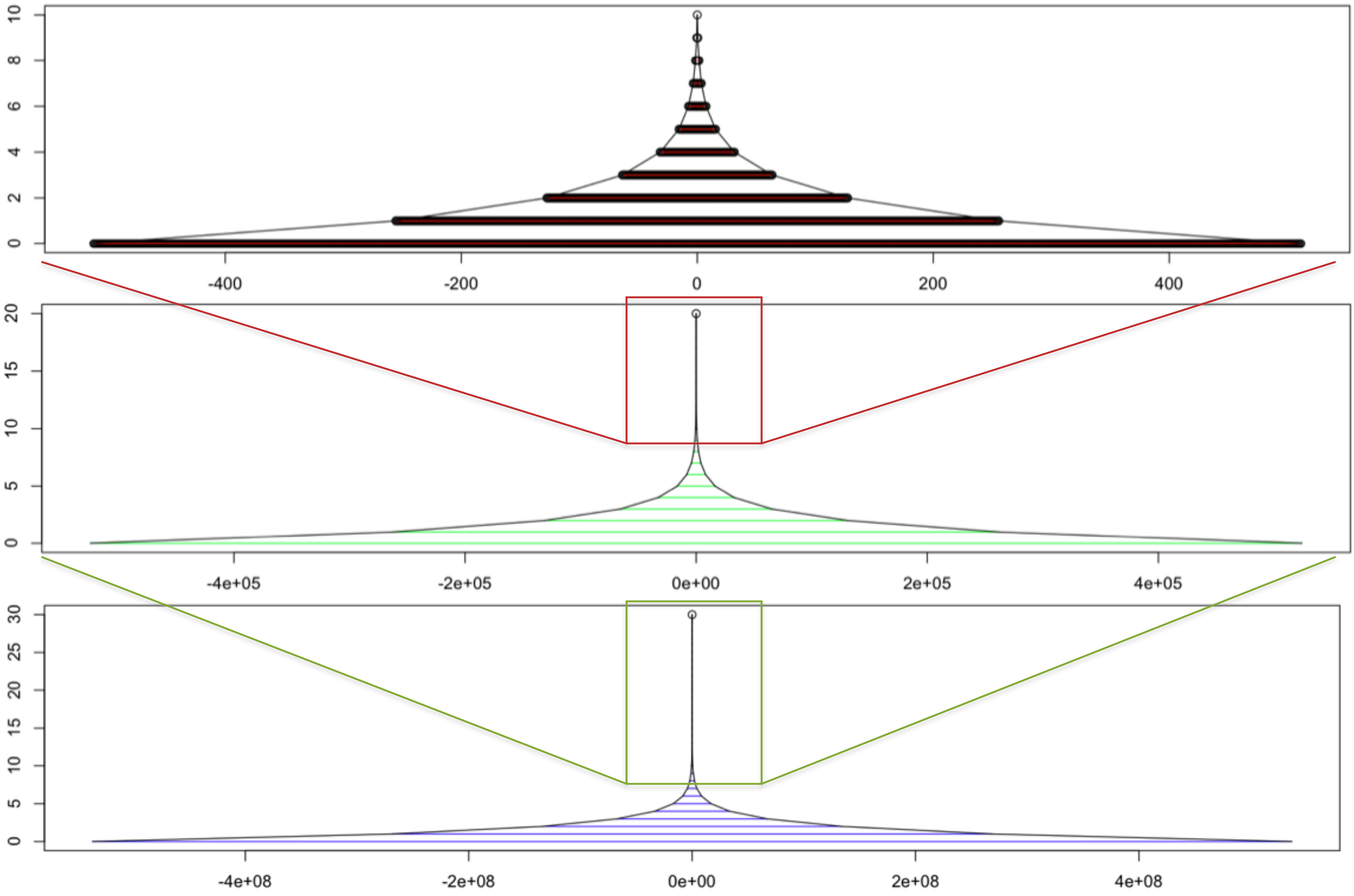
Unbalanced Tree
Non-minimum height

What is the height of a balanced binary tree?	lg n
What is the height of a balanced 3-ary tree?	$\log_3 n$
What is the height of a k-ary tree?	$\log_k n$
How much taller is a binary tree than a k-ary tree?	$\lg n / \log_k n = \lg k$
What is the maximum height of an unbalanced tree?	n-1 ☹

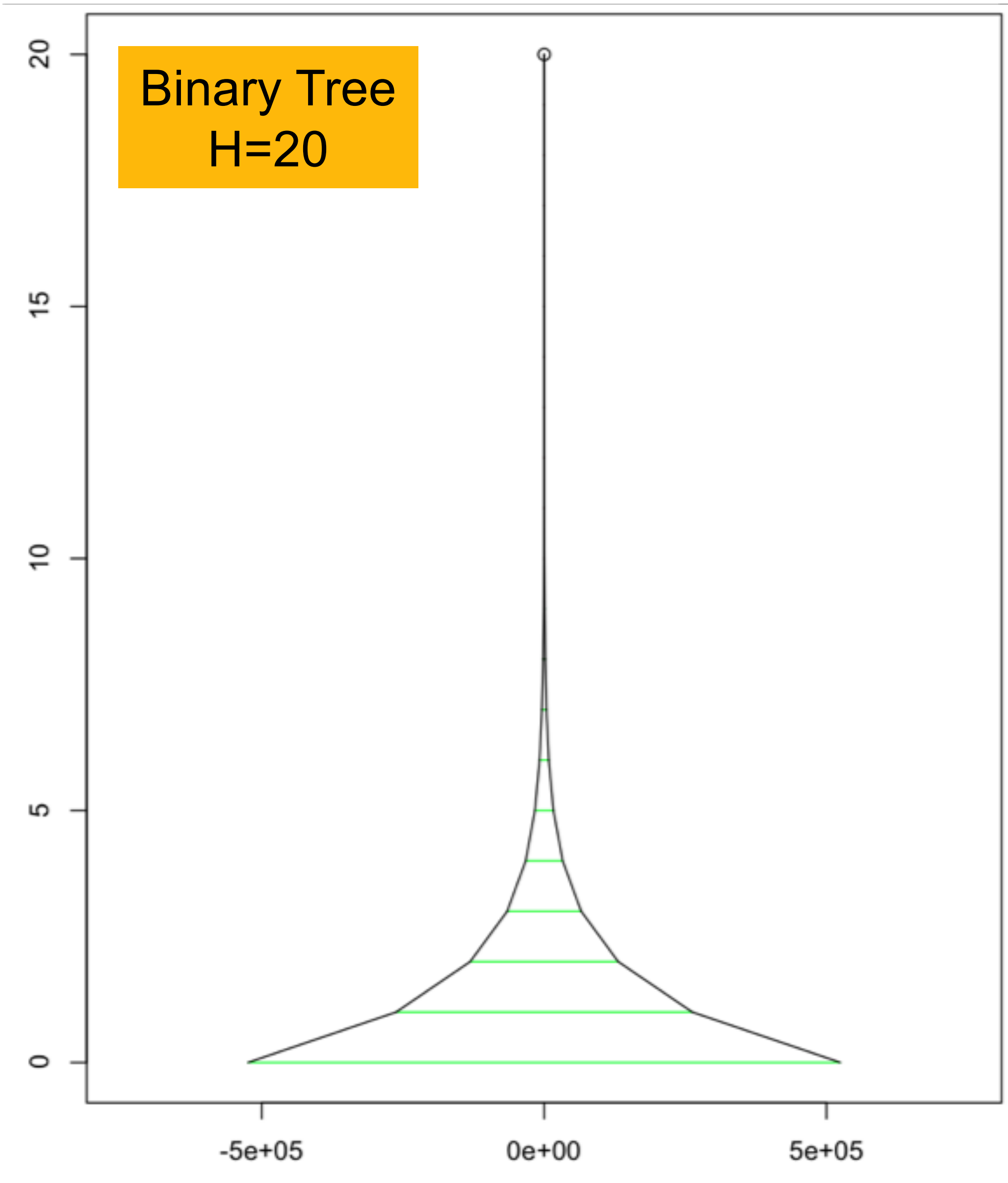
Tree Heights

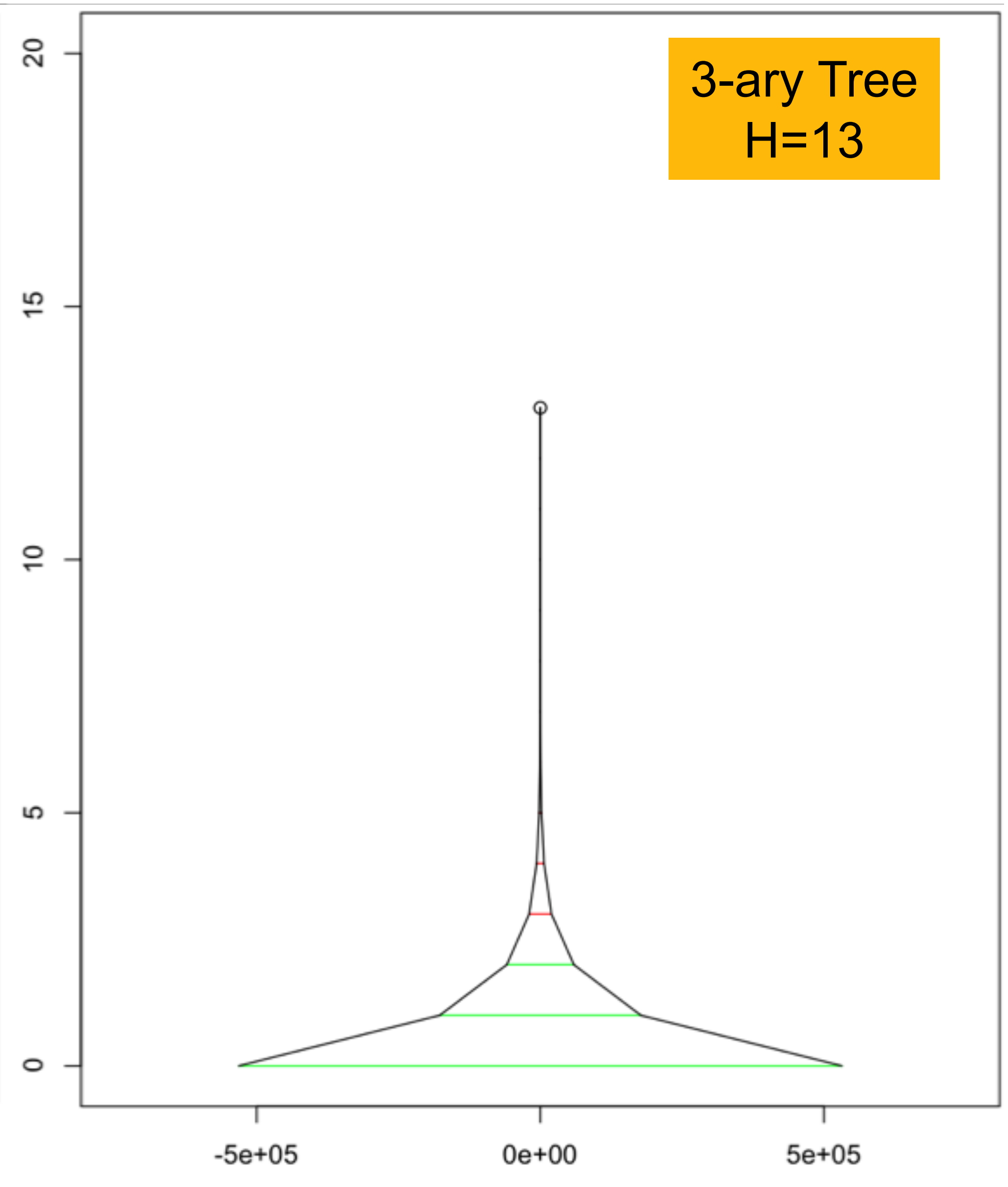


Tree Heights



Tree Heights
(~1M leaves / ~2M nodes)





Tree Interface

```
public interface Tree<T>{
    Position <T> insertRoot(T t )
        throws TreeNotEmptyException;

    Position <T> insertChild(Position <T> p, T t)
        throws InvalidPositionException;

    boolean empty();

    Position <T> root()
        throws TreeEmptyException;

    Position <T>[] children(Position <T> p)
        throws InvalidPositionException , LeafException;

    Position <T> parent(Position<T> p)
        throws InvalidPositionException ;

    boolean leaf (Position <T> p)
        throws InvalidPositionException ;

    T remove(Position<T> p)
        throws InvalidPositionException, NotALeafException;
}
```

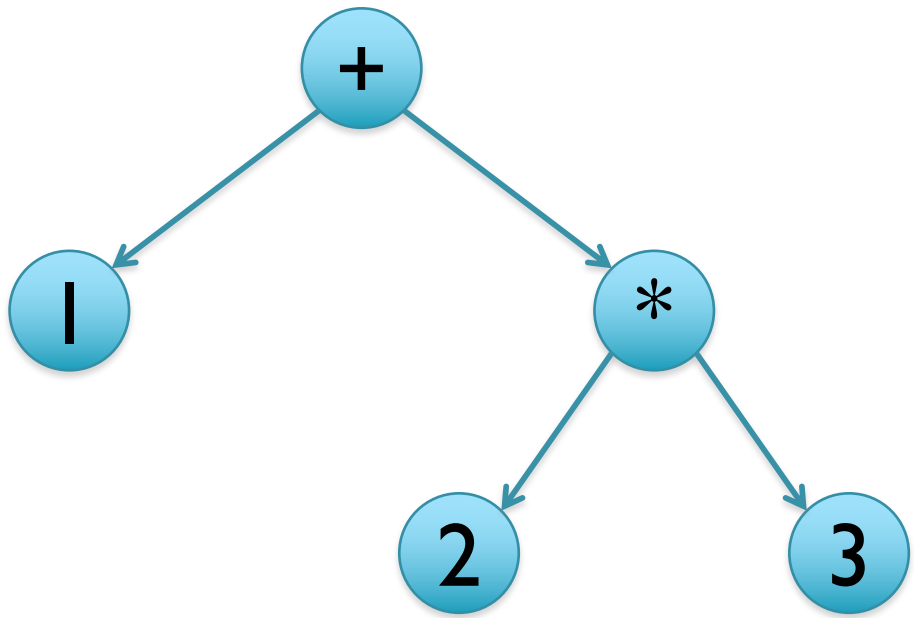
Why insertRoot?

Why children()?

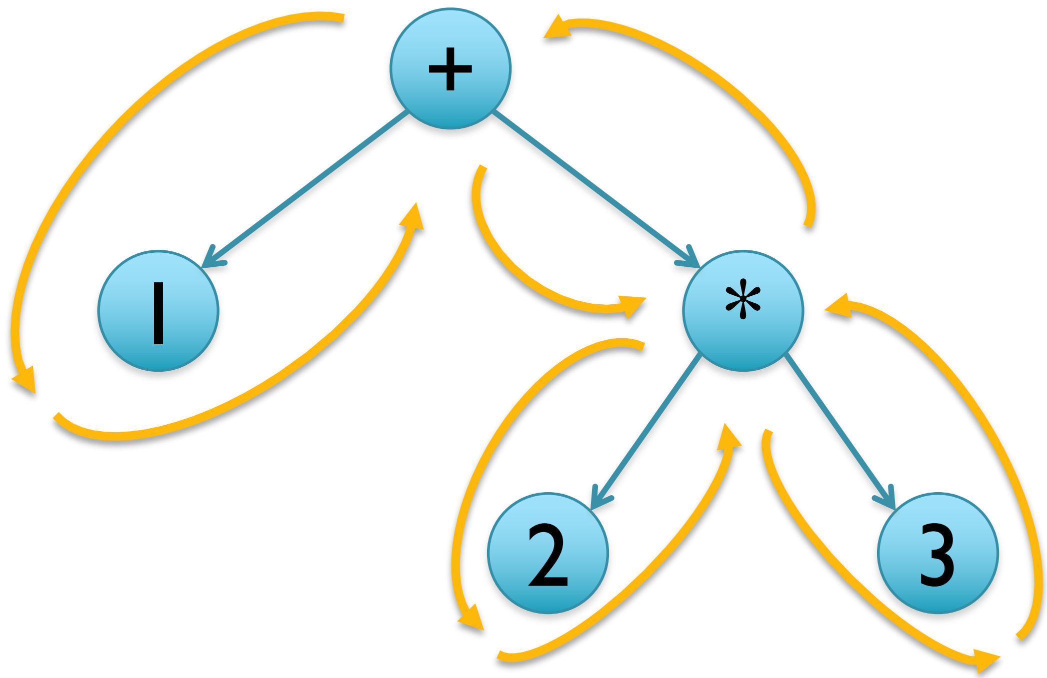
What should parent(root()) return?

What should remove(root()) do?

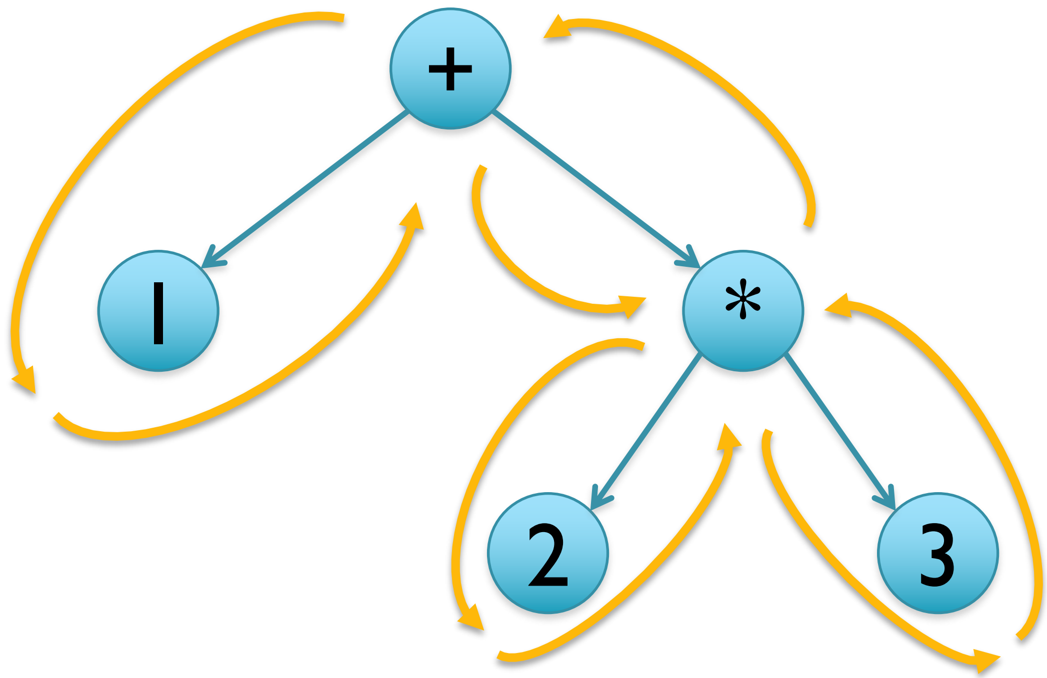
Tree Traversals



Tree Traversals



Tree Traversals

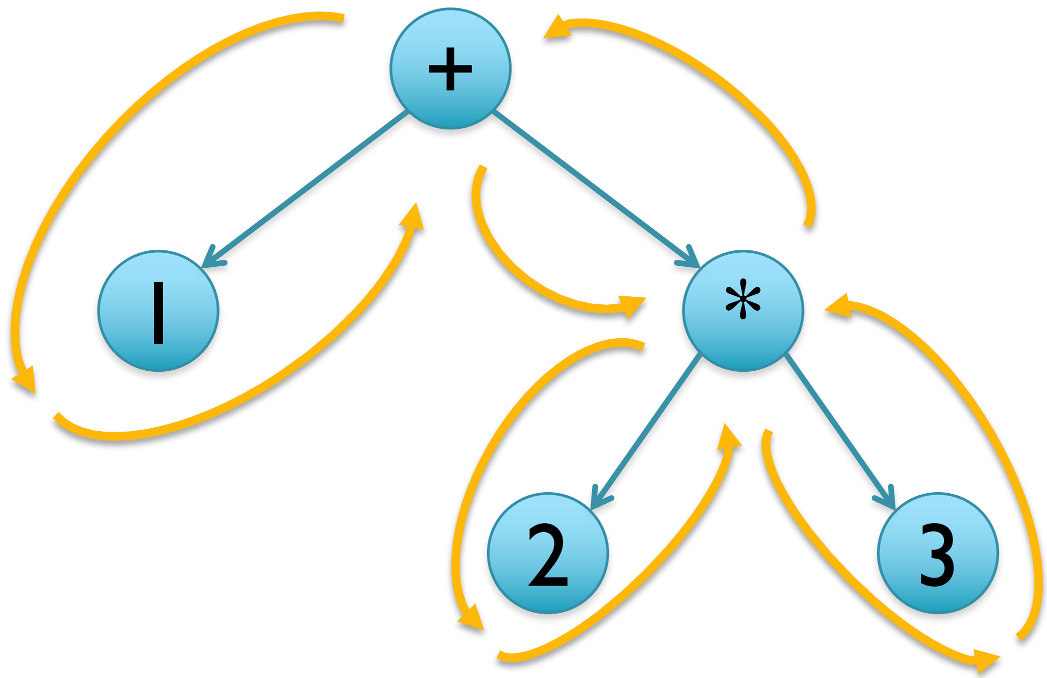


Note here we visit children from left to right, but could go right to left

Notice we visit internal nodes 3 times:
1: stepping down from parent
2: after visiting first child
3: after visiting second child

Different algs work at different times
1: preorder
2: inorder
3: postorder

Tree Traversals

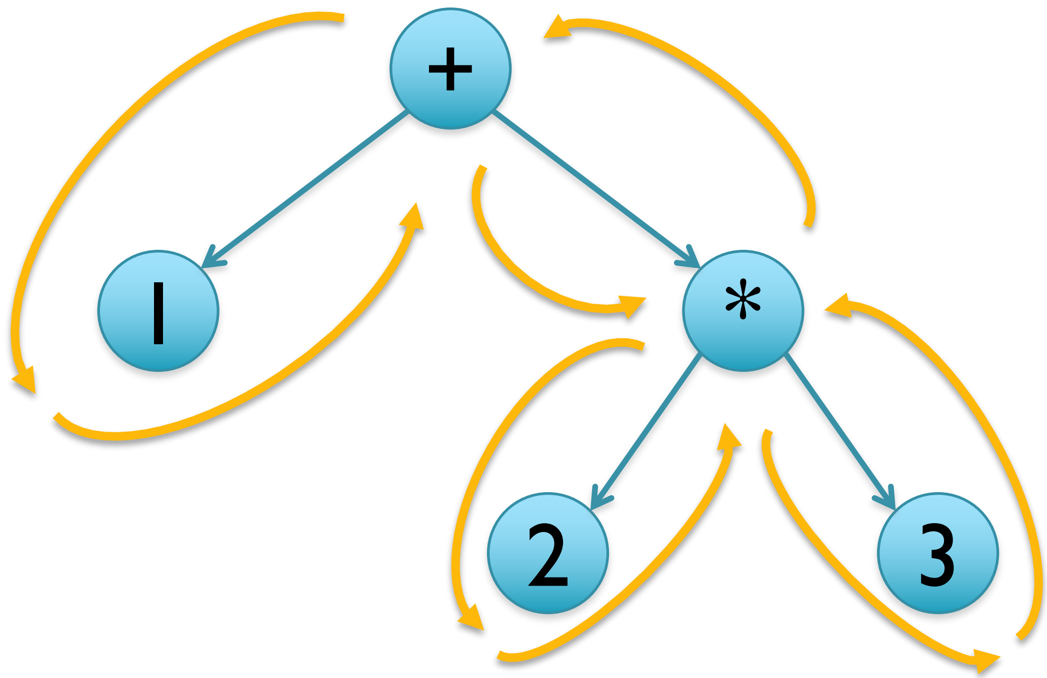


Note here we visit children from left to right, but could go right to left

Notice we visit internal nodes 3 times:
- 1: stepping down from parent
- 2: after visiting first child
- 3: after visiting second child

Different algs work at different times
- 1: preorder + 1 * 2 3
- 2: inorder
- 3: postorder

Tree Traversals

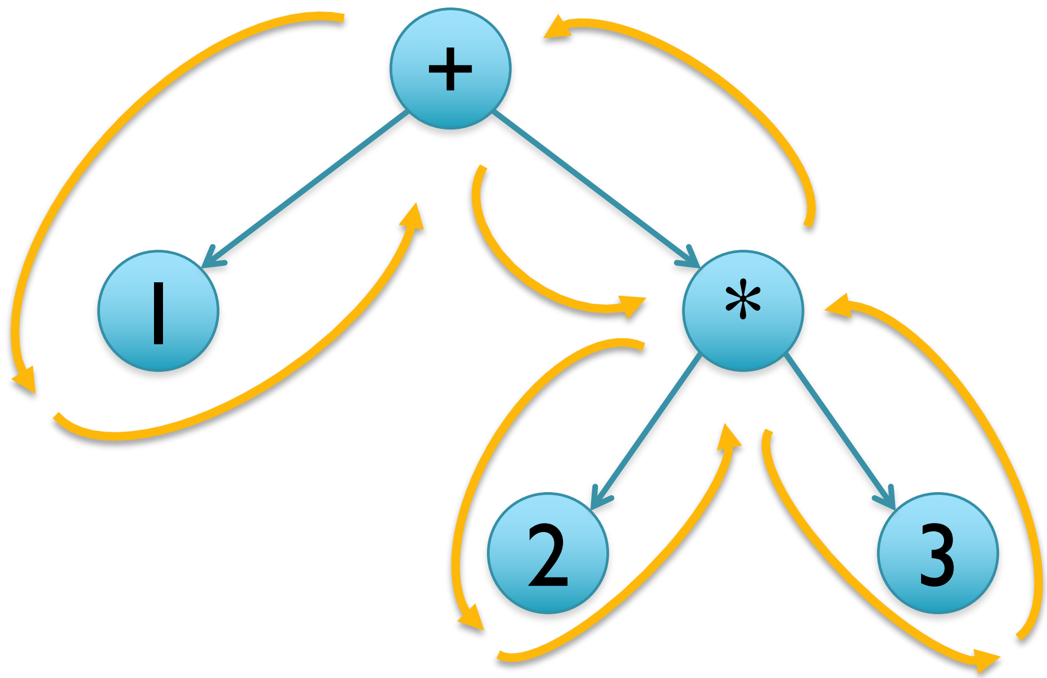


Note here we visit children from left to right, but could go right to left

Notice we visit internal nodes 3 times:
1: stepping down from parent
2: after visiting first child
3: after visiting second child

Different algs work at different times
1: preorder + 1 * 2 3
2: inorder 1 + 2 * 3
3: postorder

Tree Traversals



Note here we visit children from left to right, but could go right to left

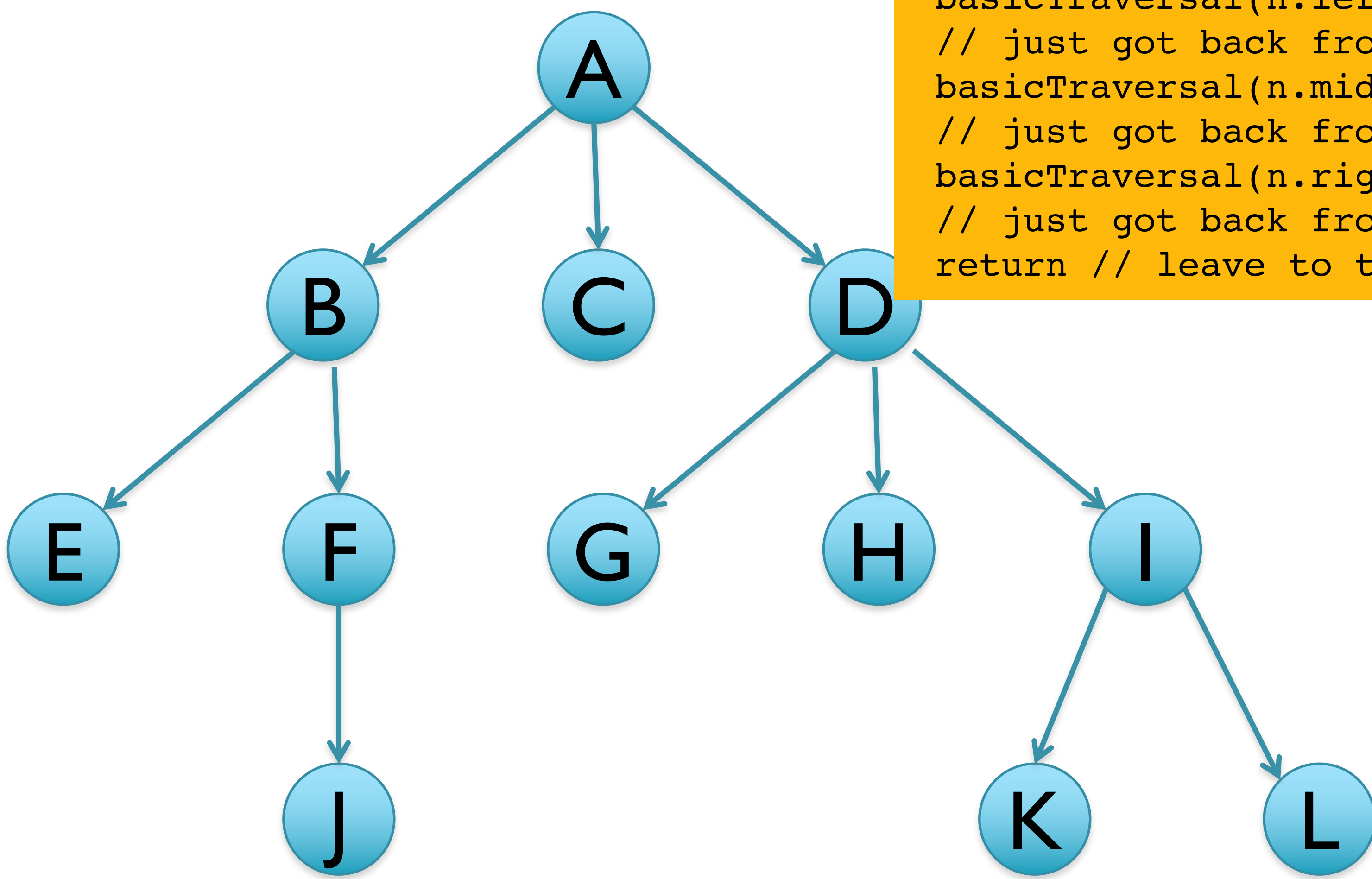
Notice we visit internal nodes 3 times:

1: stepping down from parent
2: after visiting first child
3: after visiting second child

Different algs work at different times

1: preorder	+ 1 * 2 3
2: inorder	1 + 2 * 3
3: postorder	1 2 3 * +

Traversal Implementations



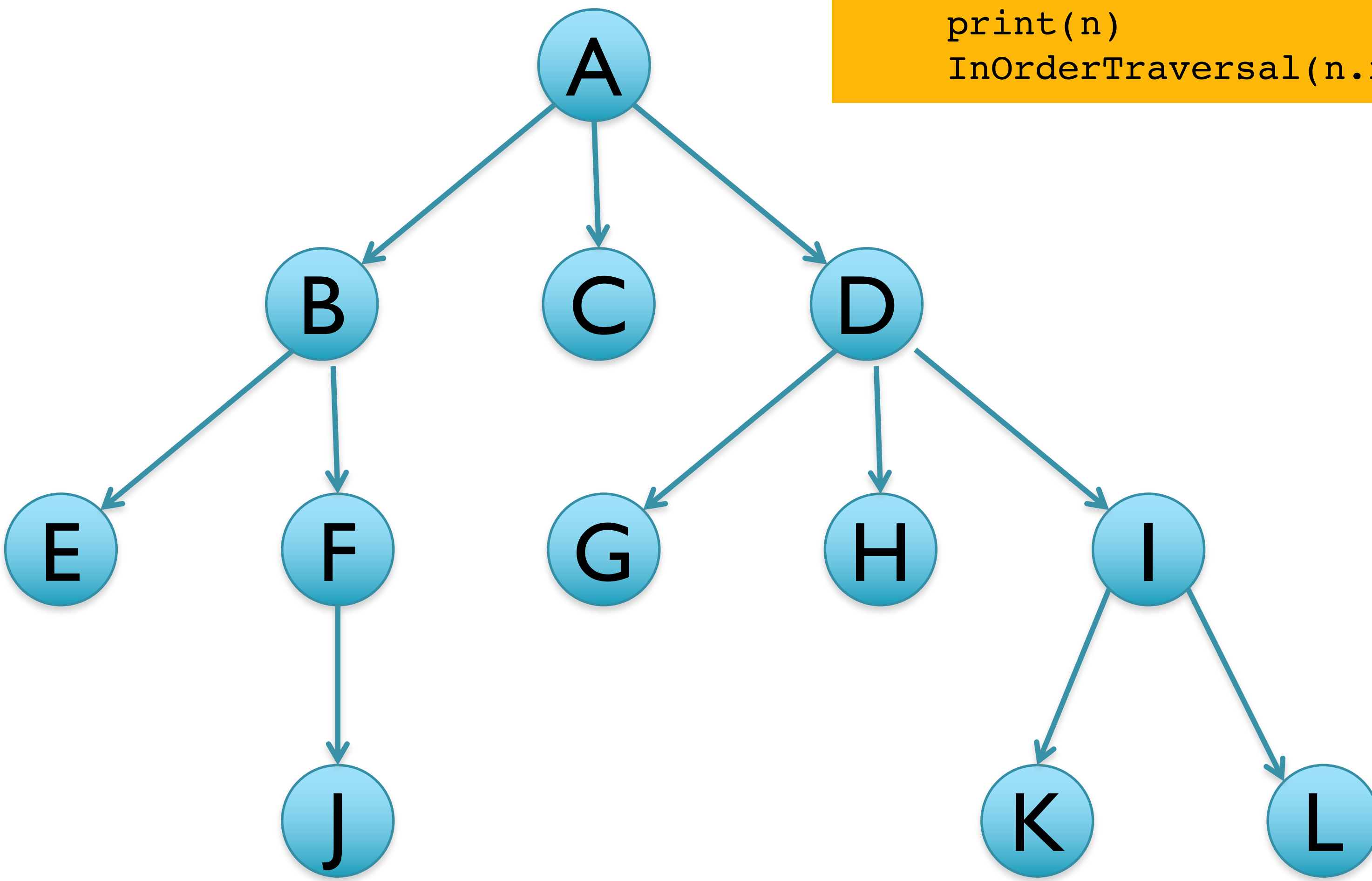
How to traverse?

```
basicTraversal(Node n):
 // just entered from top
 basicTraversal(n.left)
 // just got back from left
 basicTraversal(n.middle)
 // just got back from middle
 basicTraversal(n.right)
 // just got back from right
 return // leave to top
```

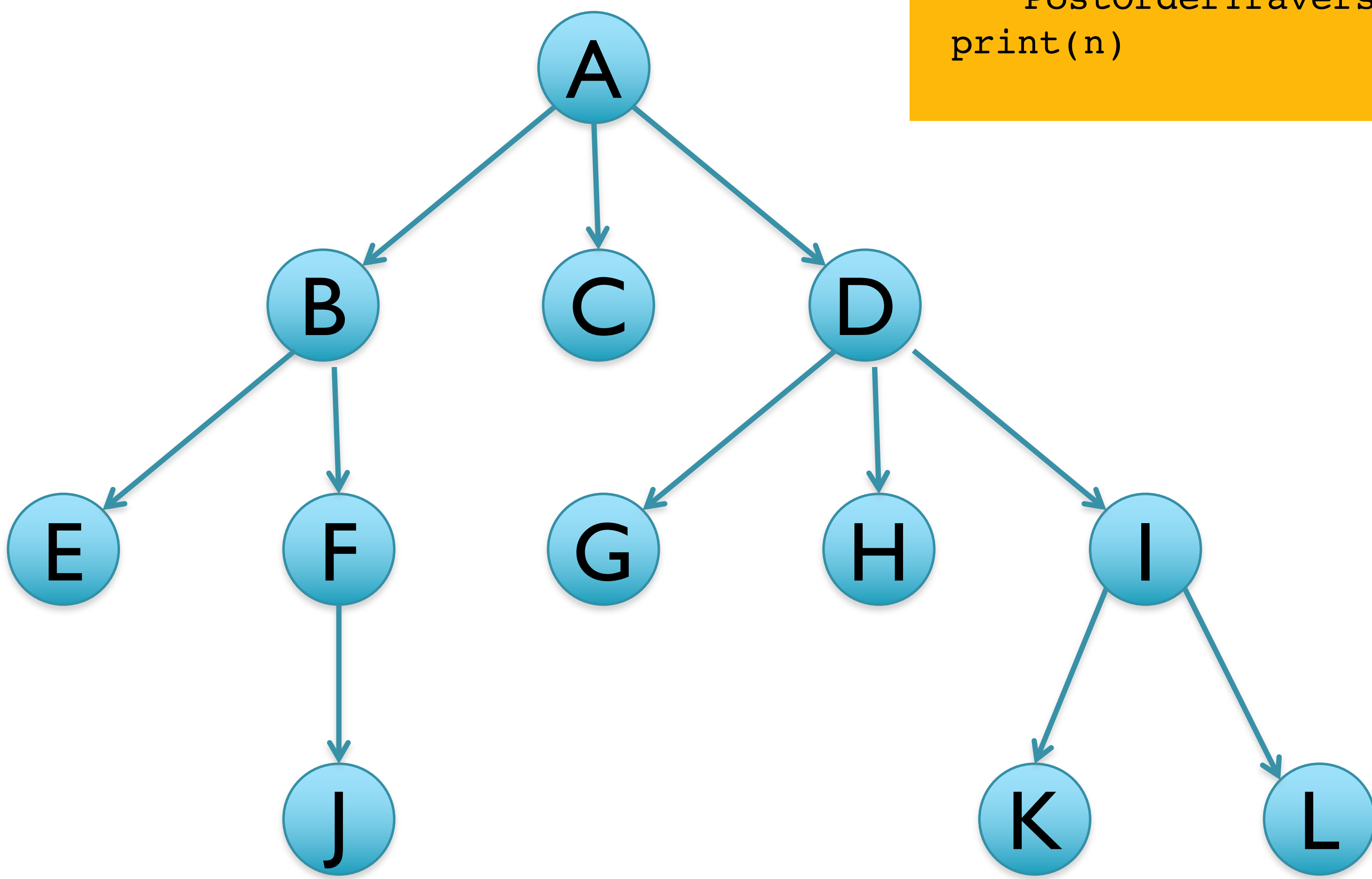
InOrder Traversals

How to inorder print?

```
InOrderTraversal(Node n):
  if n is not null
    InOrderTraversal(n.left)
    print(n)
    InOrderTraversal(n.right)
```



PostOrder Traversals

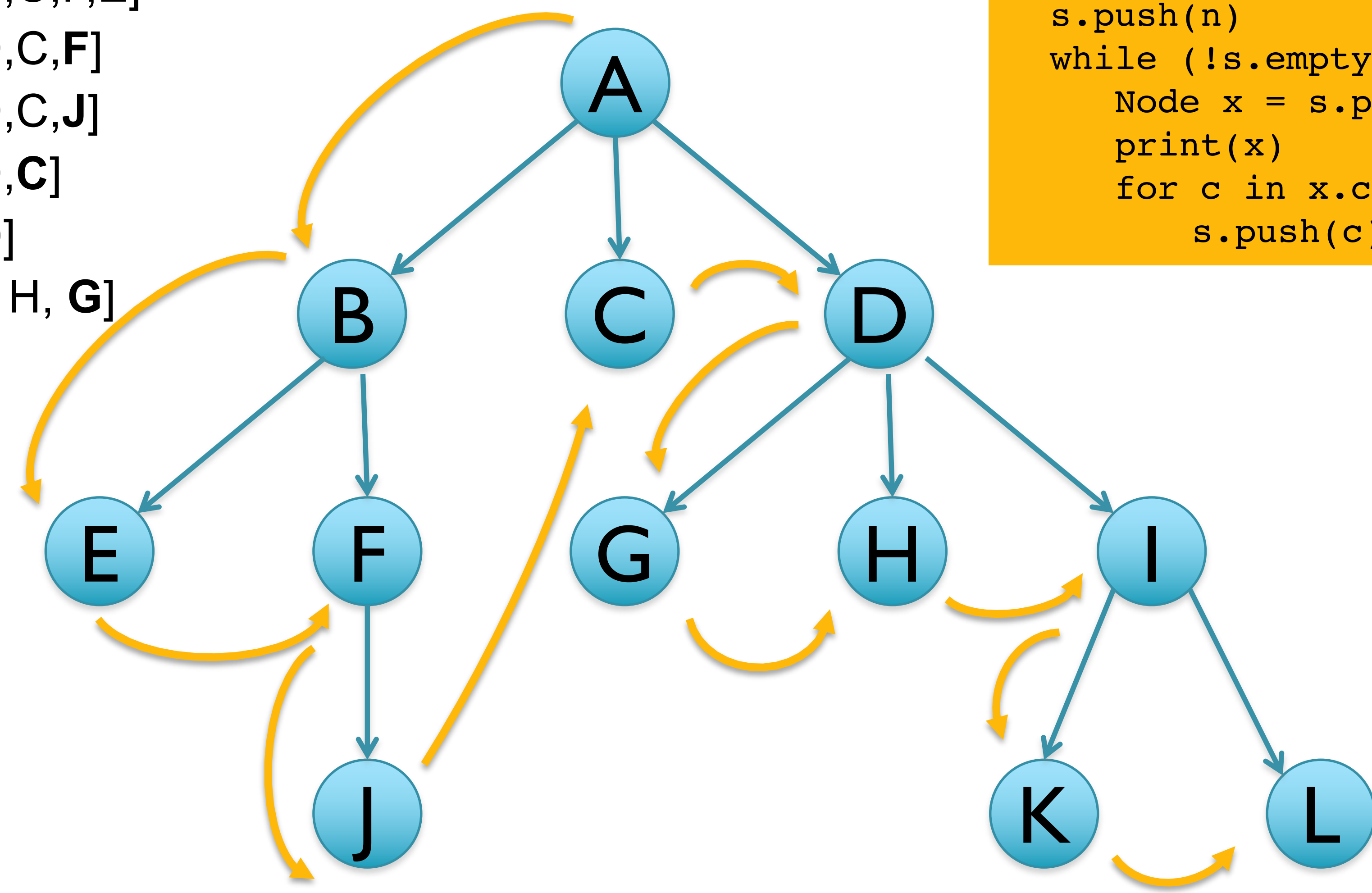


How to postorder print?

```
PostOrderTraversal(Node n):
 for c in x.children:
     PostOrderTraversal(c)
 print(n)
```

PreOrder Traversals

[**A**]
[D,C,**B**]
[D,C,F,**E**]
[D,C,**F**]
[D,C,**J**]
[D,**C**]
[**D**]
[I, H, **G**]
…



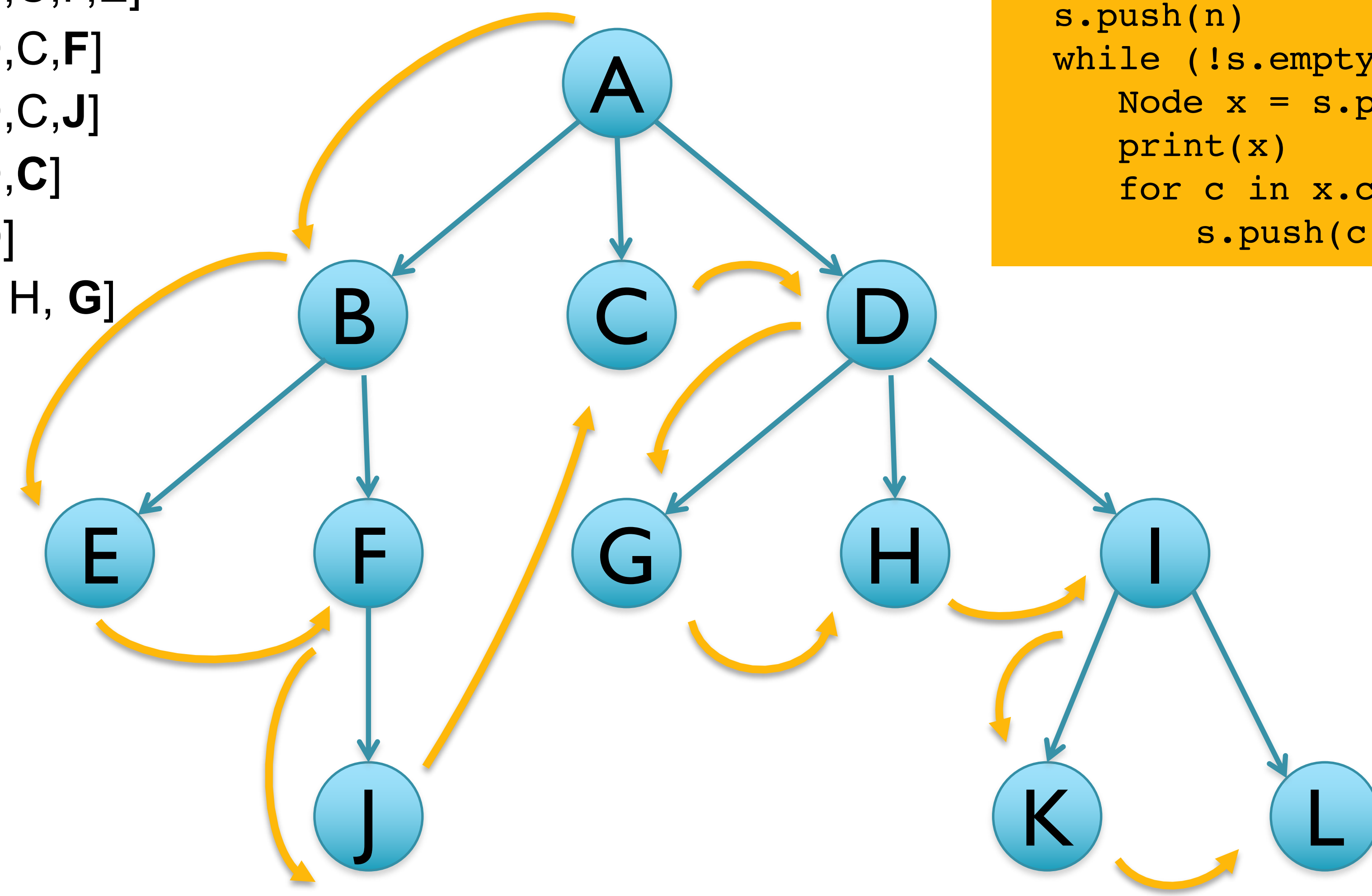
How to preorder print?

```
PreOrderTraversal(Node n):
  Stack s
  s.push(n)
  while (!s.empty()):
     Node x = s.pop()
     print(x)
     for c in x.children:
        s.push(c)
```

PreOrder Traversals

[**A**]
[D,C,**B**]
[D,C,F,**E**]
[D,C,**F**]
[D,C,**J**]
[D,**C**]
[**D**]
[I, H, **G**]
…

Stack leads to a Depth-First Search



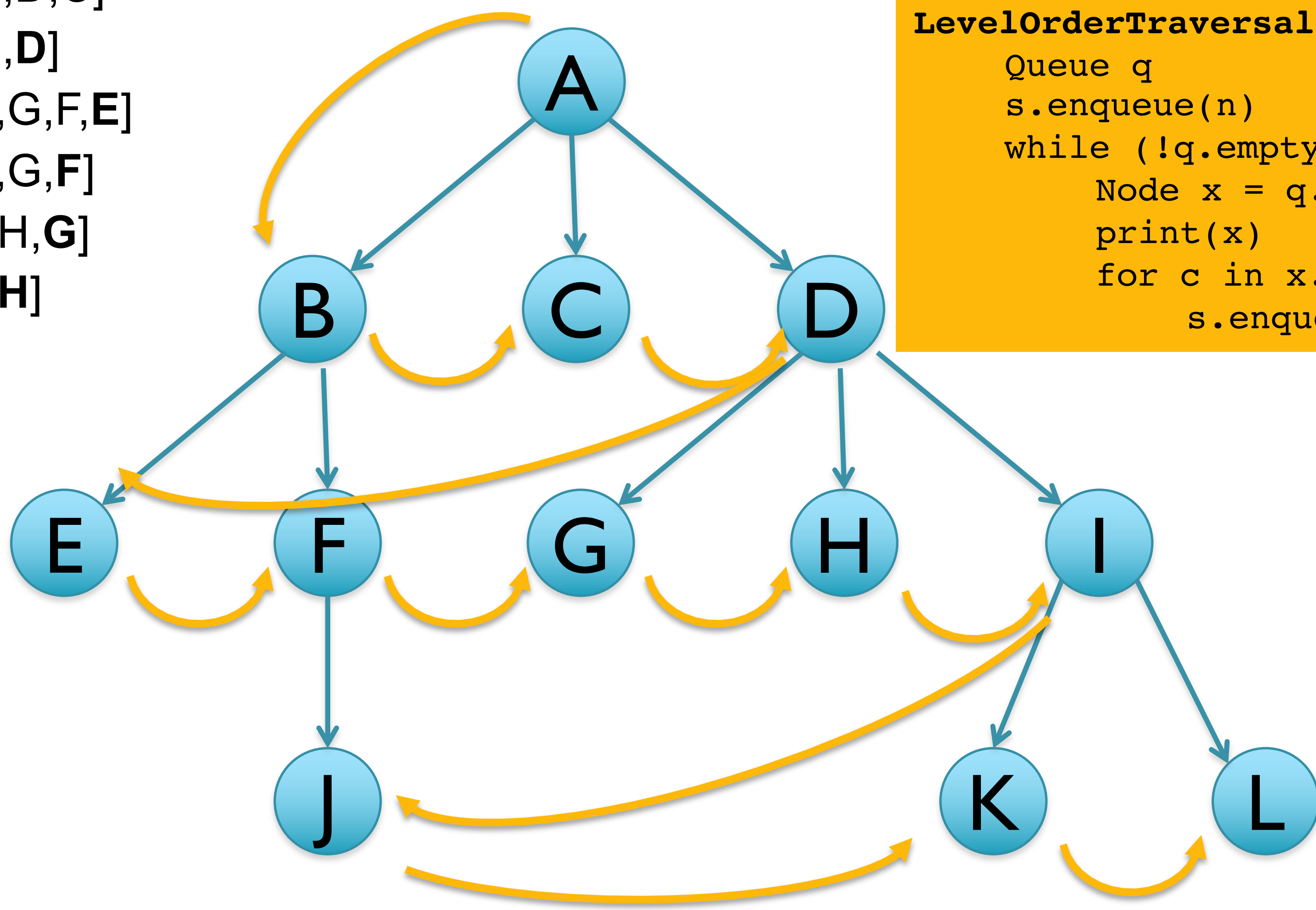
How to preorder print?

```
PreOrderTraversal(Node n):
  Stack s
  s.push(n)
  while (!s.empty()):
     Node x = s.pop()
     print(x)
     for c in x.children:
        s.push(c)
```

Level Order Traversals

[**A**]
[D,C,**B**]
[F,E,D,**C**]
[F,E,**D**]
[I,H,G,F,**E**]
[I,H,G,**F**]
[J,I,H,**G**]
[J,I,**H**]
…

Queue leads to a Breadth-First Search



How to level order print?

(A) (B C D) (E F G H I) (J K L)

```
LevelOrderTraversal(Node n):
    Queue q
    s.enqueue(n)
    while (!q.empty()):
        Node x = q.dequeue()
        print(x)
        for c in x.children:
            s.enqueue(c)
```

Call back interface

```
public interface Tree<T> {
    ...
    preorder(Operation<T> o);
    inorder(Operation<T> o);
    postorder(Operation<T> o);
    ...
}

// The Operation<T> interface would look like this:
public interface Operation<T> {
    void do(Position<T> p);
}

public class PrintOperation<T> implements Operation<T> {
    public void do(Position<T> p) {
        System.out.println(p.get());
    }
}
...
PrintOperation op = new PrintOperation();
tree.inorder(op);
```

This works, but we will need 3 separate methods that have almost exactly the same code

Multiple Traversals

```
public interface Operation<T> {
    void pre(Position<T> p);
    void in(Position<T> p);
    void post(Position<T> p);
}

public interface Tree<T> {
     ...
     traverse(Operation<T> o);
    ...
}

  // Tree implementation pseudo-code:
  niceTraversal(Node n, Operation o):
     if n is not null:
         o.pre(n)
         niceTraversal(n.left, o)
         o.in(n)
         niceTraversal(n.right, o)
         o.post(n)
}
```

Just implement the method you need

Oh wait, we would have to implement all 3 methods ☹

One methods calls client code for all 3 operators

Java Abstract Classes

An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

If a class includes abstract methods, then the class itself must be declared abstract, as in:

```
public abstract class GraphicObject {
   // declare fields
   // declare nonabstract and abstact methods
   void setPen(Pen p) { this.pen = p }
   abstract void draw();
}
```

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation. However, with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods.

Multiple Traversals

```
public abstract class Operation<T> {
    void pre(Position<T> p) {}
    void in(Position<T> p) {}
    void post(Position<T> p) {}
}

public interface Tree<T> {
    ...
    traverse(Operation<T> o);
   ...
}

  // Tree implementation pseudo-code:
  niceTraversal(Node n, Operation o):
    if n is not null:
        o.pre(n)
        niceTraversal(n.left, o)
        o.in(n)
        niceTraversal(n.right, o)
        o.post(n)
}
```

Client extends Operation<T> but overrides just the methods that are needed ☺

Implementation (1)

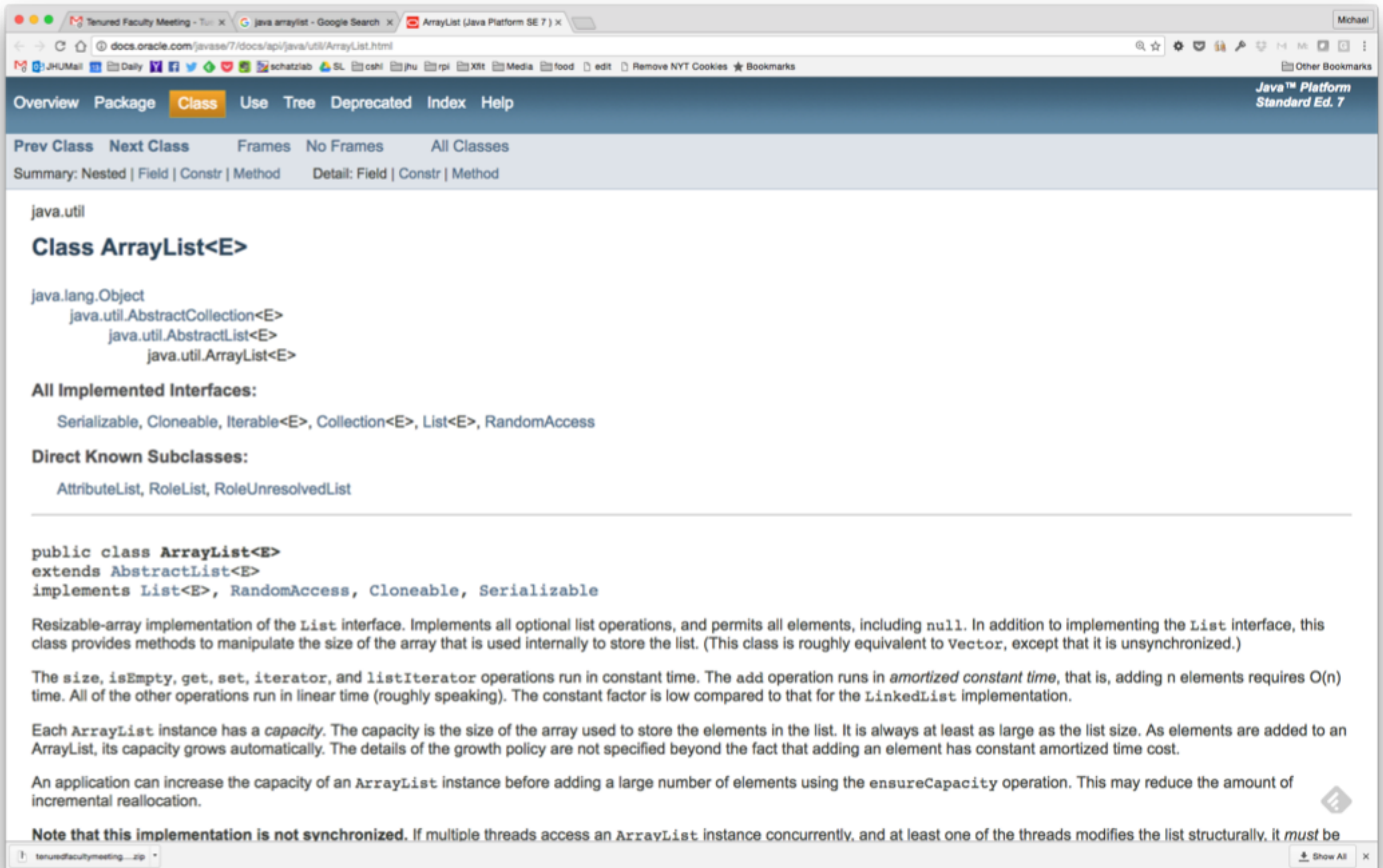
```
public class TreeImplementation<T> implements Tree<T> {
...
  private static class Node<T> implements Position<T> {
    T data;
    Node<T> parent;
    ArrayList<Node<T>> children;

    public Node(T t) {
        this.children = new ArrayList<Node<T>>();
        this.data = t;
    }

    public T get() {
        return this.data;
    }

    public void put(T t) {
        this.data = t;
    }
  }
}
```

Constructor ensures children, data are initialized correctly

What other fields might we want to include? (Hint: Position<>)

Should set the “color” field to point to this Tree so Position<> can be checked

ArrayList


Overview Package Class Use Tree Deprecated Index Help

Java™ Platform Standard Ed. 7

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

Class ArrayList<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

Each `ArrayList` instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

An application can increase the capacity of an `ArrayList` instance before adding a large number of elements using the `ensureCapacity` operation. This may reduce the amount of incremental reallocation.

Note that this implementation is not synchronized. If multiple threads access an `ArrayList` instance concurrently, and at least one of the threads modifies the list structurally, it *must* be

http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

Implementation (2)

```
public Position<T> insertRoot(T t) throws InsertionException {
    if (this.root != null) {
        throw new InsertionException();
    }
    this.root = new Node<T>(t);
    this.elements += 1;
    return this.root;
}


public Position<T> insertChild(Position<T> pos, T t)
                              throws InvalidPositionException {
    Node<T> p = this.convert(pos);
    Node<T> n = new Node<T>(t);
    n.parent = p;
    p.children.add(n);
    this.elements += 1;
    return n;
}
```

convert?

convert method (a private helper) takes a position, validates it, and then returns the Node<T> object hiding behind the position

Implementation (3)

```
public boolean empty() {
    return this.elements == 0;
}

public int size() {
    return this.elements;
}

public boolean hasParent(Position<T> p) throws
                              InvalidPositionException {
    Node<T> n = this.convert(p);
    return n.parent != null;
}

public boolean hasChildren(Position<T> p) throws
                              InvalidPositionException {
    Node<T> n = this.convert(p);
    return !n.children.isEmpty();
}
```

Traversal

```
private void recurse(Node<T> n, Operation<T> o) {
    if (n == null) { return; }
    o.pre(n);
    for (Node<T> c: n.children) {
        this.recurse(c, o);
        // figure out when to call o.in(n)
    }
    o.post(n);
}

public void traverse(Operation<T> o) {
    this.recurse(this.root, o);
}
```

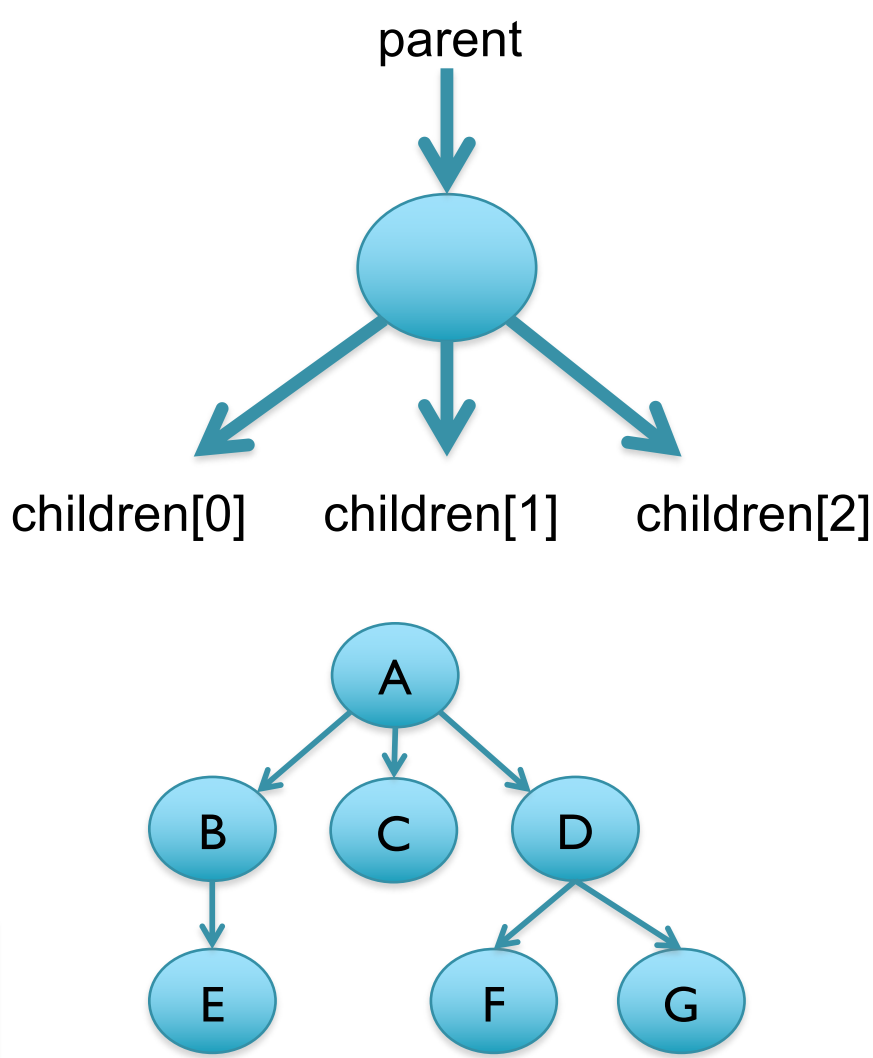
Private helper method working with Node<T> rather than Position<T>

Just make sure we start at root

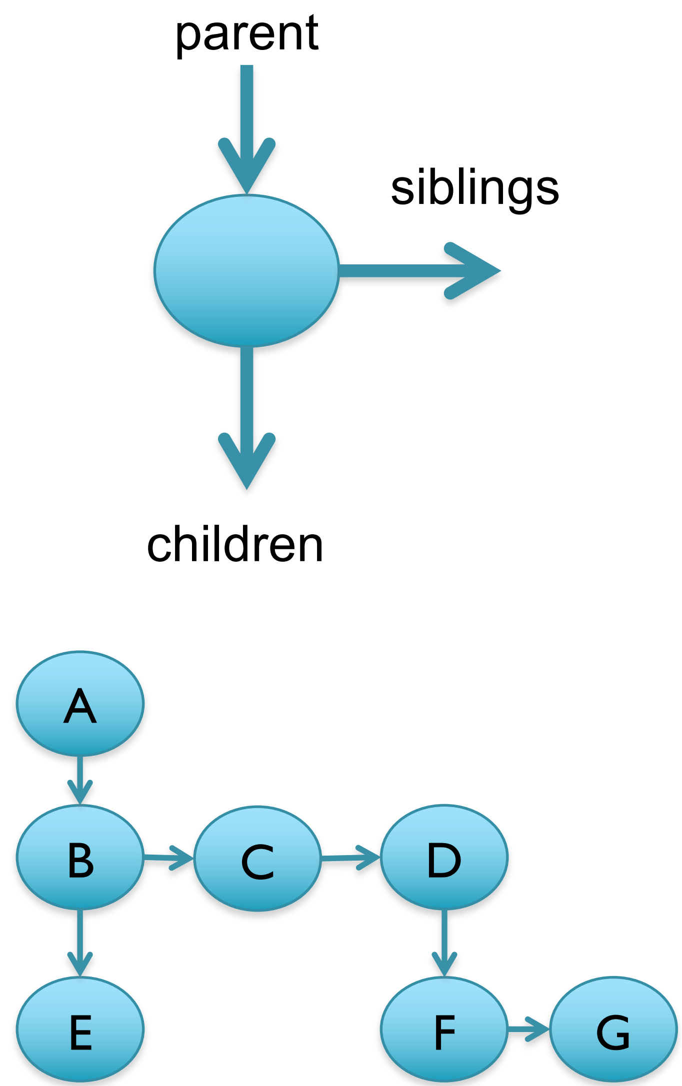
When should we call o.in()?

We don’t want to call the in method after we visit the last child.
We do want to call the in method even for a node with no children

Alternate Implementation



Simple Implementation
Overhead managing children[]



Less Space Overhead
(Except remove may require doubly linked lists)

Next Steps

1. Work on HW4
2. Check on Piazza for tips & corrections!