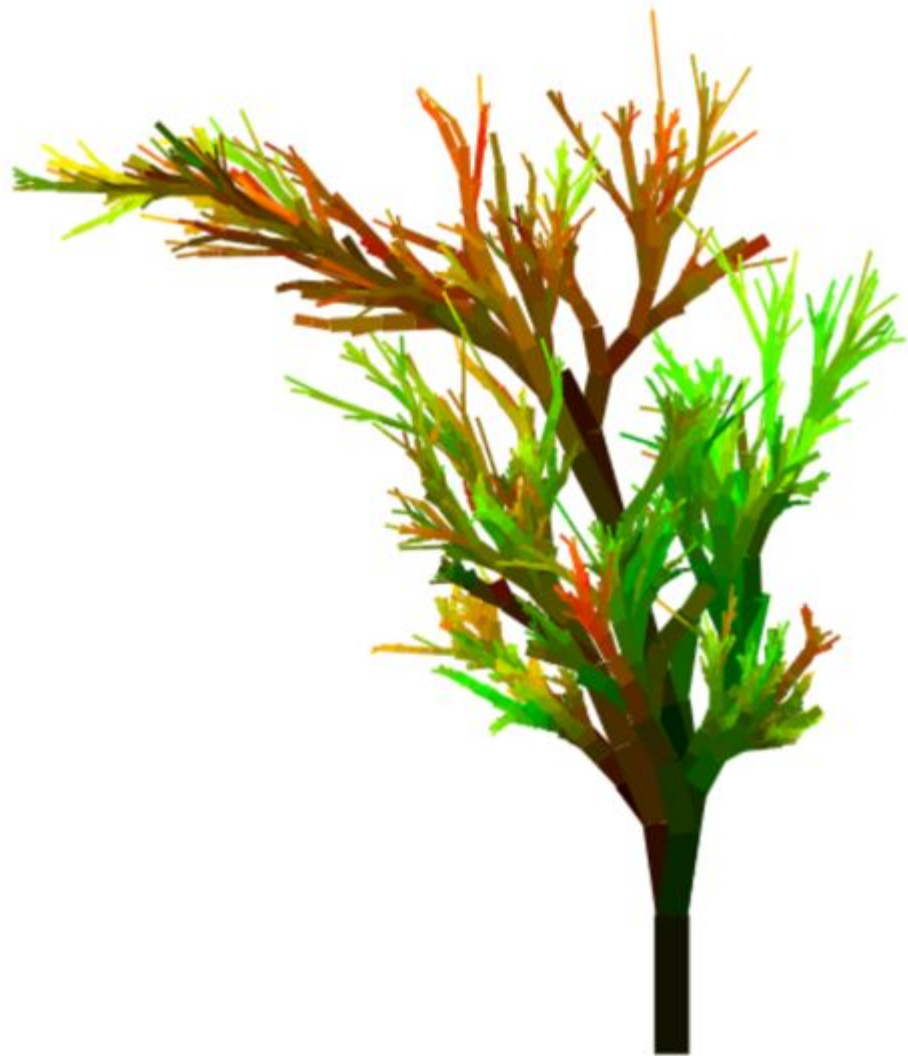


# CS61B

---

## Lecture 25: Advanced Trees

- Tree Traversals
- Level Order Traversal
- Range Finding
- Spatial (a.k.a. Geometric) Trees
- Tree Iterators (Extra)



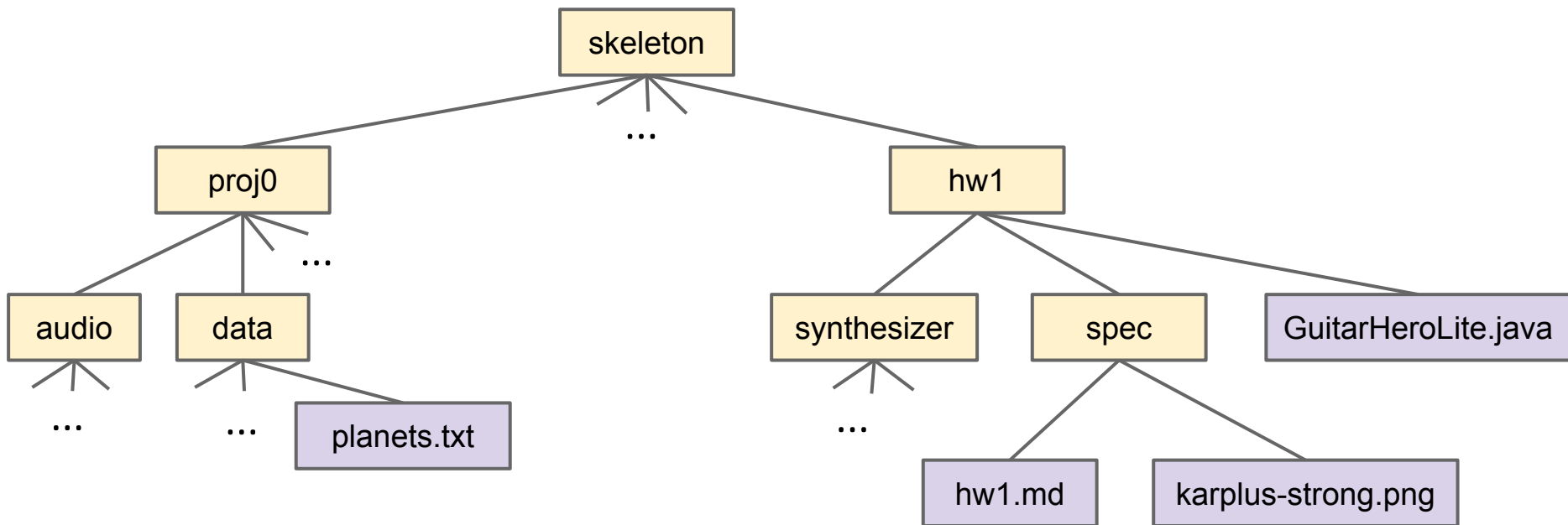
# Traversals

# Rooted Trees

---

We've used BSTs to build Maps and Sets, and Heaps to build a PQ.

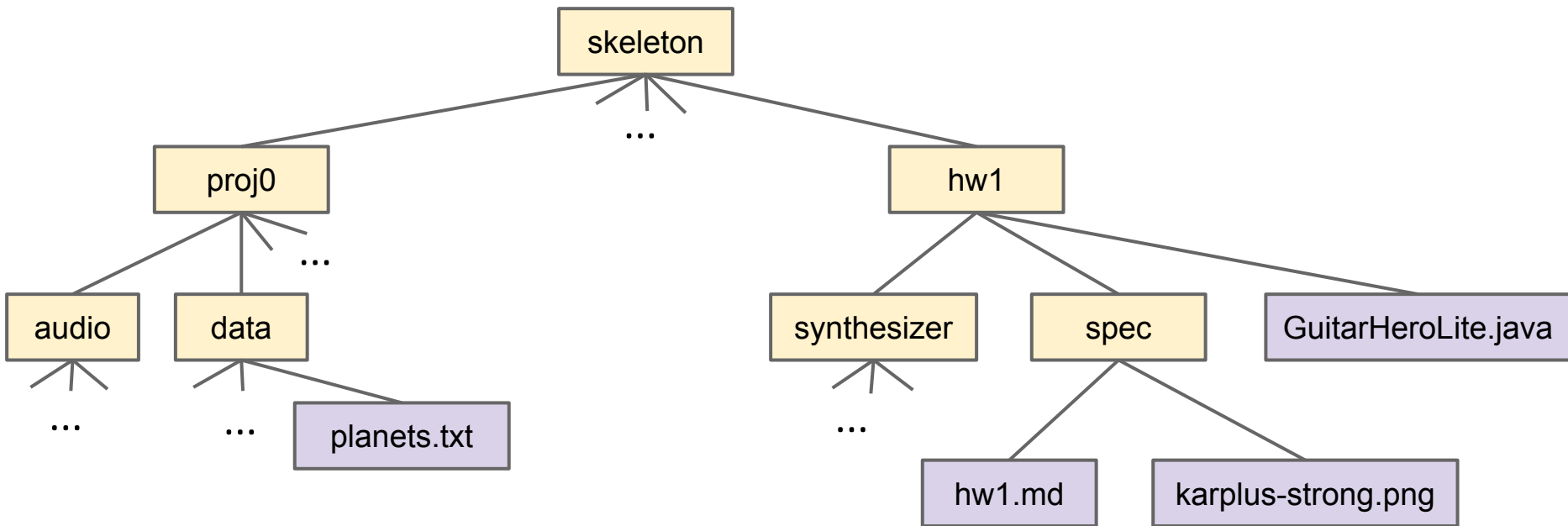
... but trees are a more general concept.



# Rooted Trees

Given such a tree, find how much disk space all the files use.

- What one might call “tree iteration” is usually called “tree traversal.”
- Unlike lists, there are many natural orderings.



# Tree Traversal

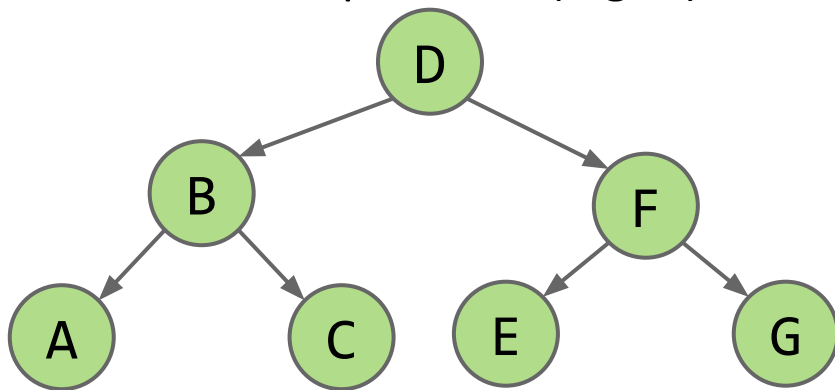
---

## Level Order

- Traverse top-to-bottom, left-to-right (like reading in English):
- We say that the nodes are 'visited' in the given order.

## Depth First Traversals

- Preorder, Inorder, Postorder
- Basic (rough) idea: Traverse “deep nodes” (e.g. A) before shallow ones (e.g. F).

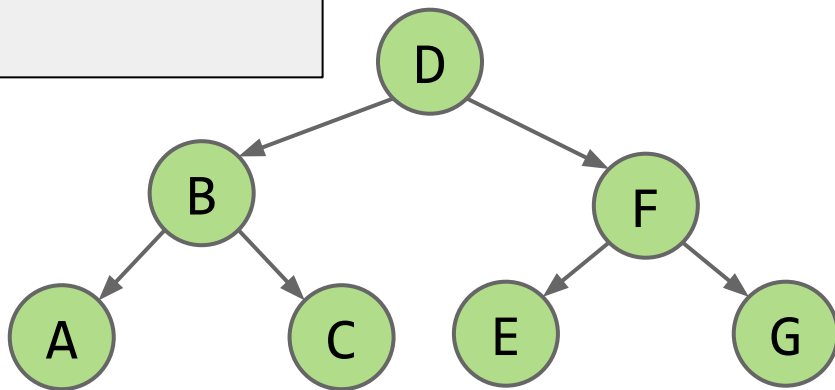


# Depth First Traversals

---

Preorder: “Visit” a node, then traverse its children: D B A C F E G

```
preOrder(BSTNode x) {  
    if (x == null) return;  
    print(x.key)  
    preOrder(x.left)  
    preOrder(x.right)  
}
```



# Depth First Traversals

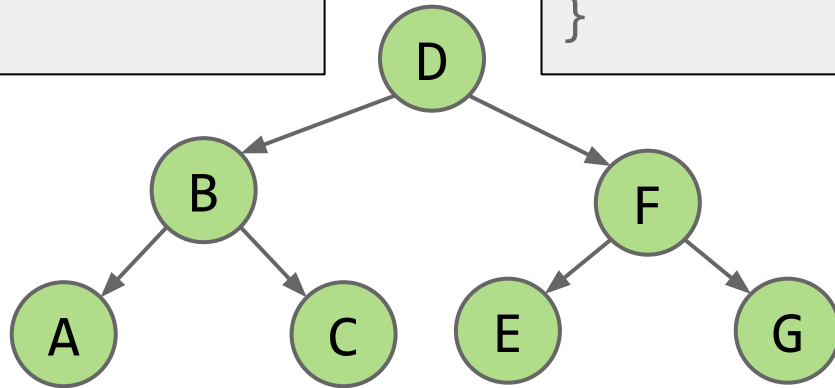
---

Preorder traversal: “Visit” a node, then traverse its children: DBACFEG

Inorder traversal: Traverse left child, visit, then traverse right child: ABCDEFG

```
preOrder(BSTNode x) {  
    if (x == null) return;  
    print(x.key)  
    preOrder(x.left)  
    preOrder(x.right)  
}
```

```
inOrder(BSTNode x) {  
    if (x == null) return;  
    inOrder(x.left)  
    print(x.key)  
    inOrder(x.right)  
}
```



# Depth First Traversals <http://yellkey.com/top>

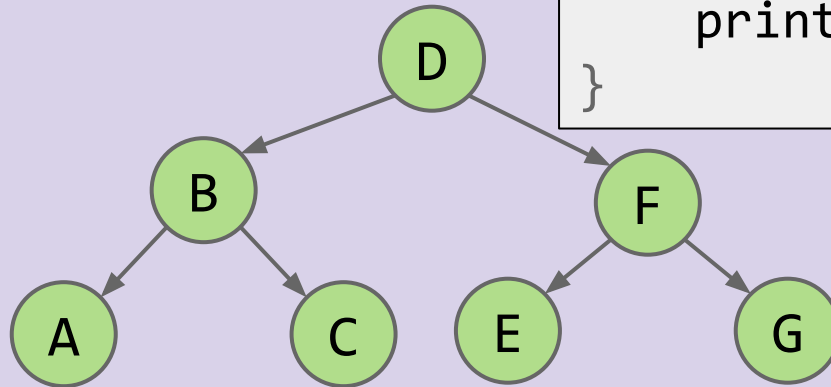
Preorder traversal: "Visit" a node, then traverse its children: DBACFEG

Inorder traversal: Traverse left child, visit, traverse right child: ABCDEFG

Postorder traversal: Traverse left, traverse right, then visit: ???????

1. DBACEFG
2. GFEDCBA
3. GEFCABD
4. ACBEGFD
5. ACBFEGD
6. Other

```
postOrder(BSTNode x) {  
    if (x == null) return;  
    postOrder(x.left)  
    postOrder(x.right)  
    print(x.key)  
}
```





# Depth First Traversals

---

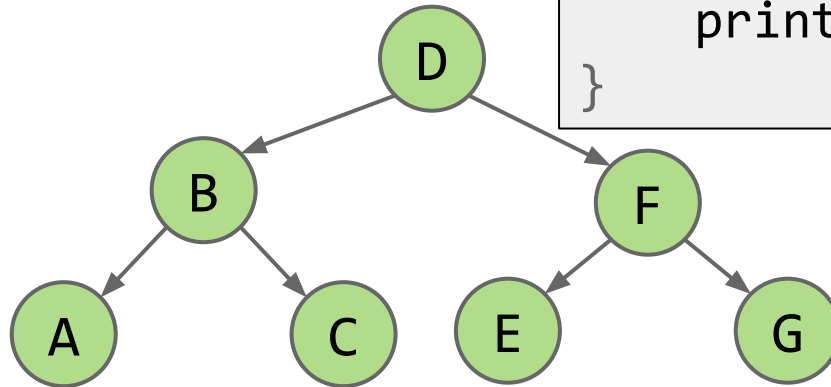
Preorder traversal: “Visit” a node, then traverse its children: DBACFEG

Inorder traversal: Traverse left child, visit, traverse right child: ABCDEFG

Postorder traversal: Traverse left, traverse right, then visit: ACBEGFD

1. DBACEFG
2. GFEDCBA
3. GEFCABD
4. **ACBEGFD**
5. ACBFEGD
6. Other

```
postOrder(BSTNode x) {  
    if (x == null) return;  
    postOrder(x.left)  
    postOrder(x.right)  
    print(x.key)  
}
```

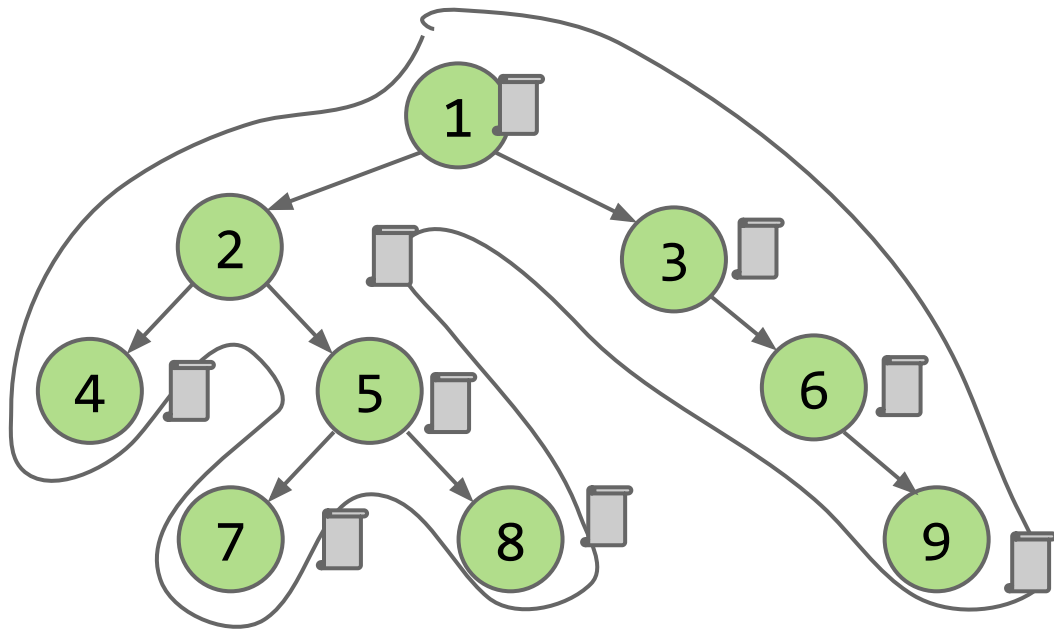


## A Weird Trick

- Preorder traversal: We walk the graph, from top going counter-clockwise. Shout every time we pass the LEFT of a node.
- Inorder traversal: Shout when you cross the bottom.
- Postorder traversal: Shout when you cross the right.

Example: Post-Order Traversal

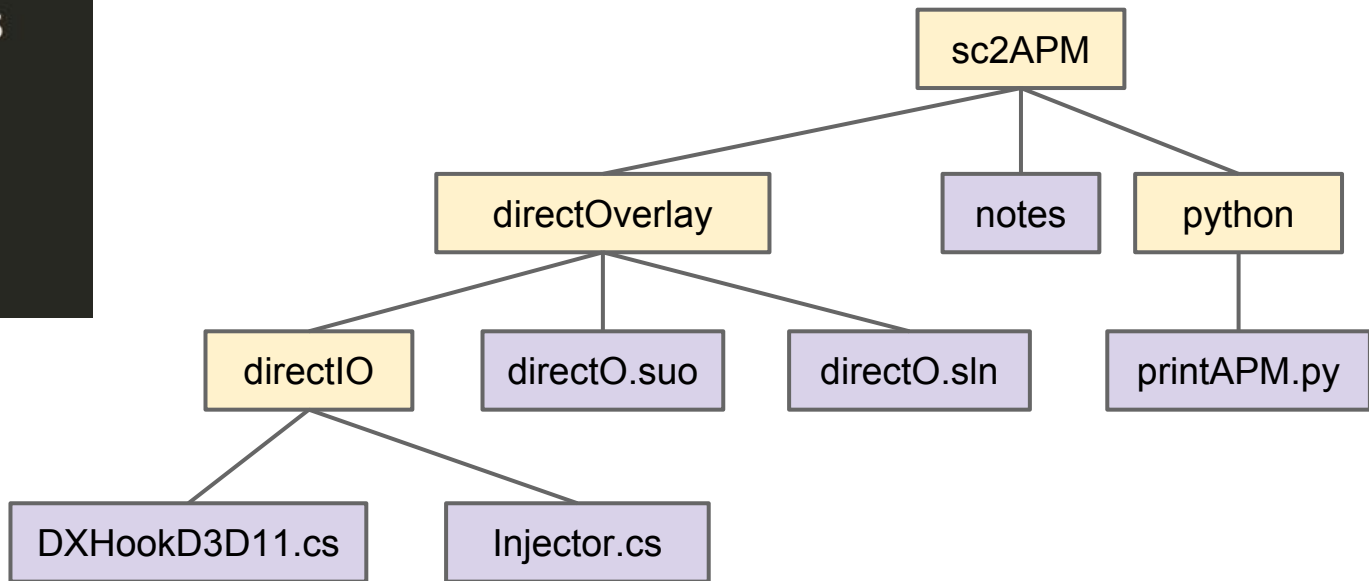
- 4 7 8 5 2 9 6 3 1



# What Good Are All These Traversals?

Example: Preorder Traversal for printing directory listing:

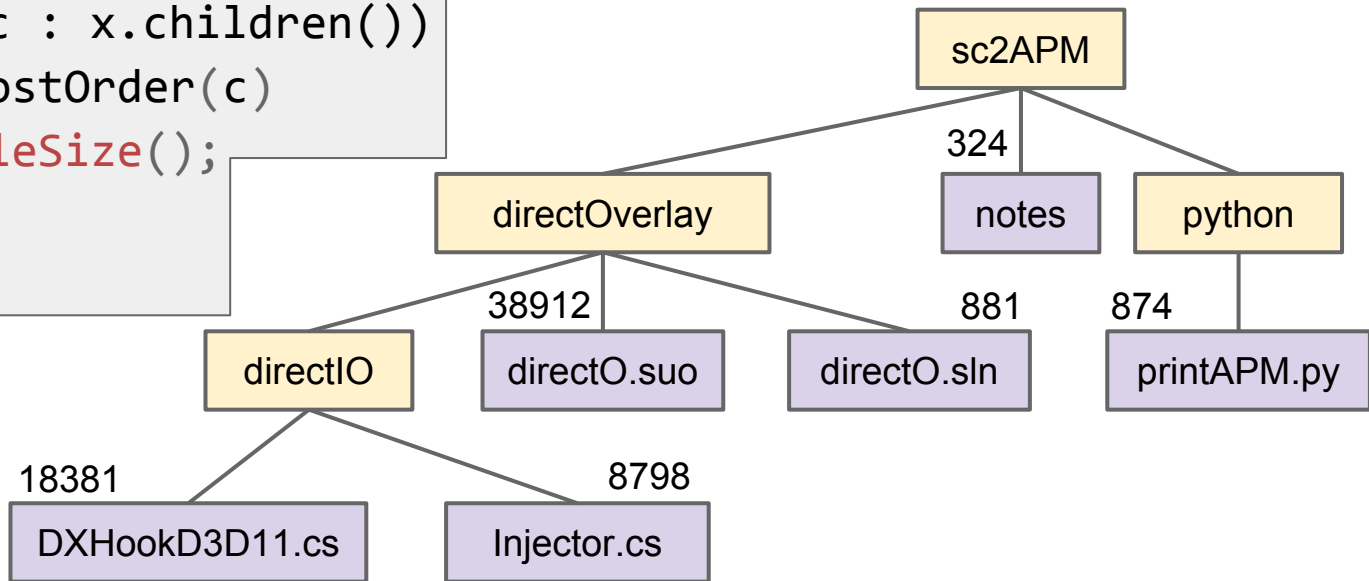
```
sc2APM/  
  directOverlay/  
    directIO/  
      DXHookD3D11.cs  
      Injector.cs  
    directO.suo  
    directO.sln  
  notes  
  python/  
    printAPM.py
```



# What Good Are All These Traversals?

Example: Postorder Traversal for gathering file sizes.

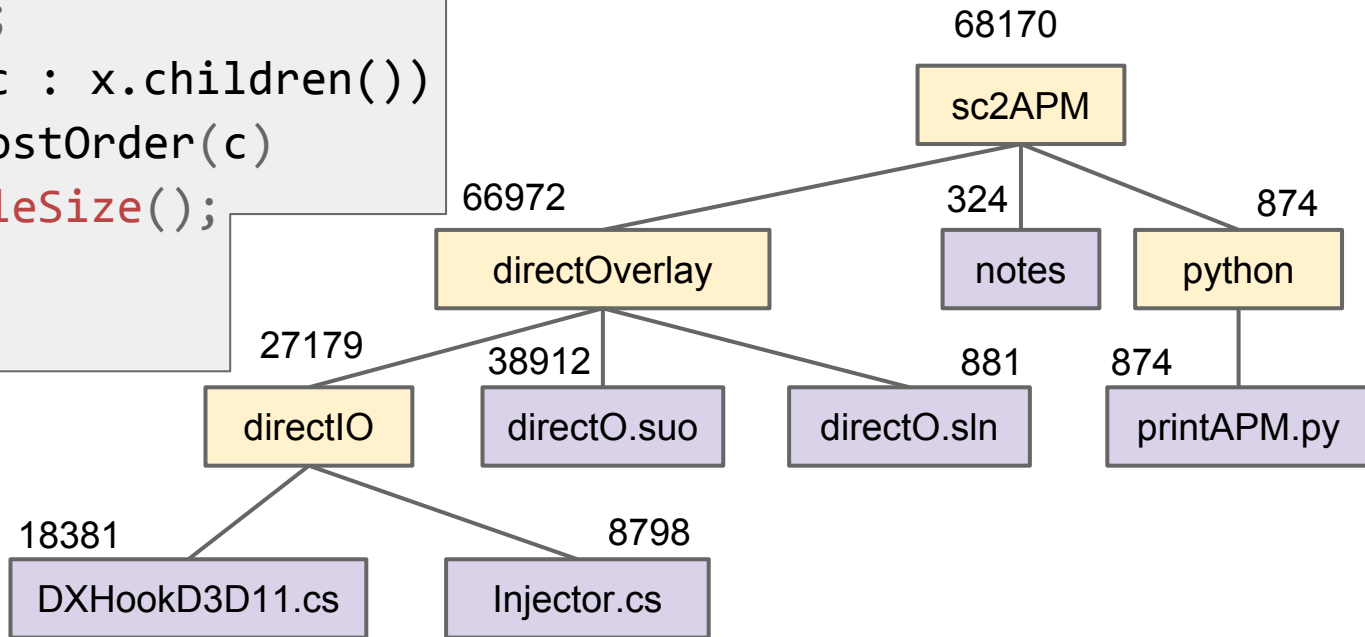
```
postOrder(BSTNode x) {  
    if (x == null) return 0;  
    int total = 0;  
    for (BSTNode c : x.children())  
        total += postOrder(c)  
    total += x.fileSize();  
    return total;  
}
```



# What Good Are All These Traversals?

Example: Postorder Traversal for gathering file sizes.

```
postOrder(BSTNode x) {  
    if (x == null) return 0;  
    int total = 0;  
    for (BSTNode c : x.children())  
        total += postOrder(c)  
    total += x.fileSize();  
    return total;  
}
```



## Visitor Pattern (Patterns)

---

When writing general tree traversal code. Avoid rewriting traversal for every task of interest (print, sum file sizes, etc.) by using the Visitor pattern.

```
void preorderTraverse(Tree<Label> T, Action<Label> whatToDo) {  
    if (T == null) { return; }  
    whatToDo.visit(T); /* before we hard coded a print */  
    preorderTraverse(T.left, whatToDo);  
    preorderTraverse(T.right, whatToDo);  
}
```

```
interface Action<Label> {  
    void visit(Tree<Label> T);  
}
```

```
class FindPig implements Action<String> {  
    boolean found = false;  
    @Override  
    void visit(Tree<String> T) {  
        if ("pig".equals(T.label))  
            { found = true; }  
    }  
}
```

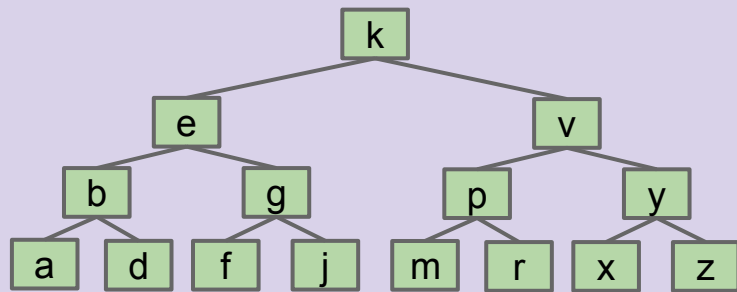
```
preorderTraverse(someTree, new FindPig());
```

The real visitor pattern is [more complex](#).

## Preorder Traversal Runtime: <http://yellkey.com/most>

What is the runtime of a preorder traversal in terms of  $N$ , the number of nodes? (in code below, assume the visit action takes constant time)

1.  $\Theta(1)$
2.  $\Theta(\log N)$
3.  $\Theta(N)$
4.  $\Theta(N \log N)$
5.  $\Theta(2^N)$



```
void preorderTraverse(Tree<Label> T, Action<Label> whatToDo) {  
    if (T == null) { return; }  
    whatToDo.visit(T);  
    preorderTraverse(T.left, whatToDo);  
    preorderTraverse(T.right, whatToDo);  
}
```

## Preorder Traversal Runtime

---

What is the runtime of a preorder traversal in terms of  $N$ , the number of nodes? (in code below, assume the visit action takes constant time)

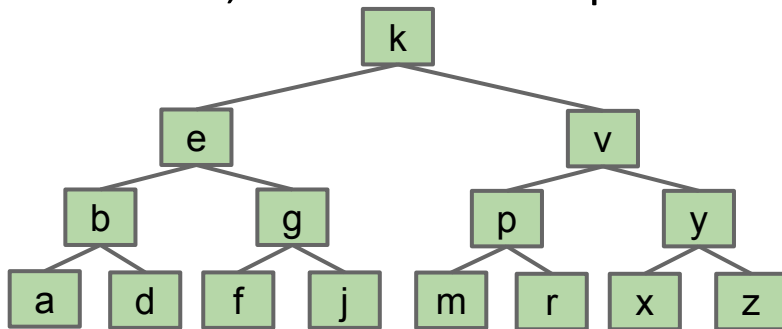
**3.  $\Theta(N)$  : Every node visited exactly once. Constant work per visit.**

Runtime is exponential in height of the tree, not number of items.

- $\Theta(2^H)$ , but  $H = \Theta(\log N)$
- This is not a proof of runtime, but rather a response to a possible objection.

$N = 15$

$H = 3$





# Level Order Traversal

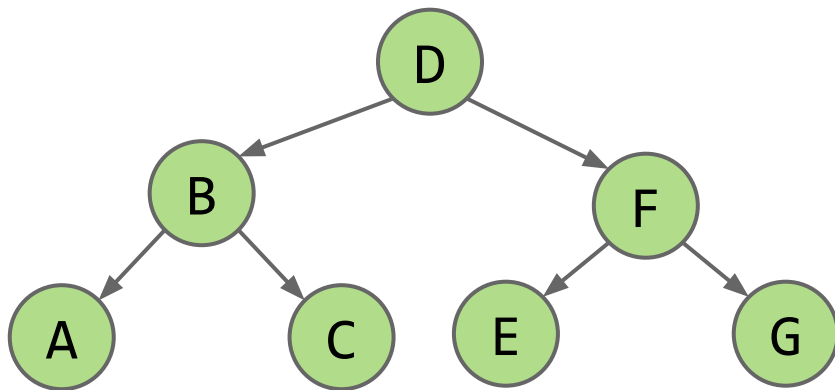
# Tree Traversal: Level Order Traversal

---

The Level Order Traversal is the result of reading the nodes “like a book”, one level at a time.

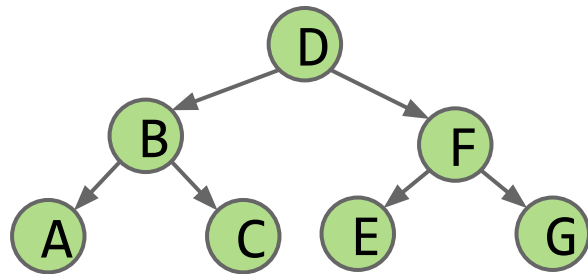
How would we implement a level order traversal?

- Level order: D B F A C E G
- Goal: Visit nodes on 0th level, then 1st level, then 2nd level, etc.



# Level-Order Traversal through Iterative Deepening

```
public void levelOrder(Tree T, Action toDo) {  
    for (int i = 0; i < T.height(); i += 1) {  
        visitLevel(T, i, toDo);  
    }  
}
```



Run visitLevel H times,  
one for each level.

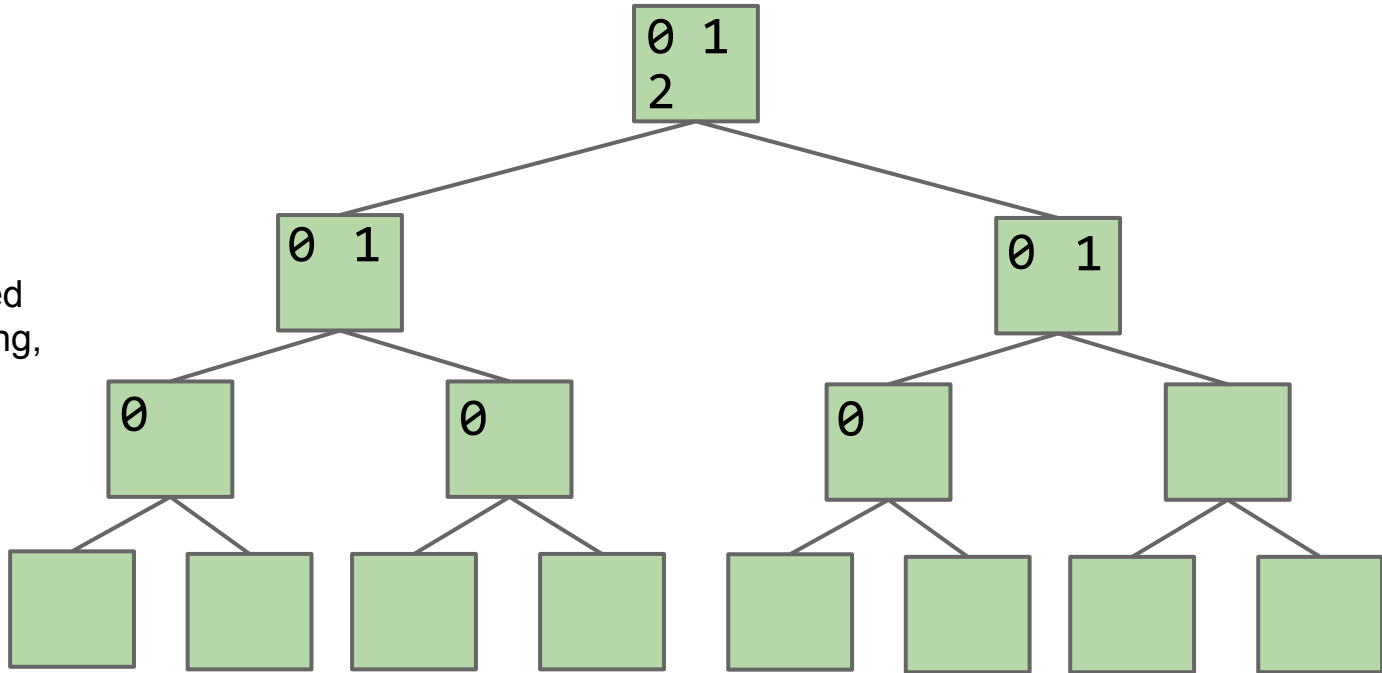
```
public void visitLevel(Tree T, int level, Action toDo) {  
    if (T == null)  
        { return; }  
    if (lev == 0)  
        { toDo.visit(T.key); }  
    else {  
        visitLevel(T.left(), lev - 1, toDo);  
        visitLevel(T.right(), lev - 1, toDo);  
    }  
}
```

The strategy described on  
this slide is called “Iterative  
Deepening”.

# Level-Order Traversal through Iterative Deepening

---

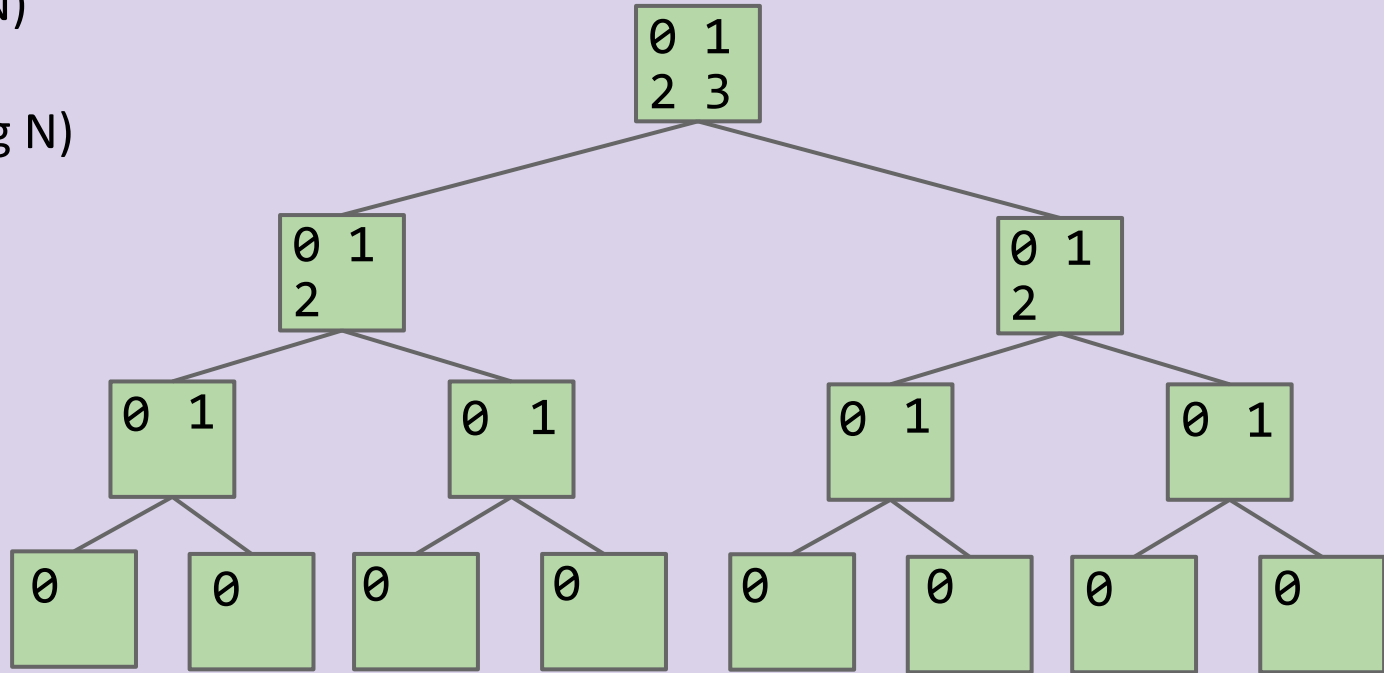
Partially completed  
Iterative Deepening,



## Iterative Deepening Runtime: <http://yellkey.com/nor>

What is the runtime to complete iterative deepening on a **complete** tree (as shown below) as a function of node count  $N$ ?

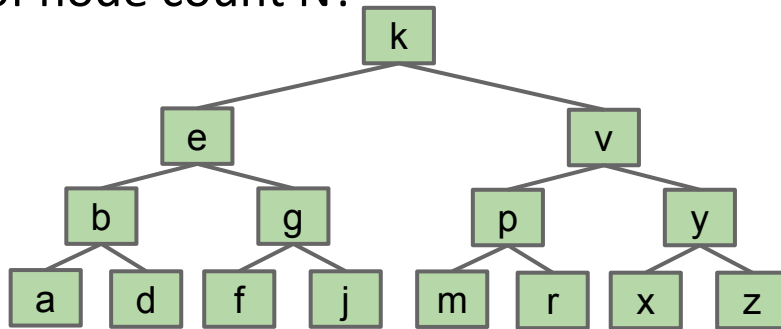
1.  $\Theta(\log N)$
2.  $\Theta(N)$
3.  $\Theta(N \log N)$
4.  $\Theta(N^2)$
5.  $\Theta(2^N)$



# Preorder Traversal and Prefix Expressions

What is the runtime to complete iterative deepening on a **complete** tree (as shown below) as a function of node count  $N$ ?

1.  $\Theta(N)$



Top level considered: 1

Then top two levels considered:  $1 + 2 = 3$

Then top three levels considered:  $1 + 2 + 4 = 7$

Then top four:  $1 + 2 + 4 + 8 = 15$

Top  $H$  levels:  $2^1 + 2^2 + 2^3 + \dots + 2^H - H = \Theta(N)$

Note: Exact sum doesn't matter, the order of growth (and hence the pattern) is what is important.

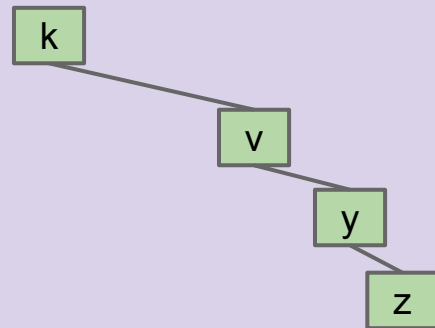
Interesting aside: Much harder to solve as “4 visits at level 0” then “6 visits at level 1”, etc.

## Iterative Deepening Runtime: <http://yellkey.com/work>

---

What is the runtime for iterative deepening on a “spindly” tree?

1.  $\Theta(\log N)$
2.  $\Theta(N)$
3.  $\Theta(N \log N)$
4.  $\Theta(N^2)$
5.  $\Theta(2^N)$



# Iterative Deepening Runtime

---

What is the runtime for iterative deepening on a “spindly” tree?

4.  $\Theta(N^2)$

Top level considered: 1

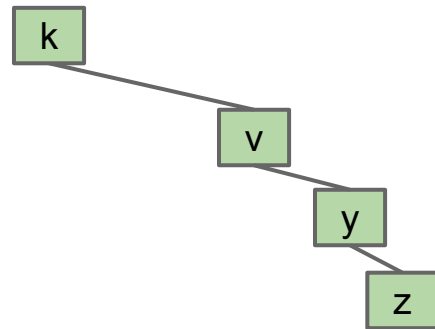
Then top two levels:  $1 + 1 = 2$

Then top three levels:  $1 + 1 + 1 = 3$

Top  $H$  levels:  $H$

Total:  $1 + 2 + 3 + \dots + H = H^2$

$H = N - 1$ , so  $\Theta(N^2)$

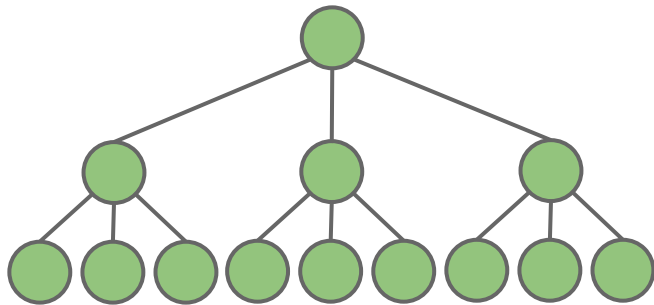




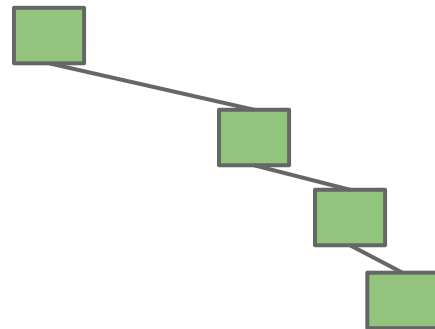
# Tree Height

---

For algorithms whose runtime depends on height, difference between bushy tree and spindly tree can be huge!



$$H = \Theta(\log(N))$$



$$H = \Theta(N)$$

Iterative deepening runtimes:  $\Theta(N)$  vs.  $\Theta(N^2)$

- Note: No simple mapping from height to runtime.
- Extra for experts: Write a better level order Traversal algorithm.

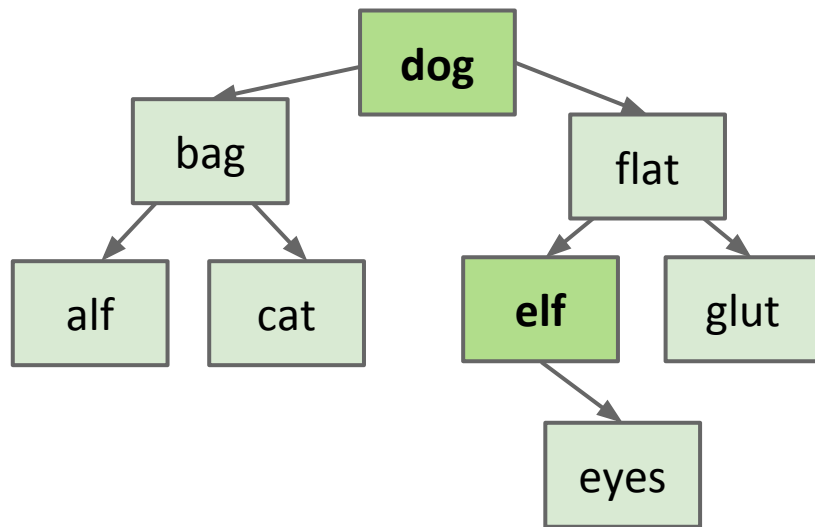
# Range Finding

# Geometric Search

---

Suppose we want an operation that returns all items in a range:

- `public Set<Label> findInRange(Tree T, Label min, Label max)`



Example:

- `findInRange(T, "dog", "elves")`
- Should return:
  - `{"dog", "elf"}`

# Geometric Search

---

Easy approach, just do a traversal of the whole tree, and use visitor pattern to collect matching items.

```
class rangeFind implements Action<String> {
    private Label min, max;
    public Set<Label> inRange;
    public rangeFind(Label min, Label max) {
        this.min = min; this.max = max;
        inRange = new HashSet<Label>();
    }

    void action(Tree<Label> T) {
        if (T.label ≤ max && T.label ≥ min) {
            inRange.add(T.label);
        }
    }
}
```

Runtime is  $\Theta(N)$

# Geometric Search

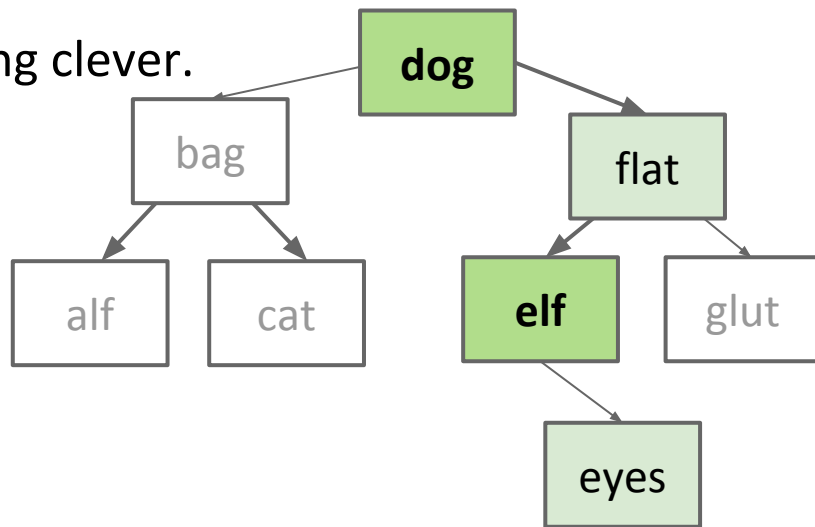
Suppose we want an operation that returns all items in a range:

- **public** Set<Label> findInRange(Tree T, Label min, Label max)

Can avoid need to traverse entire tree by being clever.

Example:

- findInRange(T, “dog”, “elves”)
- No need to look:
  - Left of dog.
  - Right of flat.



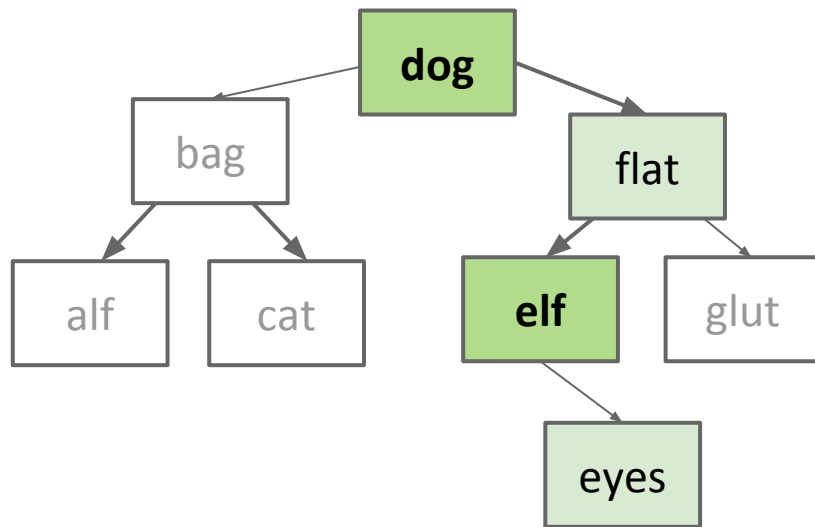
Nodes inspected: dog, flat, elf, eyes  
Nodes matching: dog, elf

## Pruning and findInRange Runtime

Suppose we want an operation that returns all items in a range:

- **public** Set<Label> findInRange(Tree T, Label min, Label max)

**Pruning:** Restricting our search to only nodes that might contain the answers we seek.



Nodes inspected: dog, flat, elf, eyes  
Nodes matching: dog, elf

## Pruning and findInRange Runtime

Suppose we want an operation that returns all items in a range:

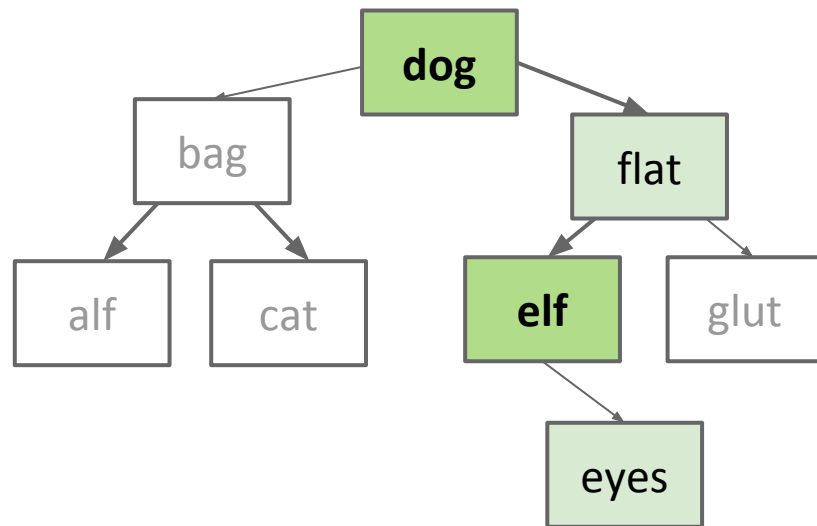
- **public** Set<Label> findInRange(Tree T, Label min, Label max)

**Pruning:** Restricting our search to only nodes that might contain the answers we seek.

Runtime for our search:  $\Theta(\log N + R)$

- N: Total number of items in tree.
- R: Number of matches.

See study guide A-level problems for proof.



Nodes inspected: dog, flat, elf, eyes  
Nodes matching: dog, elf

# Spatial Trees



## 2D Range Finding

---

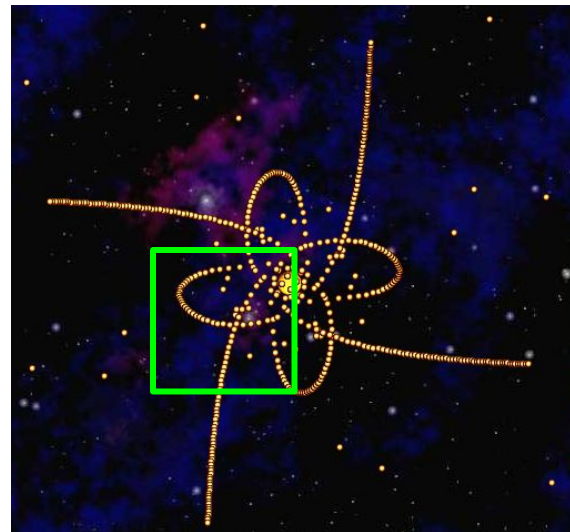
Suppose we want to do range finding on Planets in space.

- Query: How many objects are in the highlighted rectangle?

Could iterate through all objects in  $\Theta(N)$  time.

- But could we do some sort of tree + pruning?

Pruning implies we need some kind of tree, but ...



# Building Trees of Two Dimensional Data

So far, we've only considered one dimensional data.

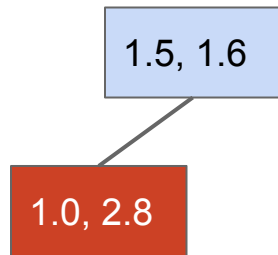
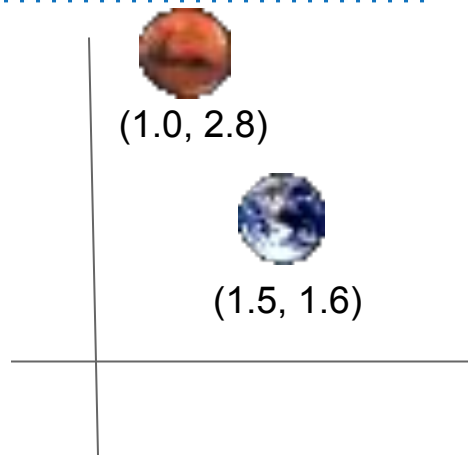
- There exists a total order!
  - $5 < 10$
  - "alf" < "elf"

Some data is two dimensional, e.g. the location of Planets.

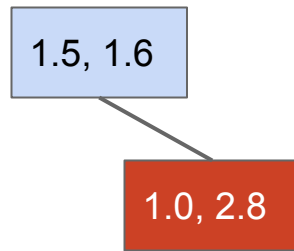
- earth.xPos = 1.5, earth.yPos = 1.6
- mars.xPos = 1.0, mars.yPos = 2.8

If we're comparing by location:

- In xPos, Mars < Earth
- In yPos, Mars > Earth



X-Based Tree



Y-Based Tree

# Handling Multidimensional Data: Quadrees

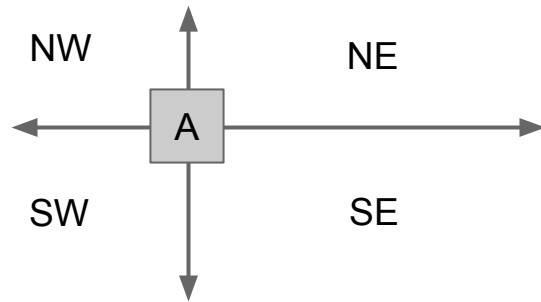
---

Quadrees:

- Divide and conquer by splitting 2D space into four quadrants.
  - Store items into appropriate quadrant.
  - Repeat recursively if more than one item in a quadrant.

Definition, quadtree is either:

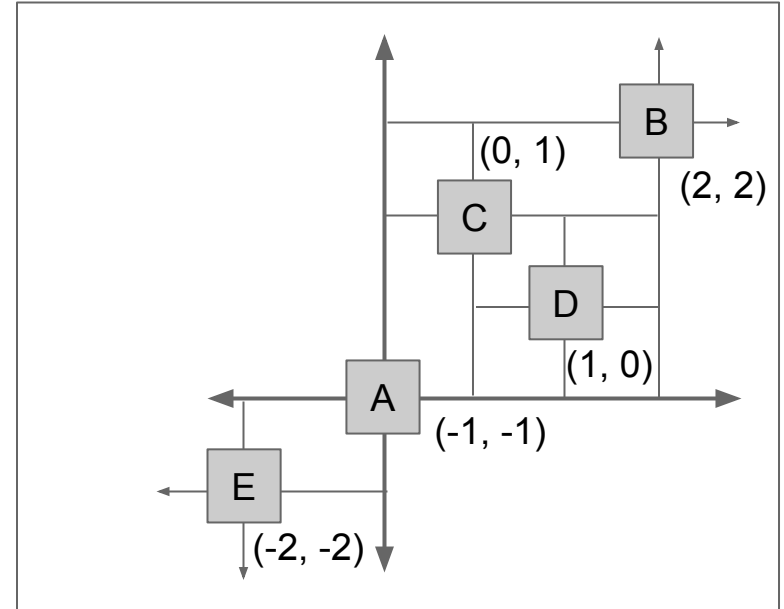
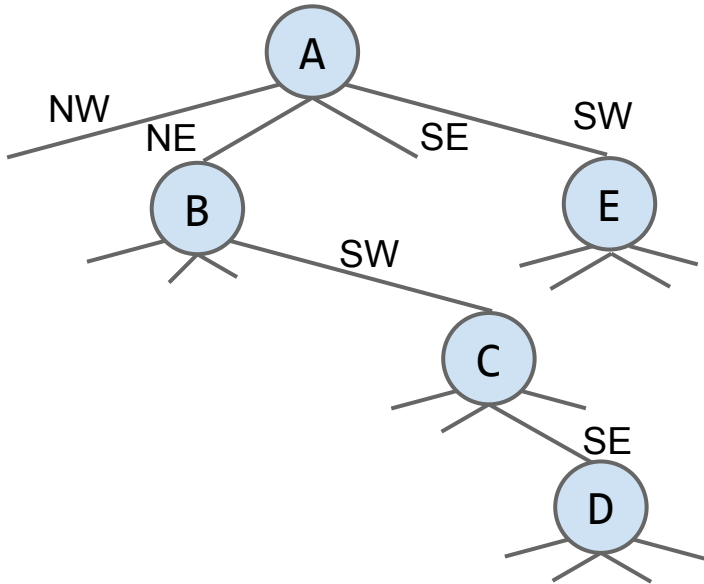
- Empty
- A 'root' item at some position  $(x, y)$  AND four quadrees that are northwest, northeast, southwest, southeast of  $(x, y)$
- Use TWO compares to decide which direction to go.



# Quadtree Demo

Below: Quadtree Representation of 5 objects in 2D space.

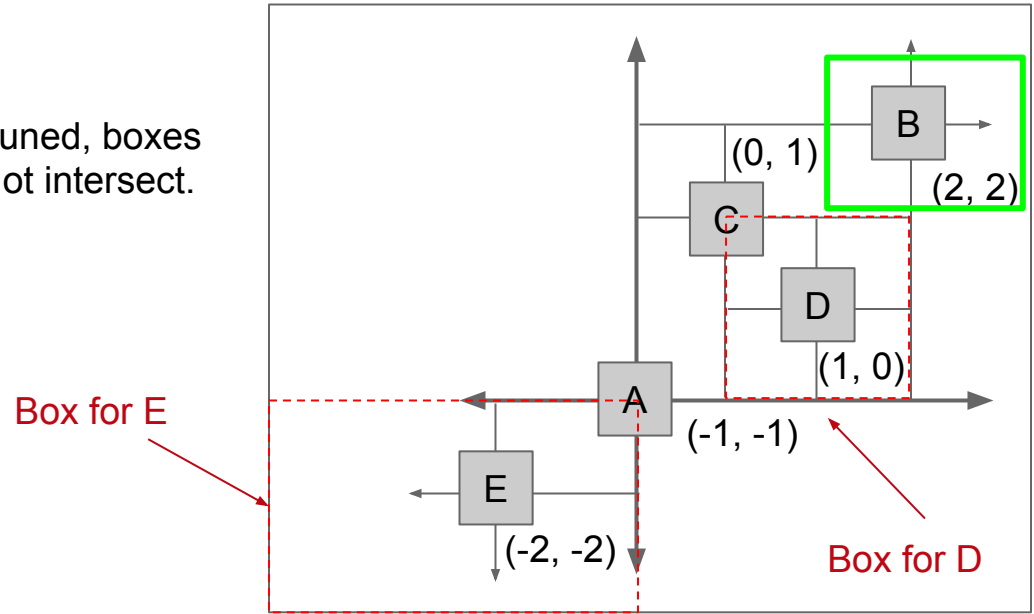
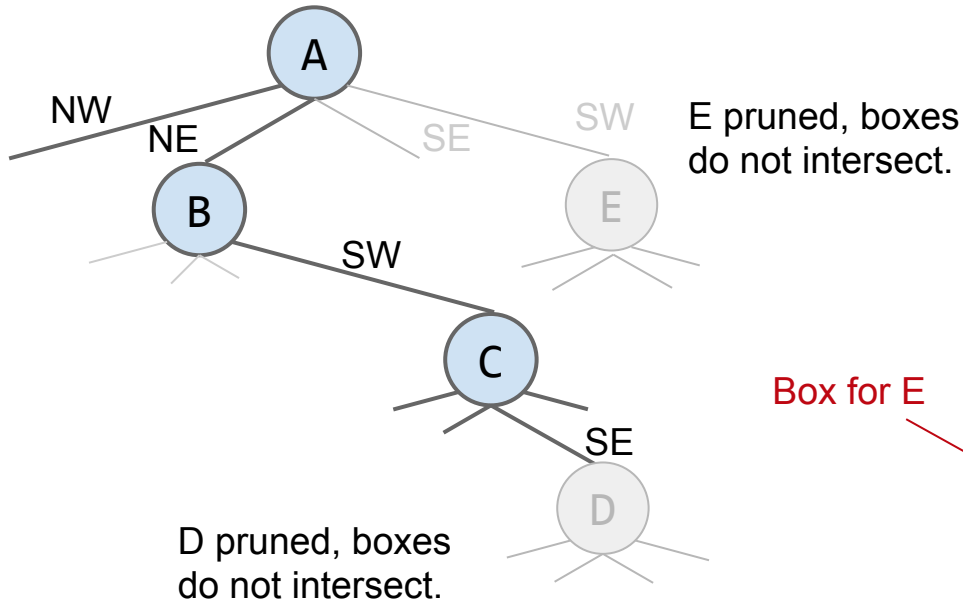
- Demo: [Link](#)



# Quadtree Demo

Quadtrees allow us to prune when performing a rectangle search.

- Basic rule: Prune a branch if the search rectangle doesn't overlap a quadrant of potential interest.



Only item that intersects box is B.

# Optional: Tree Iterators

# Iterators

---

Suppose we want to iterate through a tree using the : operator.

How can we adapt our traversal code to implement next() and hasNext()?

```
void preorderTraverse (Tree<Label> T, Action<Label> whatToDo)
{
    if (T != null) {
        whatToDo.action (T);
        for (int i = 0; i < T.numChildren (); i += 1)
            preorderTraverse (T.child (i), whatToDo);
    }
}
```

## Iteration: The Obvious Way

---

One approach: Create an action class that puts visited item in a list.

```
public class ListBuilder<Label> implements Action<Label> {
    public List<Label> L = new ArrayList<Label>();
    public void action (Tree<Label> T) {
        L.add(T.label());
    }
    void preorderTraverse (Tree<Label> T, Action<Label> whatToDo)
    {
        if (T != null) {
            whatToDo.action (T);
            for (int i = 0; i < T.numChildren (); i += 1)
                preorderTraverse (T.child (i), whatToDo);
        }
    }
}
```



## Iteration: The Obvious Way

---

One approach: Create an action class that puts visited item in a list.

- iterator method creates such a list and returns an iterator to it.
- What's the downside of this solution?

```
public class ListBuilder<Label> implements Action<Label> {  
    public List<Label> L = new ArrayList<Label>();  
    public void action (Tree<Label> T) {  
        L.add(T.label());  
    }  
}
```


```
public Iterator<Label> jankyIterator(Tree<Label> T) {  
    ListBuilder<Label> lb = new ListBuilder<Label>();  
    T.preorderTraverse(T, lb);  
    return lb.L.iterator();  
}
```

## Iteration: Space-saving Approach

---

Tricky question: How could convert our recursive traversal into iterative code using a stack?

```
void preorderTraverse (Tree<Label> T, Action<Label> whatToDo)
{
    if (T != null) {
        whatToDo.action (T);
        for (int i = 0; i < T.numChildren (); i += 1)
            preorderTraverse (T.child (i), whatToDo);
    }
}
```



Observation: Each call to preorderTraverse is the equivalent of putting a call on the call stack.

## Iteration: Space-saving Approach

---

Tricky question: How could convert our recursive traversal into iterative code using a stack?

```
public void preorderTraverseIterative(Tree<Label> T, Action<Label> whatToDo)
{
    Stack<Tree<Label>> s = new Stack<Tree<Label>>();
    s.push(T);
    while (!s.isEmpty()) {
        Tree<Label> node = s.pop();
        if (node == null)
            continue;

        whatToDo.action (node);
        for (int i = node.numChildren()-1; i >= 0; i -= 1)
            s.push(node.child(i));
    }
}
```

## Iteration: Space-saving Approach

---

Use our stack-based approach, but use next() instead of looping.

```
private class preorderIterator implements Iterator<Label>{
    Stack<Tree<Label>> s = new Stack<Tree<Label>>();
    public preorderIterator() {
        s.push(Tree.this); /* new syntax, Tree.this is parent tree */
    }
    public boolean hasNext() {
        return (!s.isEmpty());
    }
    public Label next() {
        Tree<Label> node = s.pop();
        for (int i = node.numChildren()-1; i >= 0; i -= 1)
            s.push(node.child(i));
        return node.label;
    }
}
```

## Citations

---

Title figure: A thing I made (one of the first Java programs I wrote during my teaching career)

Pruning image:

[https://res.cloudinary.com/dc8hy36qb/image/upload/v1435213404/Fruit-Tree-Pruning-Methods\\_o7ieen\\_atkmmq.jpg](https://res.cloudinary.com/dc8hy36qb/image/upload/v1435213404/Fruit-Tree-Pruning-Methods_o7ieen_atkmmq.jpg)

Jonathan Shewchuk: Nice intuitive use cases for various traversals.