CSE 373

APRIL 14TH – TREES PT 2

ASSORTED MINUTIAE

- **HW3 Out last night**
  - No need to submit testing code
  - System.currentTimeMillis()
- **HW1 wrong submissions**
  - Grades posted by Sunday – Canvas announcement
- **Regrade requests**
  - https://catalyst.uw.edu/webq/survey/ejmcc/330190

TODAY'S LECTURE

- **Tree traversals**
  - Depth first search
  - Breadth first search
- **Tree properties**
  - Balance

TREE TRAVERSALS

- **What is the point of a traversal?**
  - Some way to get through elements of the tree
  - Useful for more than just trees

TREE TRAVERSALS

- **Array implementations**
  - Traversal is easy, search left-to-right
  - Traversal is complete, no element is missed
  - Doesn’t take advantage of heap property

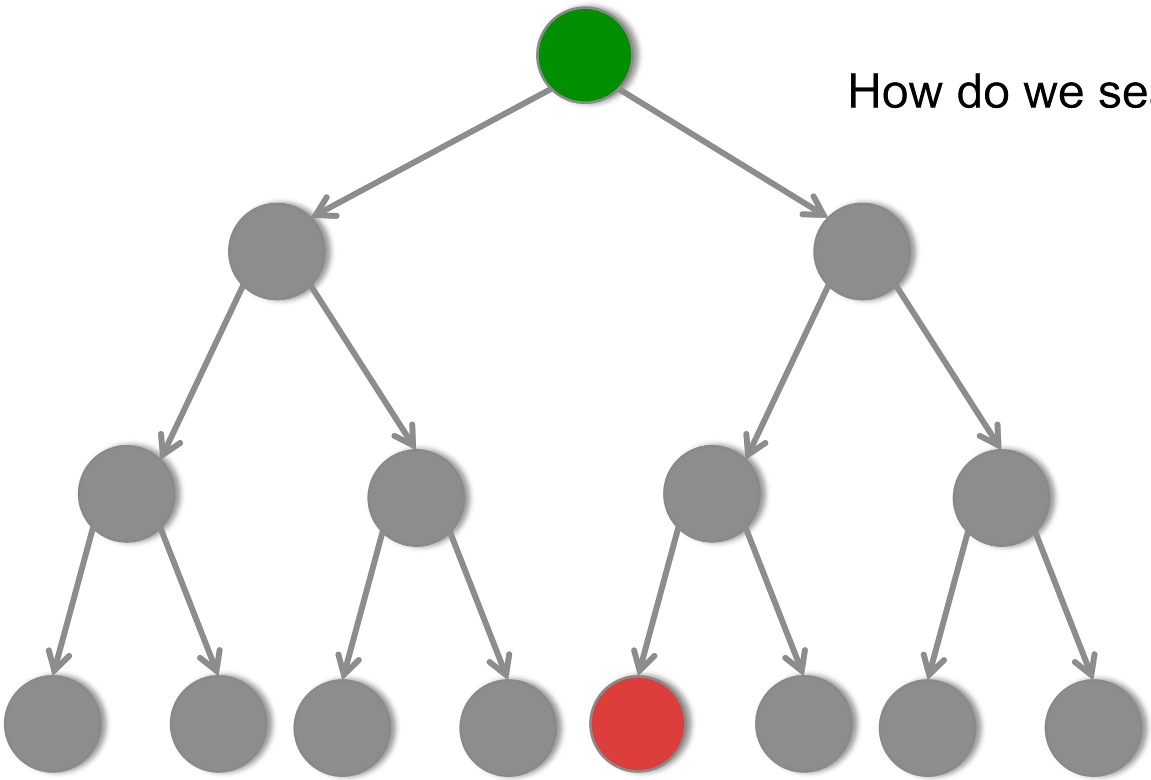
TREE TRAVERSALS

- **Node implementations**
  - Not as easy
  - No inherent ordering
  - Two main approaches:
    - Depth first search
    - Breadth first search

DEPTH FIRST SEARCH

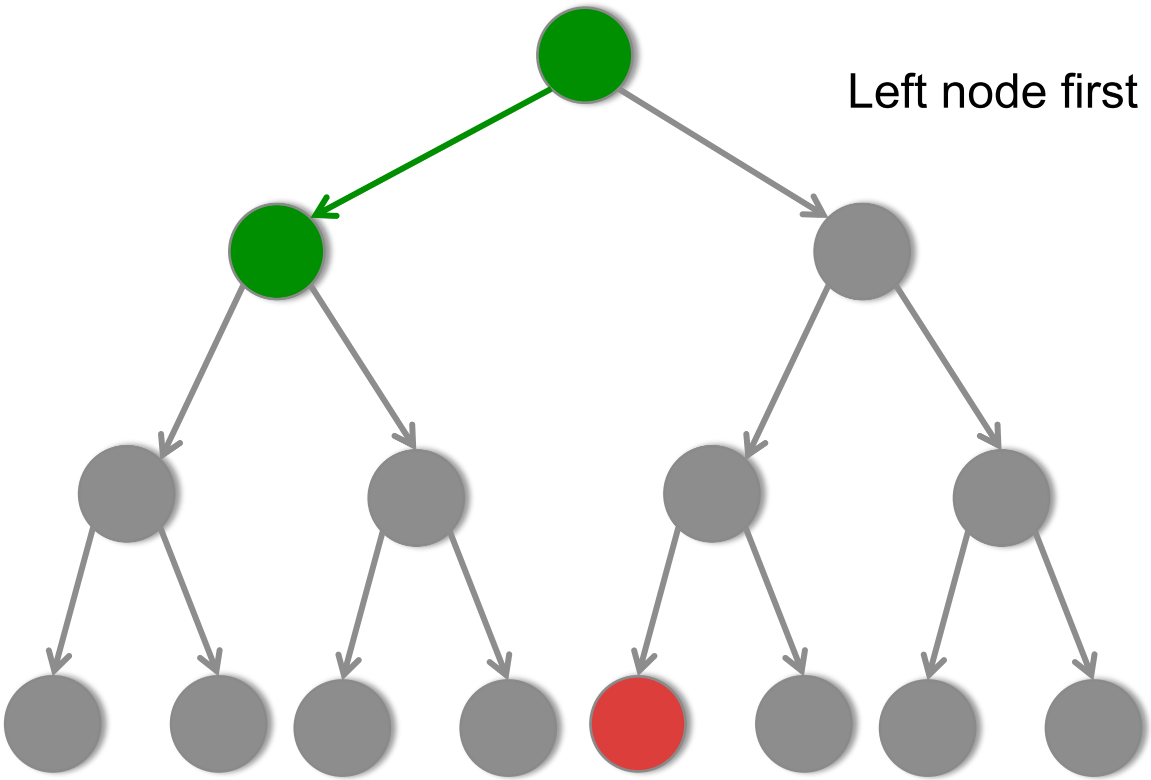
- All tree traversals start at the root
- As the name implies, traverse down the tree first.
- Left or right does not explicitly matter, but left usually comes first.

DEPTH FIRST SEARCH

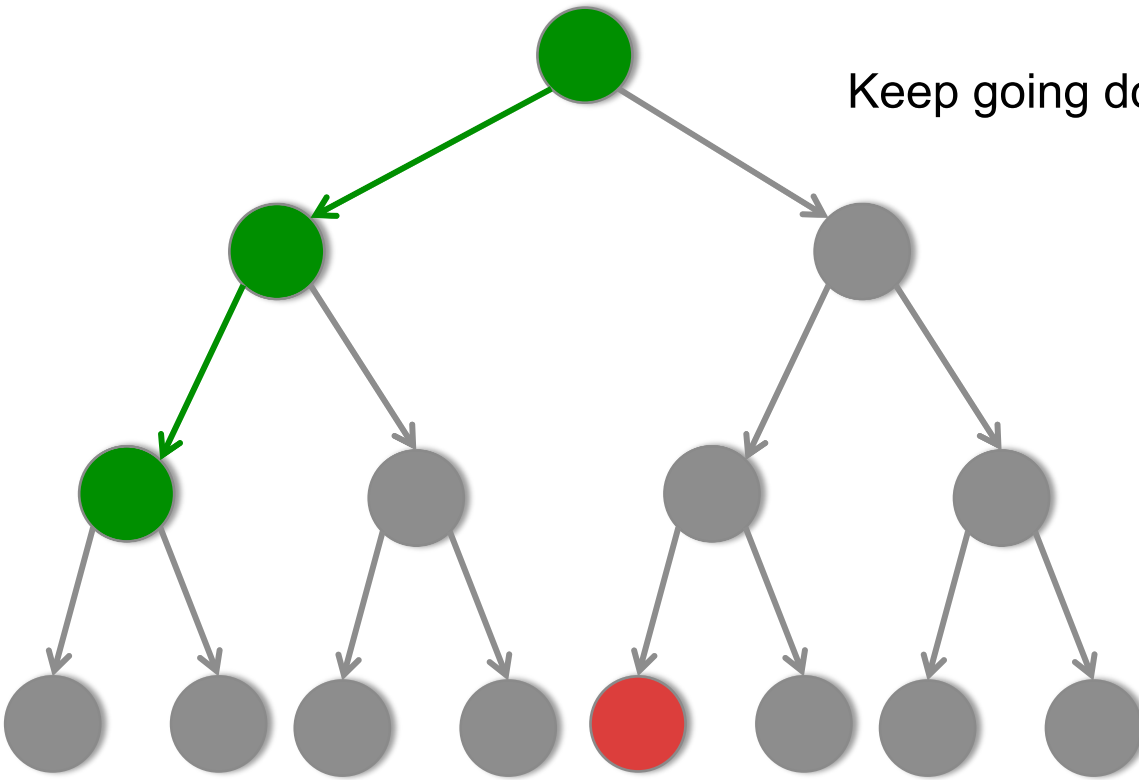


How do we search this tree?

DEPTH FIRST SEARCH

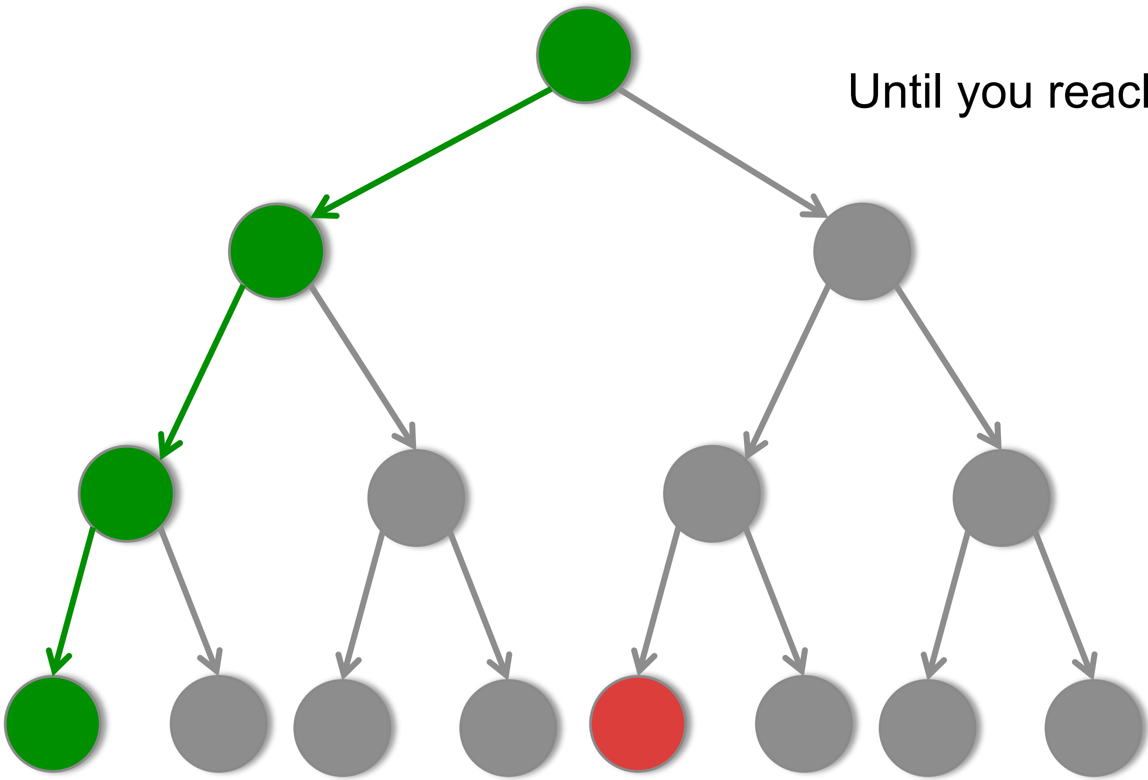


DEPTH FIRST SEARCH



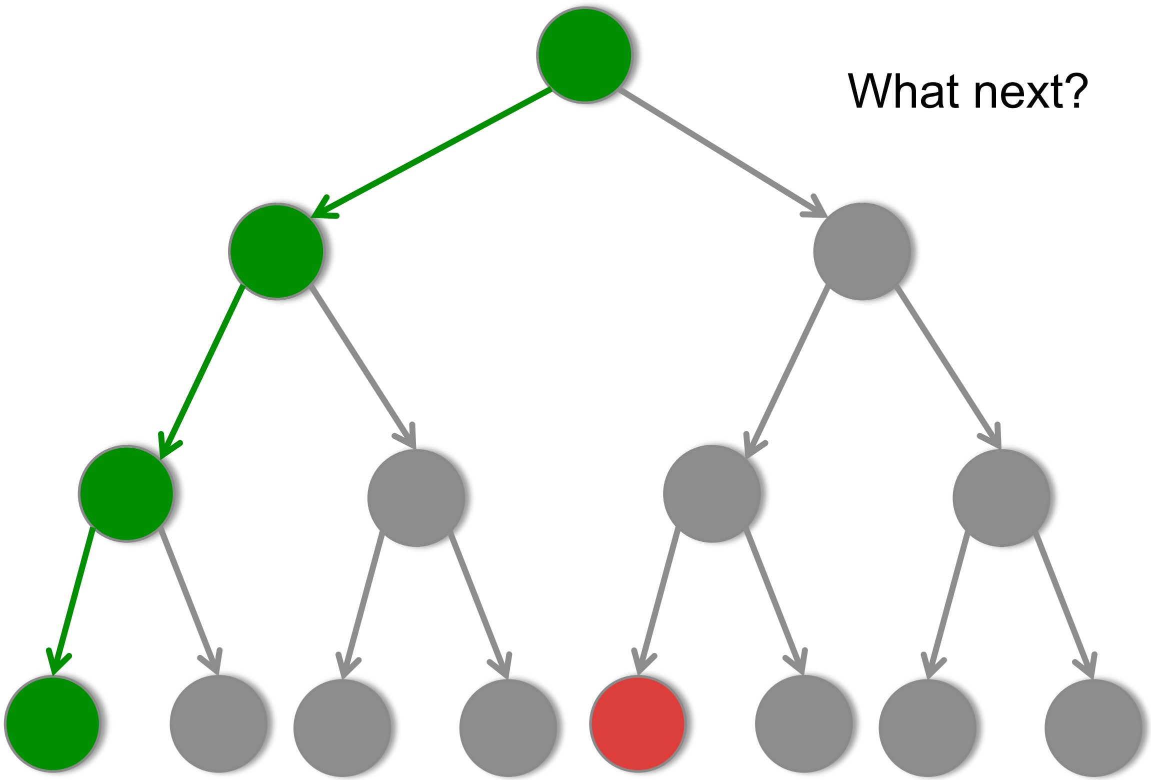
Keep going down the left nodes

DEPTH FIRST SEARCH



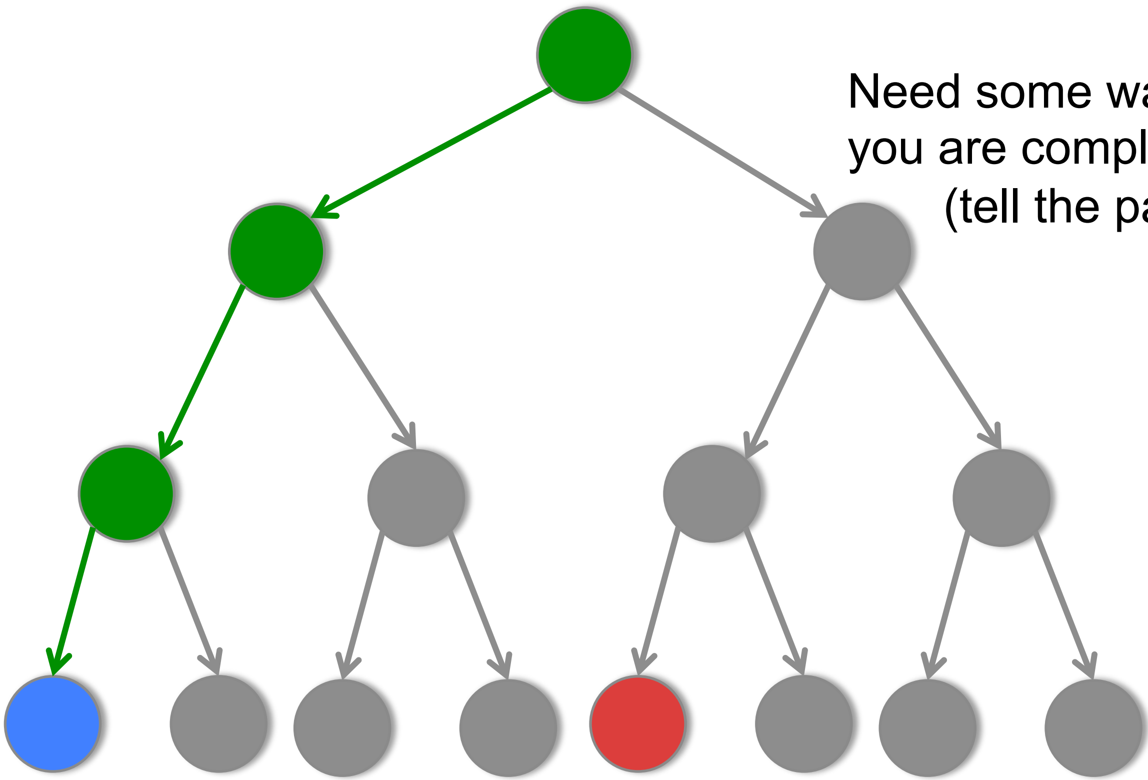
Until you reach the bottom

DEPTH FIRST SEARCH



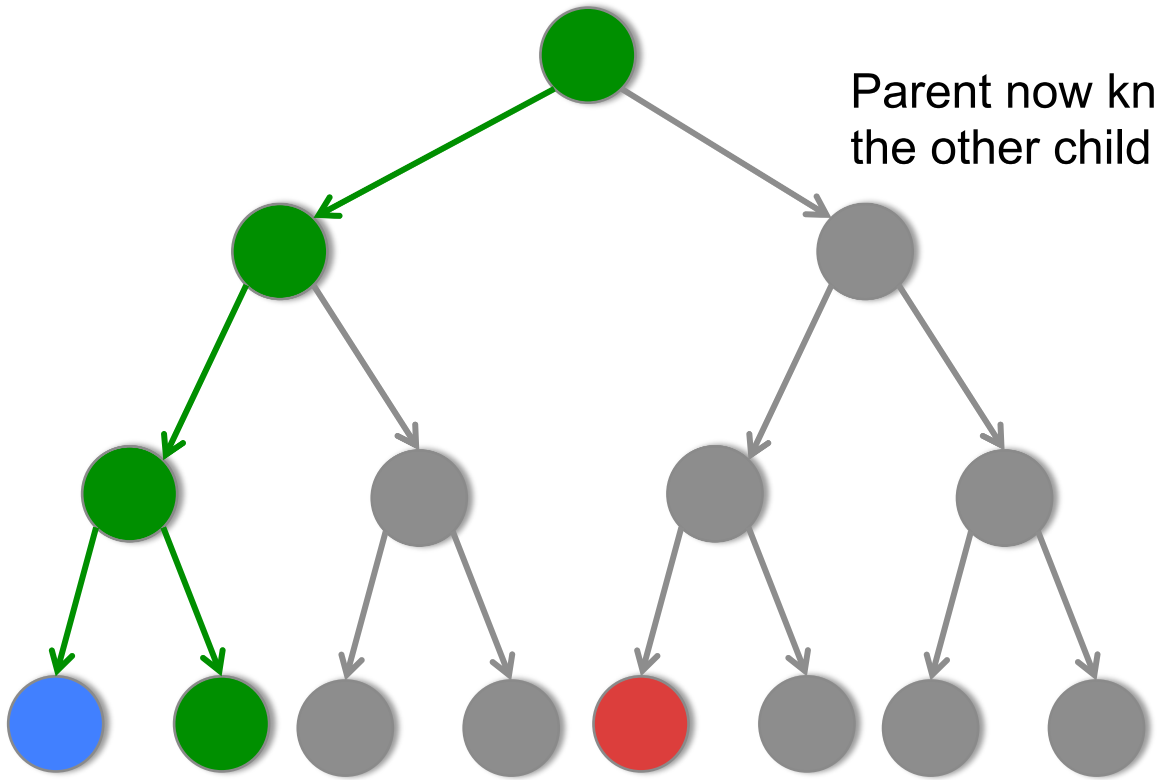
What next?

DEPTH FIRST SEARCH



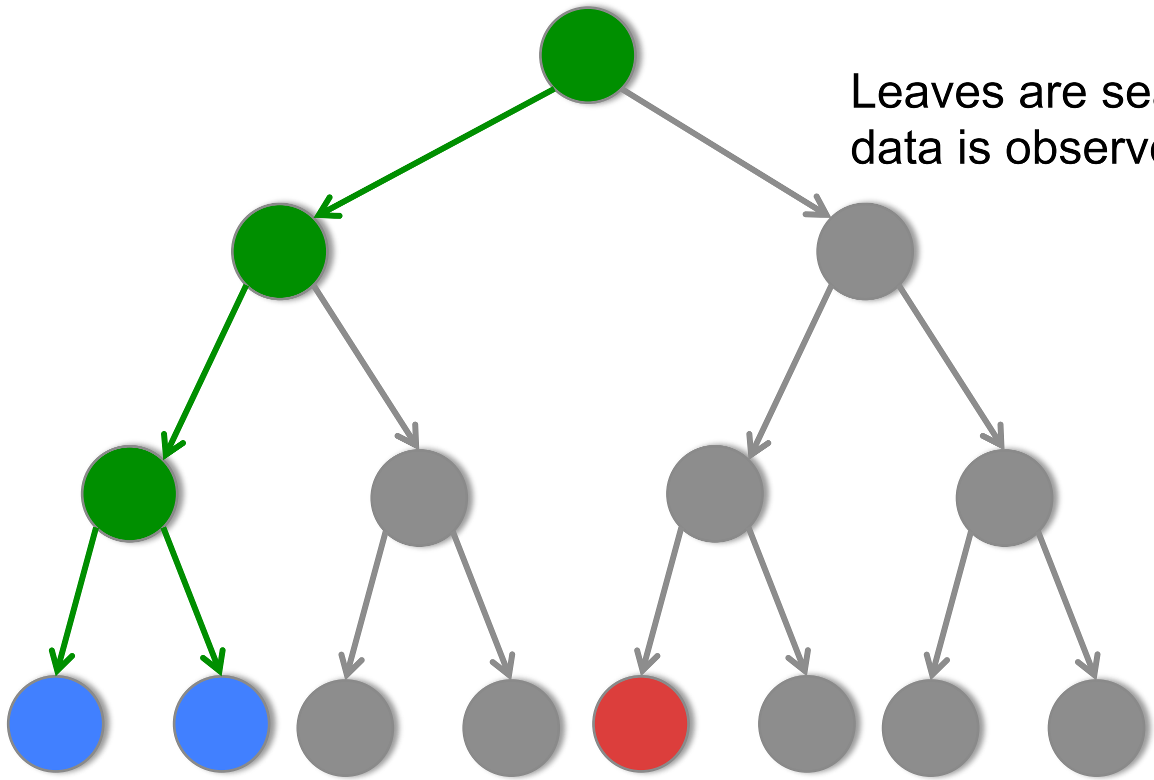
Need some way to indicate that you are completely searched
(tell the parent)

DEPTH FIRST SEARCH



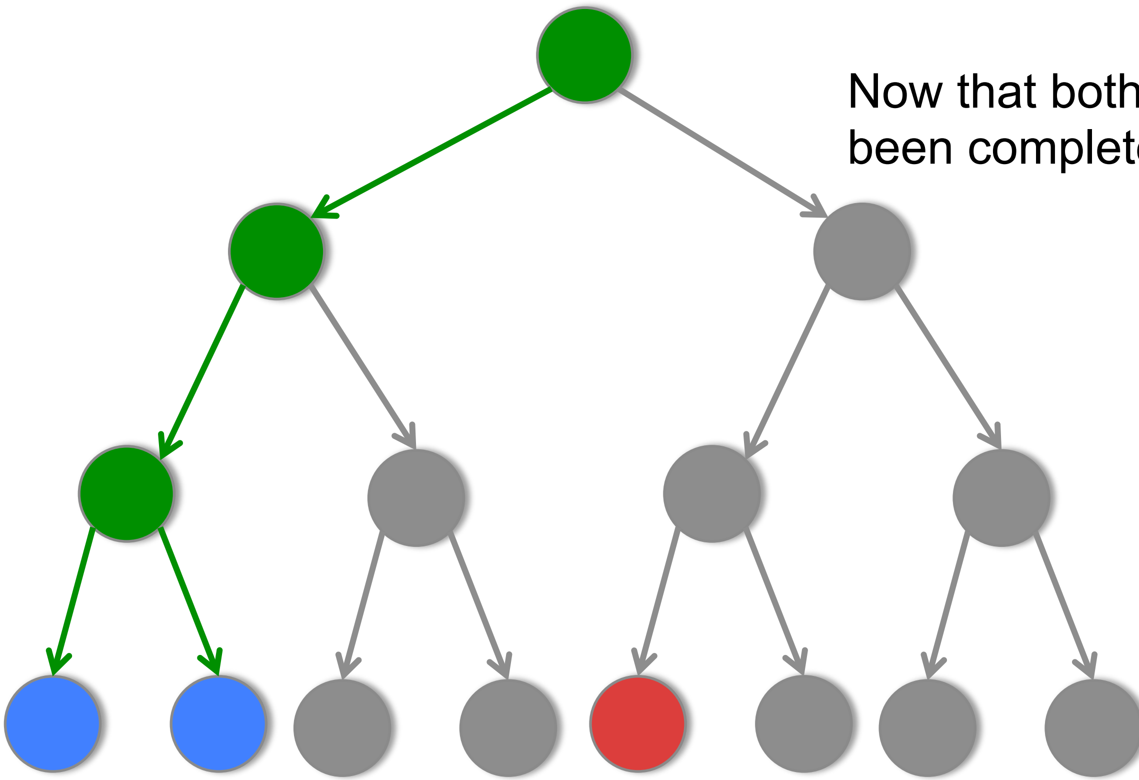
Parent now knows it is can search the other child

DEPTH FIRST SEARCH



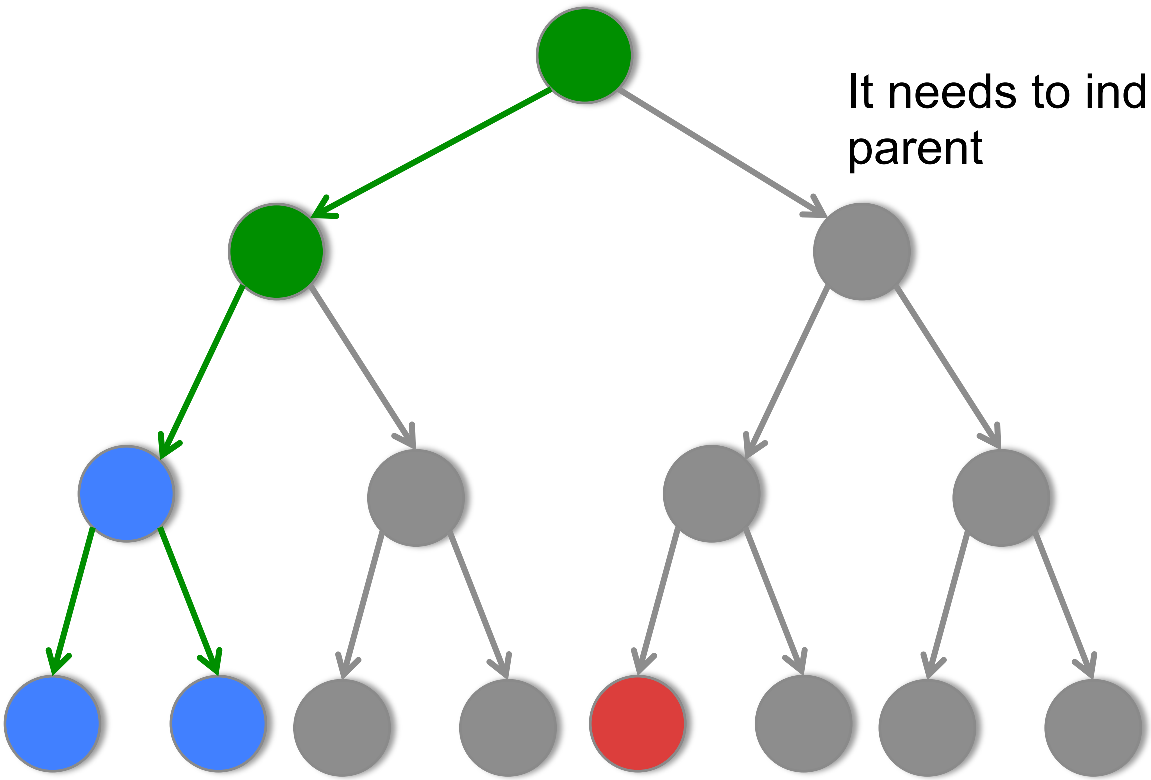
Leaves are searched when their data is observed

DEPTH FIRST SEARCH



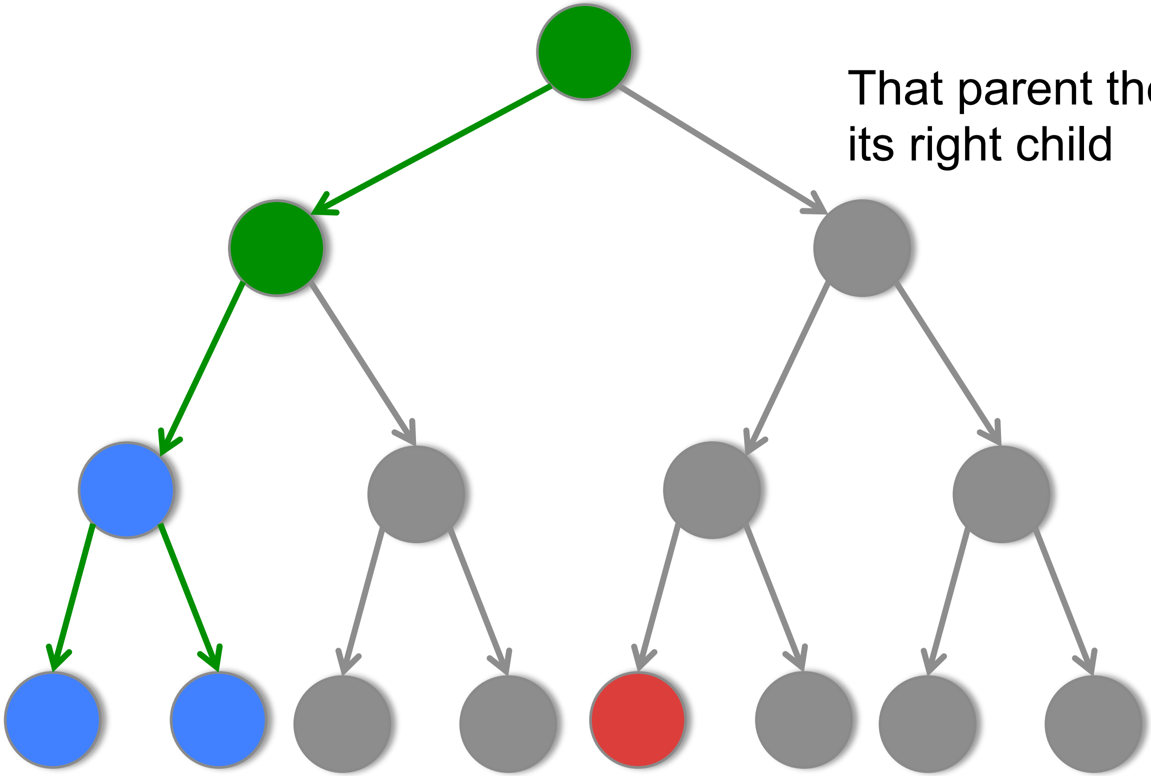
Now that both of its children have been completely searched

DEPTH FIRST SEARCH



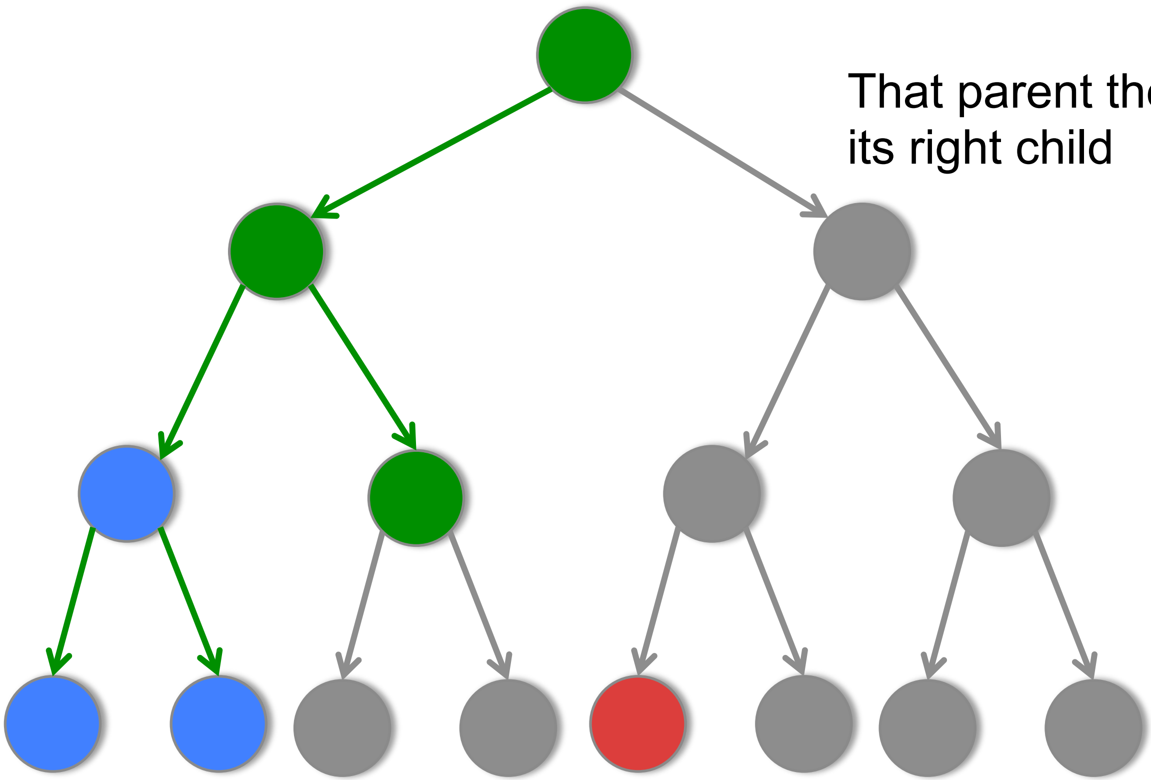
It needs to indicate that to its parent

DEPTH FIRST SEARCH



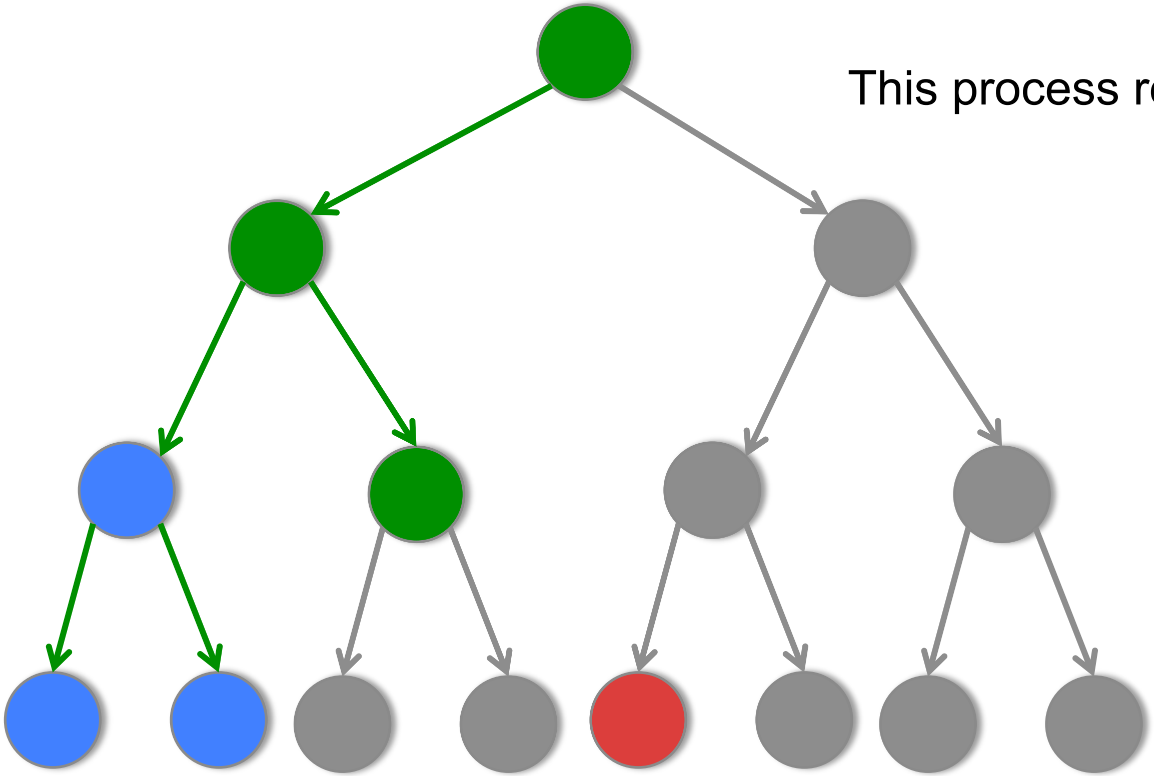
That parent then knows to search its right child

DEPTH FIRST SEARCH



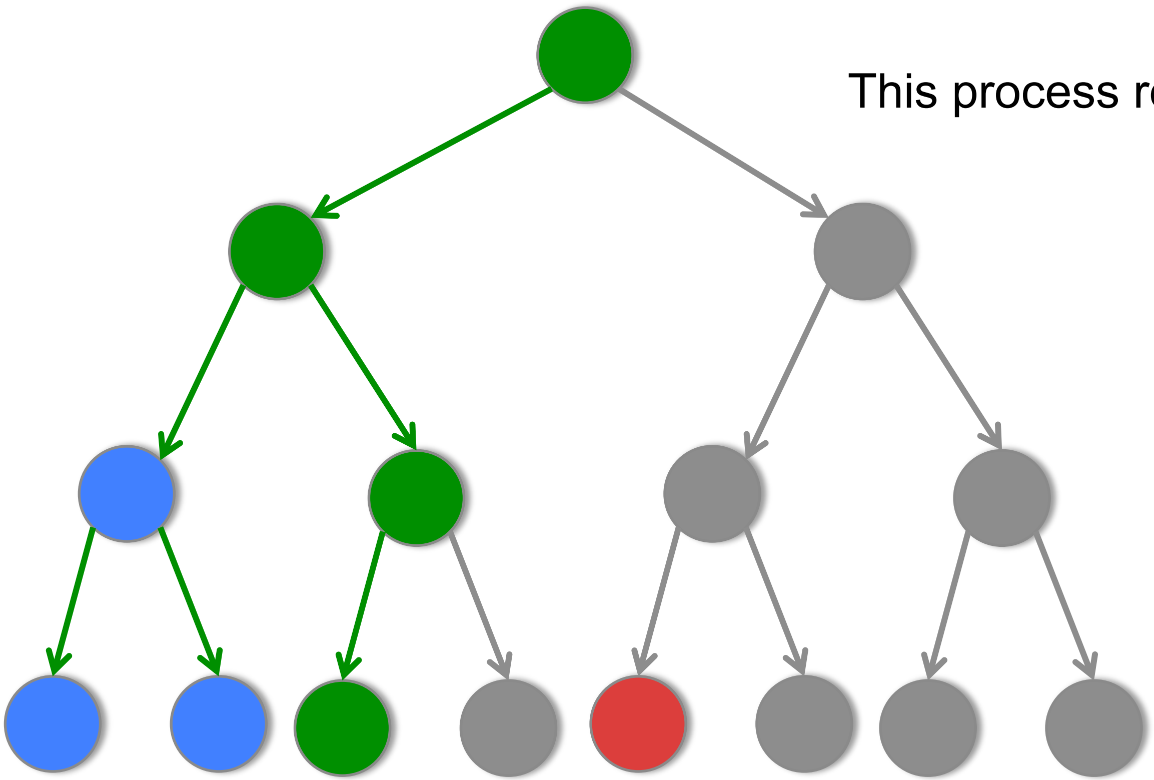
That parent then knows to search its right child

DEPTH FIRST SEARCH



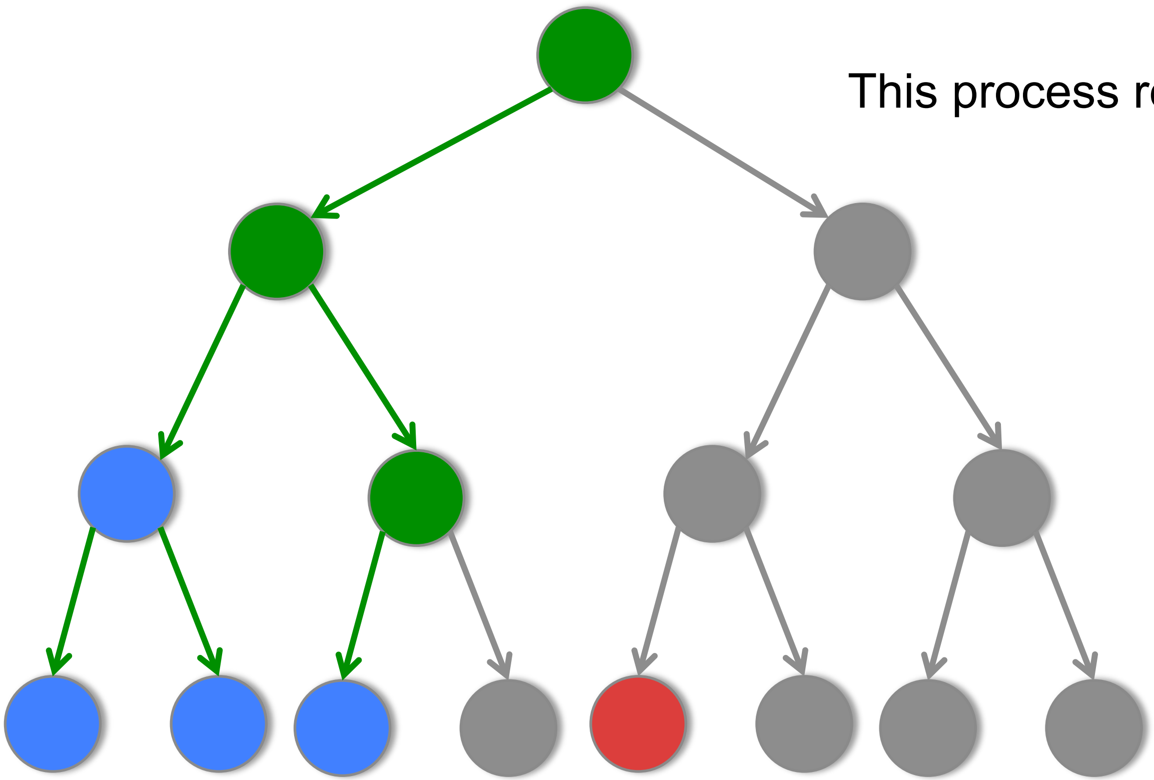
This process repeats

DEPTH FIRST SEARCH



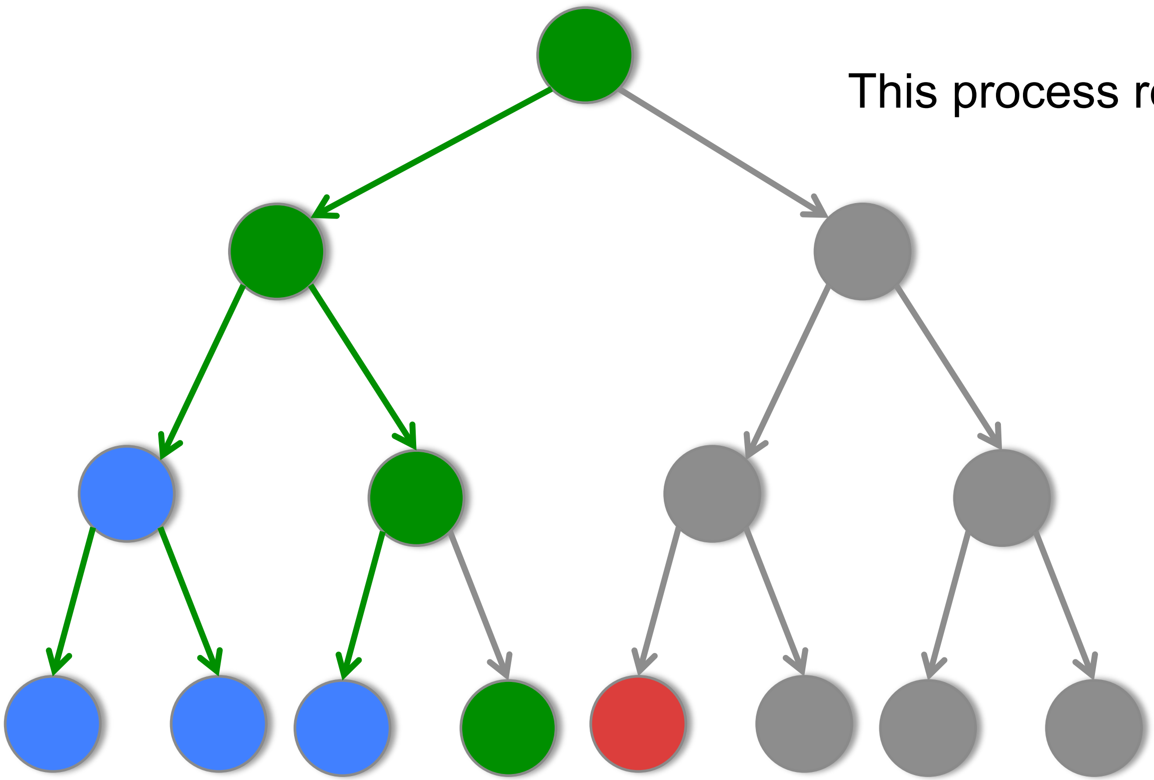
This process repeats

DEPTH FIRST SEARCH



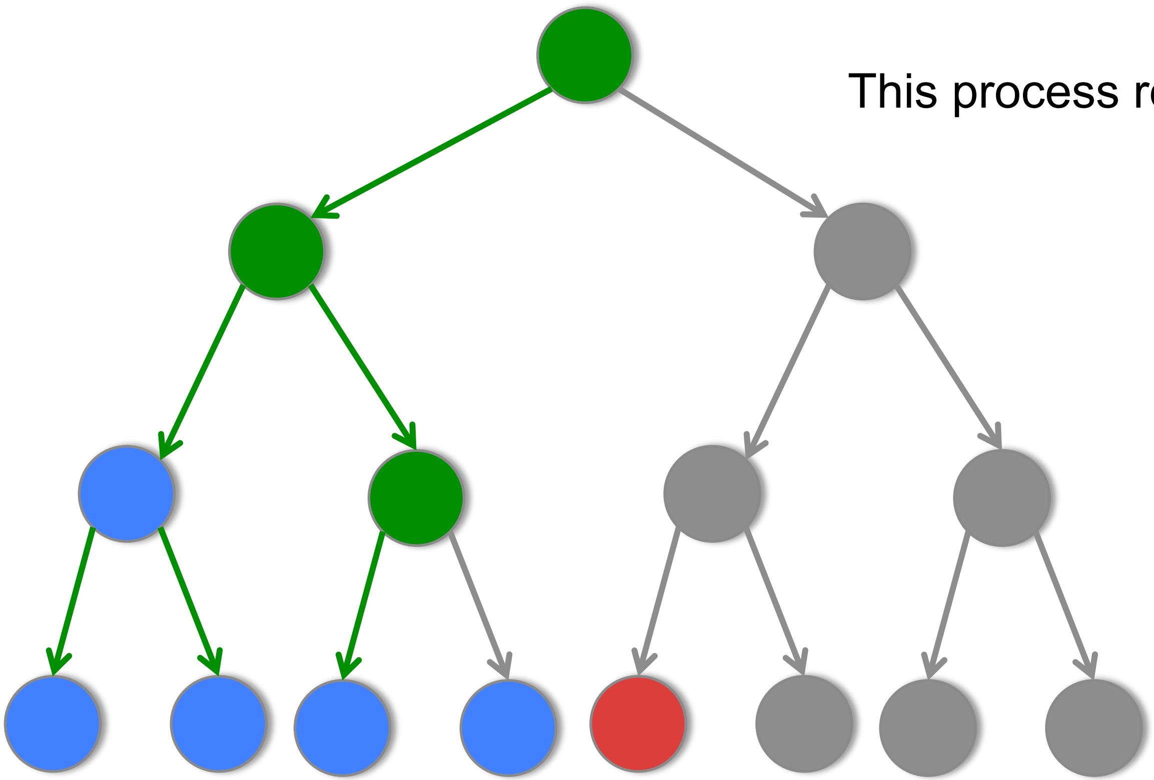
This process repeats

DEPTH FIRST SEARCH



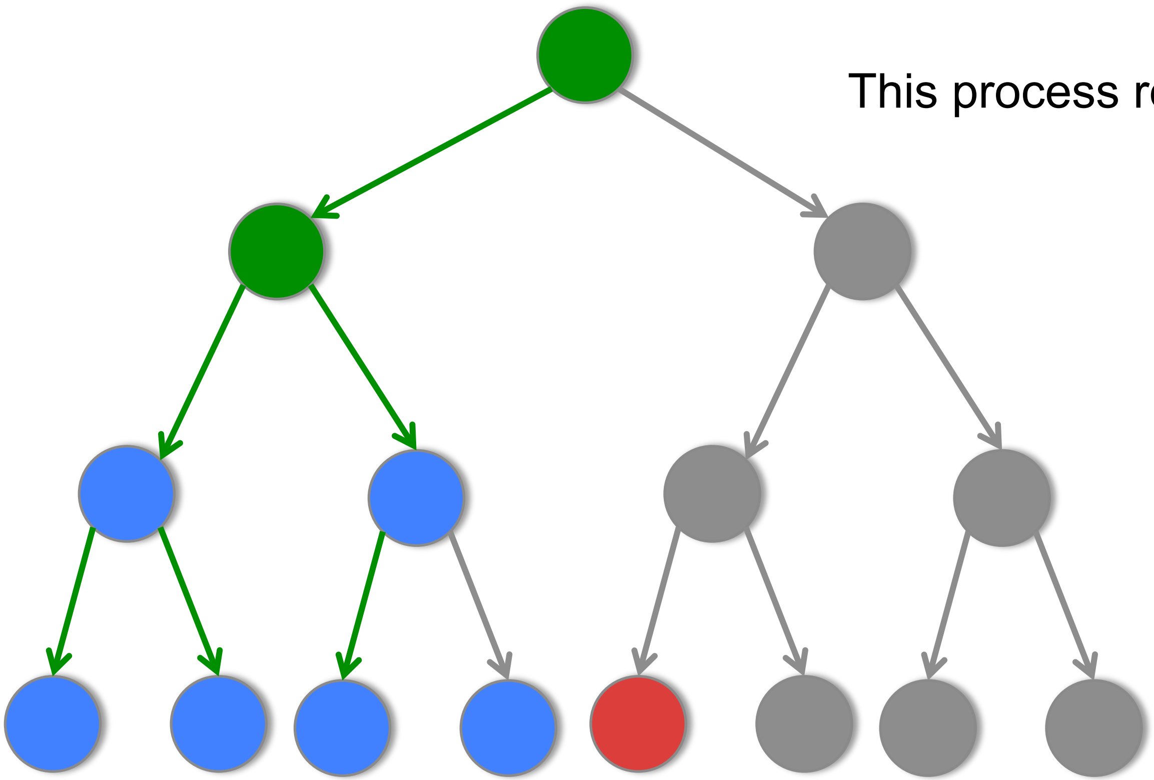
This process repeats

DEPTH FIRST SEARCH



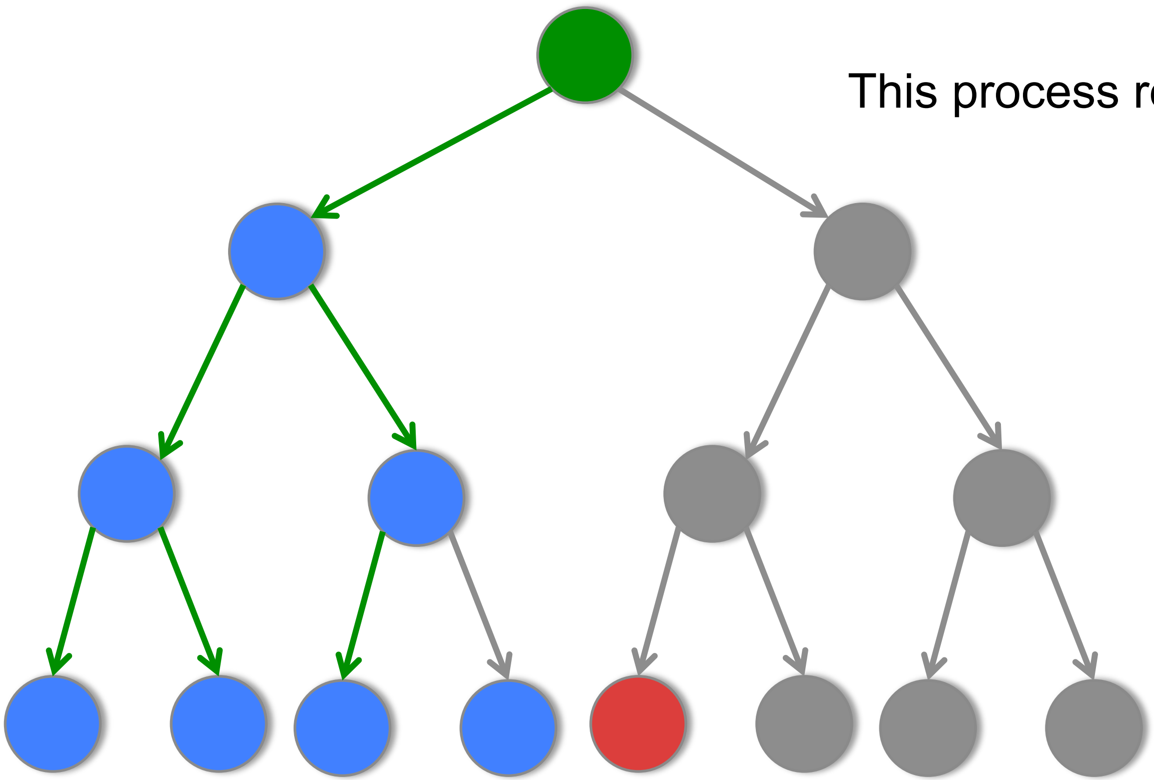
This process repeats

DEPTH FIRST SEARCH



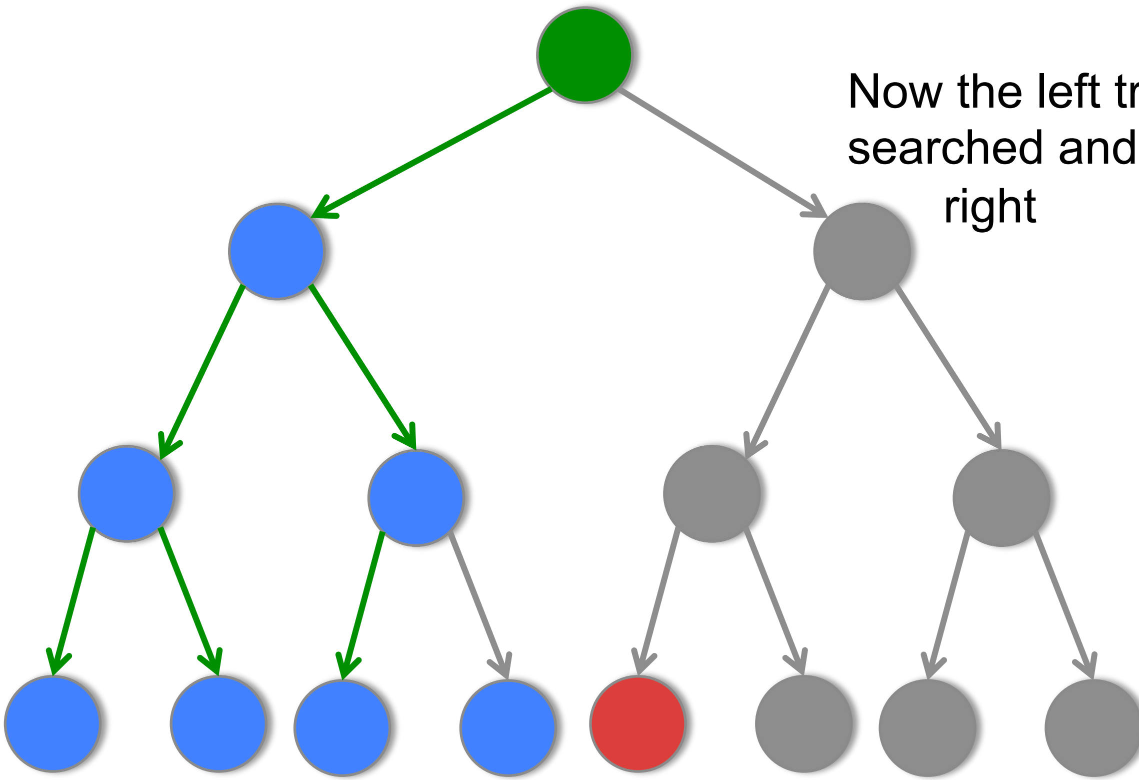
This process repeats

DEPTH FIRST SEARCH



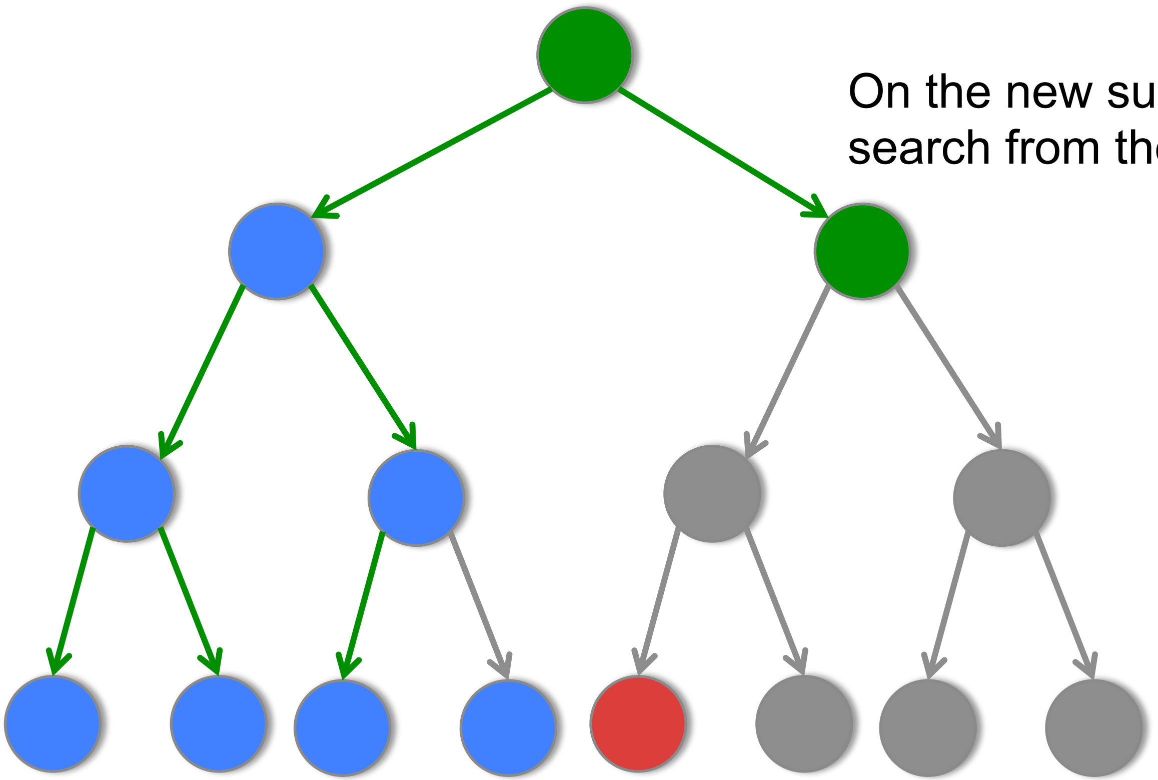
This process repeats

DEPTH FIRST SEARCH



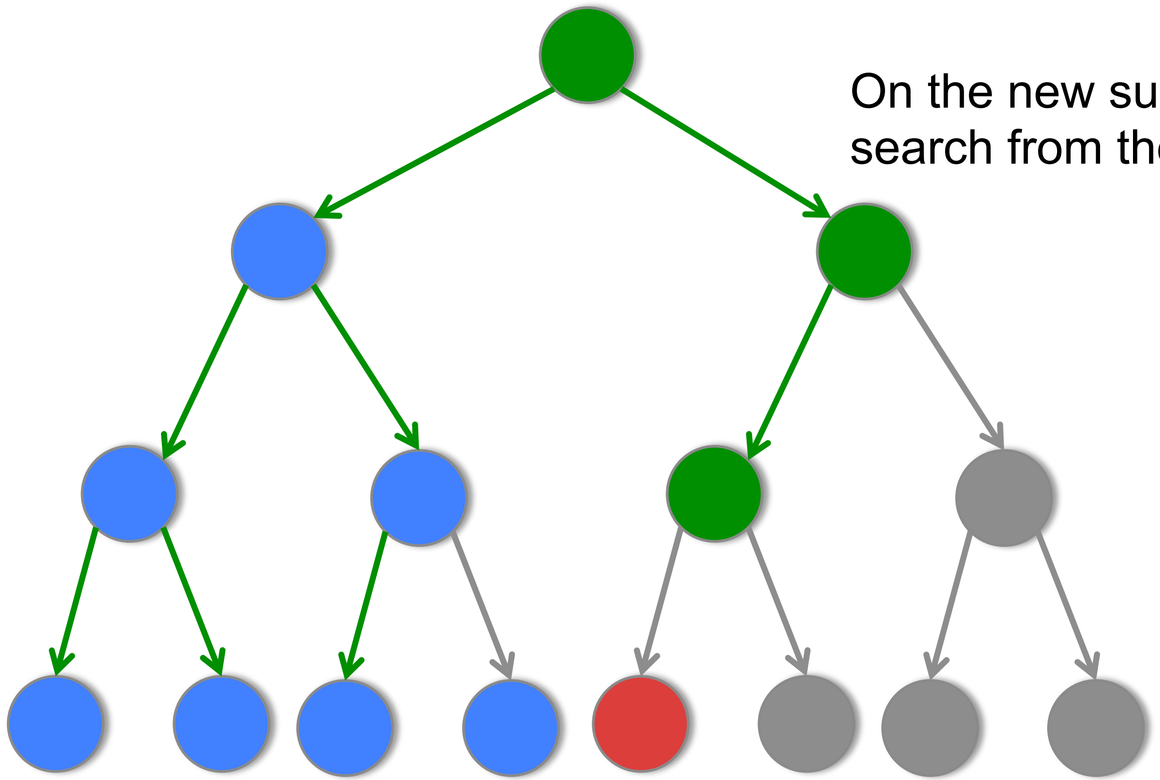
Now the left tree is completely searched and we can search the right

DEPTH FIRST SEARCH



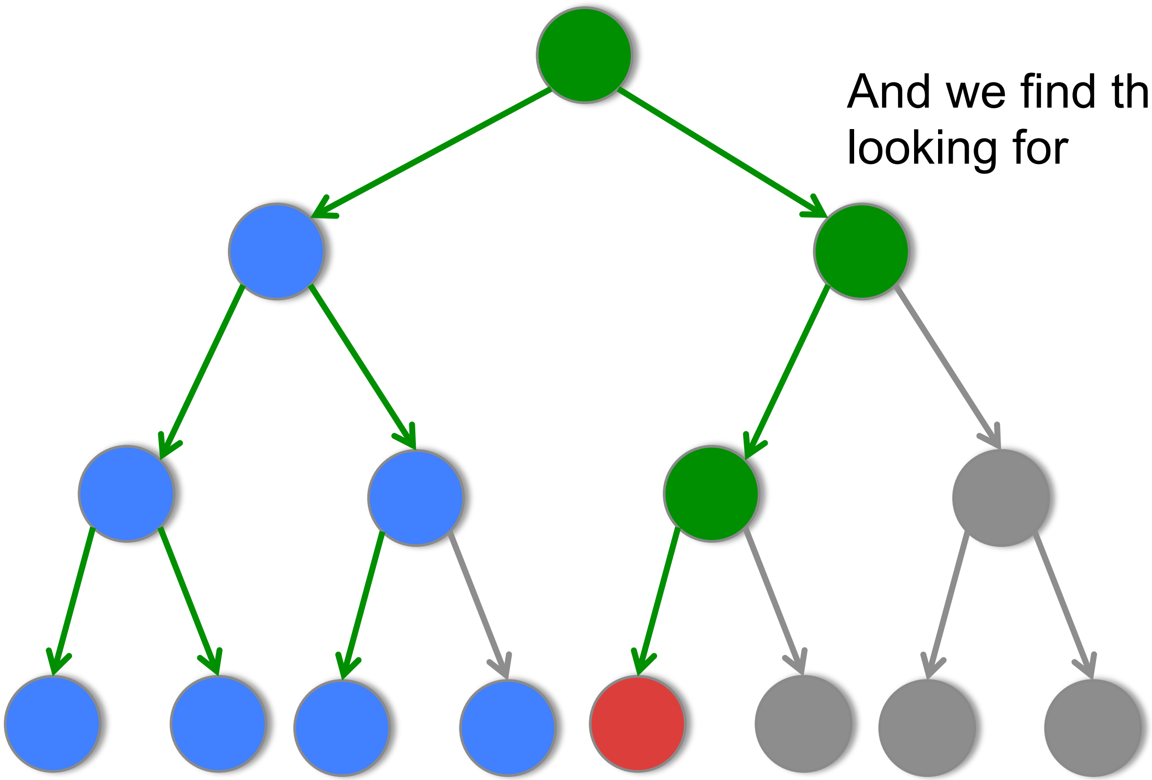
On the new subtree, we begin search from the left

DEPTH FIRST SEARCH



On the new subtree, we begin search from the left

DEPTH FIRST SEARCH



And we find the object we're looking for

DEPTH FIRST SEARCH

- **How does this work in application?**
  - For each node, it searches its left subtree entirely and then moves to the right tree
  - Here search works by breaking the problem down into sub-problems
  - This is a good indication that we use recursion

RECURSION

- **Recap from 143**
- **What is recursion?**
  - A problem that calls itself (with a smaller version of the input)
- **Example:**
  - Linked list search
  - *Take a few minutes and discuss a recursive approach (sorted or unsorted)*

RECURSION

```
public boolean LLsearch(int toFind){
      return LLsearch(toFind,first)
}
private boolean LLsearch(int toFind,Node curr){
      if(curr == null) return false;
      if(curr.data == toFind) return true;
      return LLsearch(toFind,curr.next);
}
```

RECURSION

- **How do we apply this approach to DFS?**
  - *Discuss among yourselves*
  - *Consider what is the "subproblem"*
- **What does it look like?**

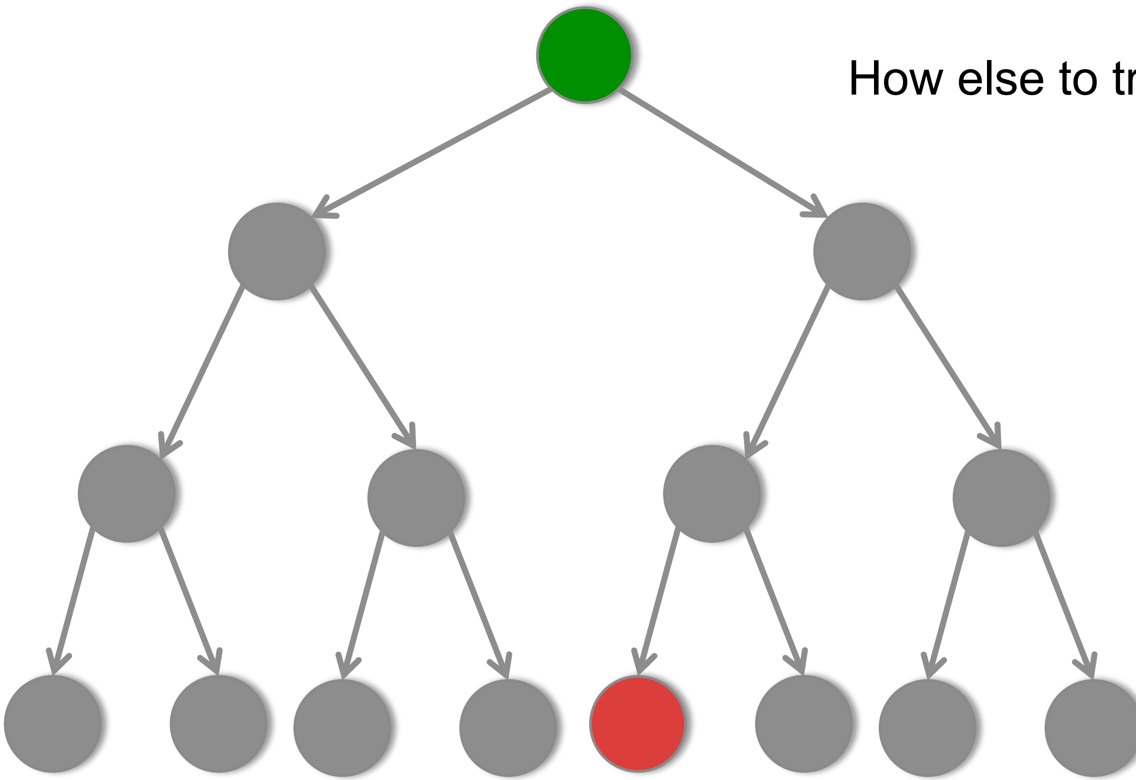
RECURSION

```
public boolean DFSearch(int toFind){
	return DFSearch(toFind,root)
}
private boolean DFSearch(int toFind,Node curr){
	if(curr == null) return false;
	if(curr.data == toFind) return true;
	if(DFSearch(toFind,curr.left)) return true;
	if(DFSearch(toFind,curr.right)) return true;
	return false;
}
```

DEPTH FIRST SEARCH

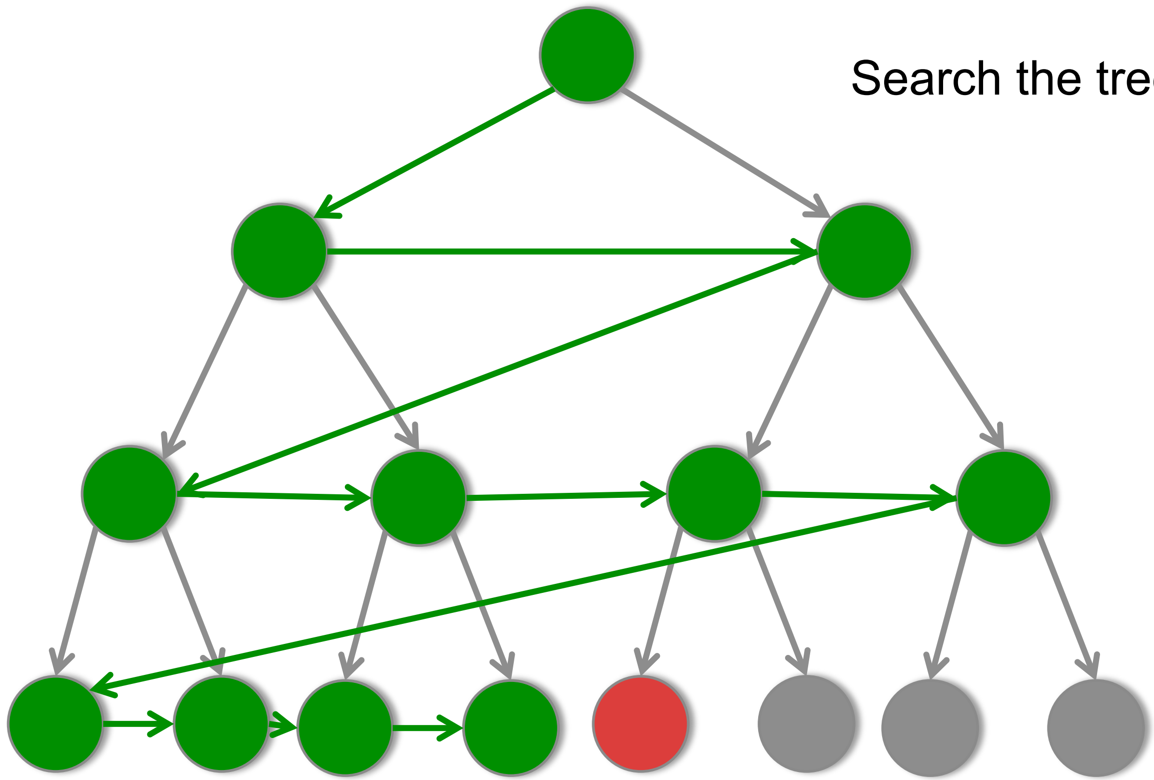
- Treat each subtree as a subproblem and solve recursively.
- Will go to maximum depth first.
- When the node is found, the result will return up the stack
- What might be a different approach?

ALTERNATE APPROACH



How else to traverse?

ALTERNATE APPROACH



Search the tree from top to bottom

BREADTH FIRST SEARCH

- **If we want to traverse the tree from top to bottom, how might we go about doing this?**
  - *Discuss among yourselves for a minute*
  - *Can this approach be reduced to a subproblem?*
  - Not easily!

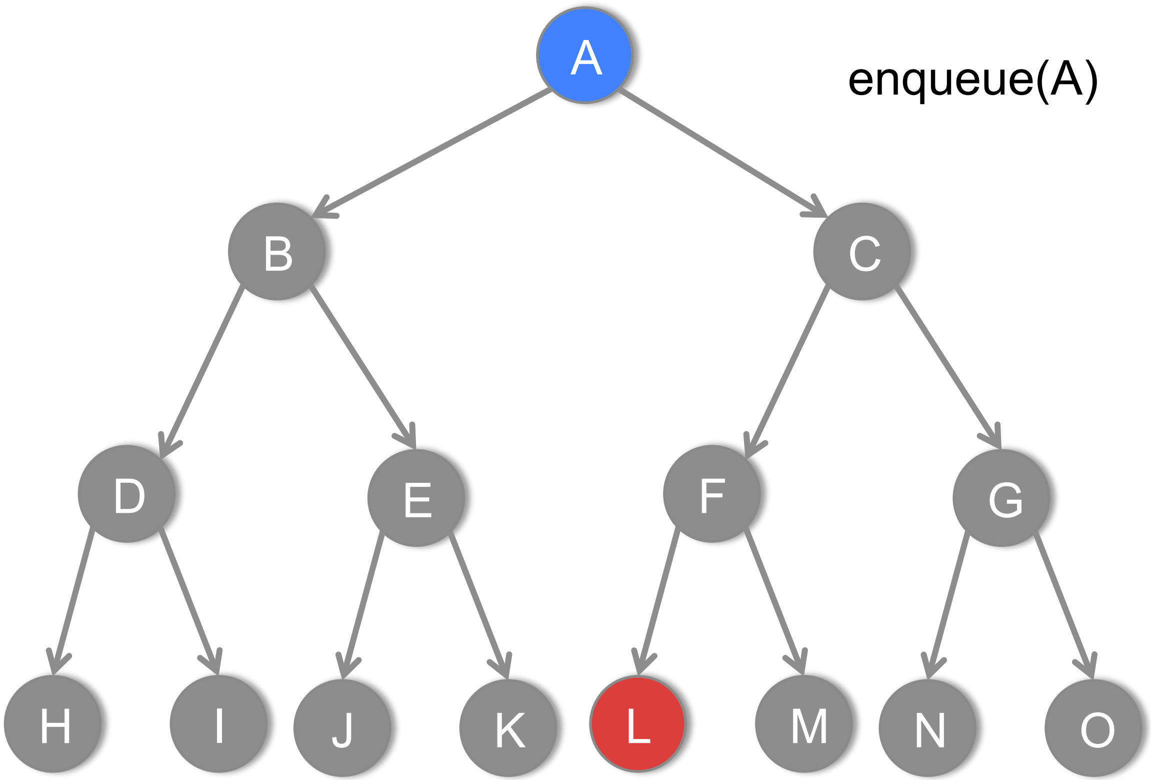
BREADTH FIRST SEARCH

- **Consider the approach**
  - Start with the root
  - Search all nodes of depth 1
  - Search all nodes of depth 2
  - …
  - *How do we get this ordering?*

BREADTH FIRST SEARCH

- **What if we use a Queue?**
  - Enqueue the root
  - Then what?

BREADTH FIRST SEARCH



enqueue(A)

Queue:

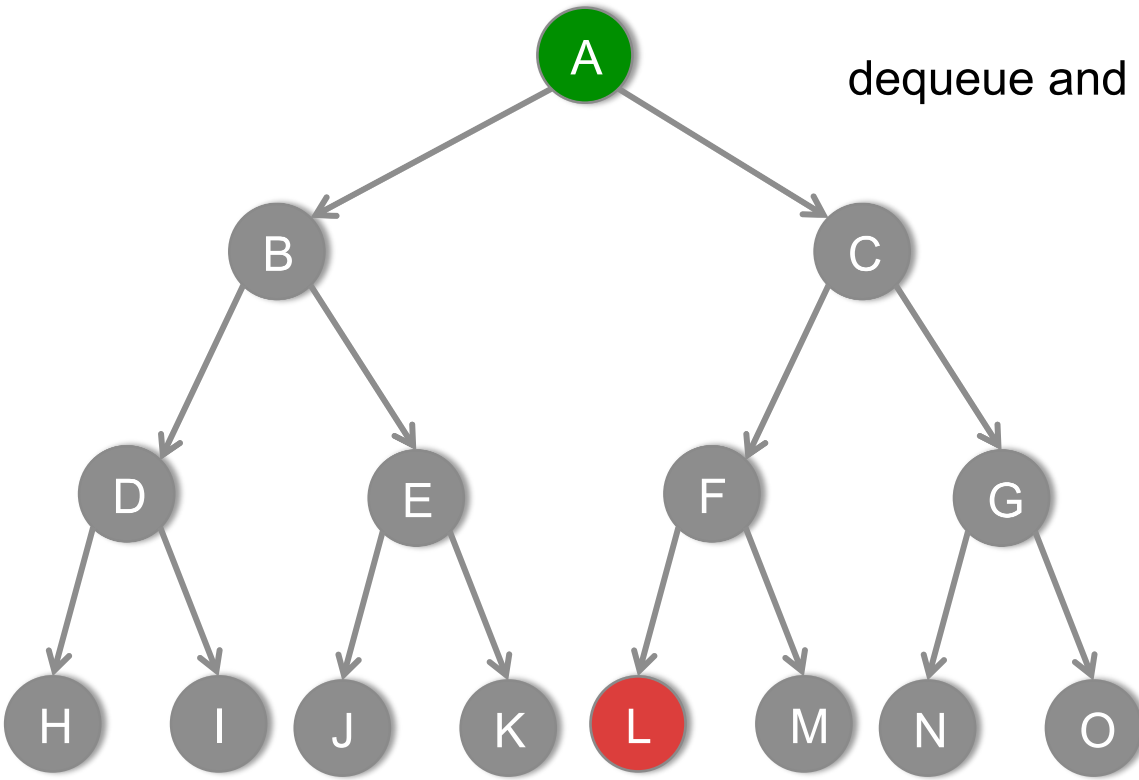
BREADTH FIRST SEARCH

- **What if we use a Queue?**

```
enqueue the root

while the queue has elements:
      dequeue the node
      if it matches our search string
            return true
      if it doesn't,
            enqueue its non-null children
return false;
```

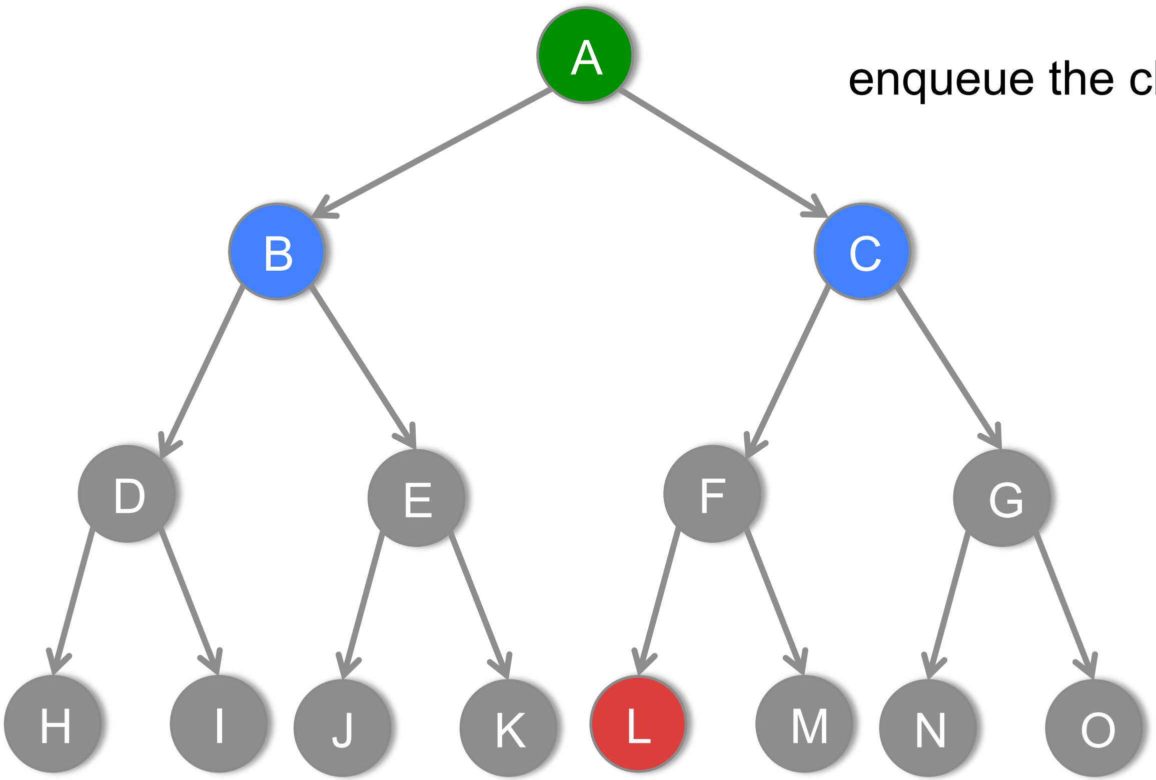
BREADTH FIRST SEARCH



dequeue and check the node

Queue:

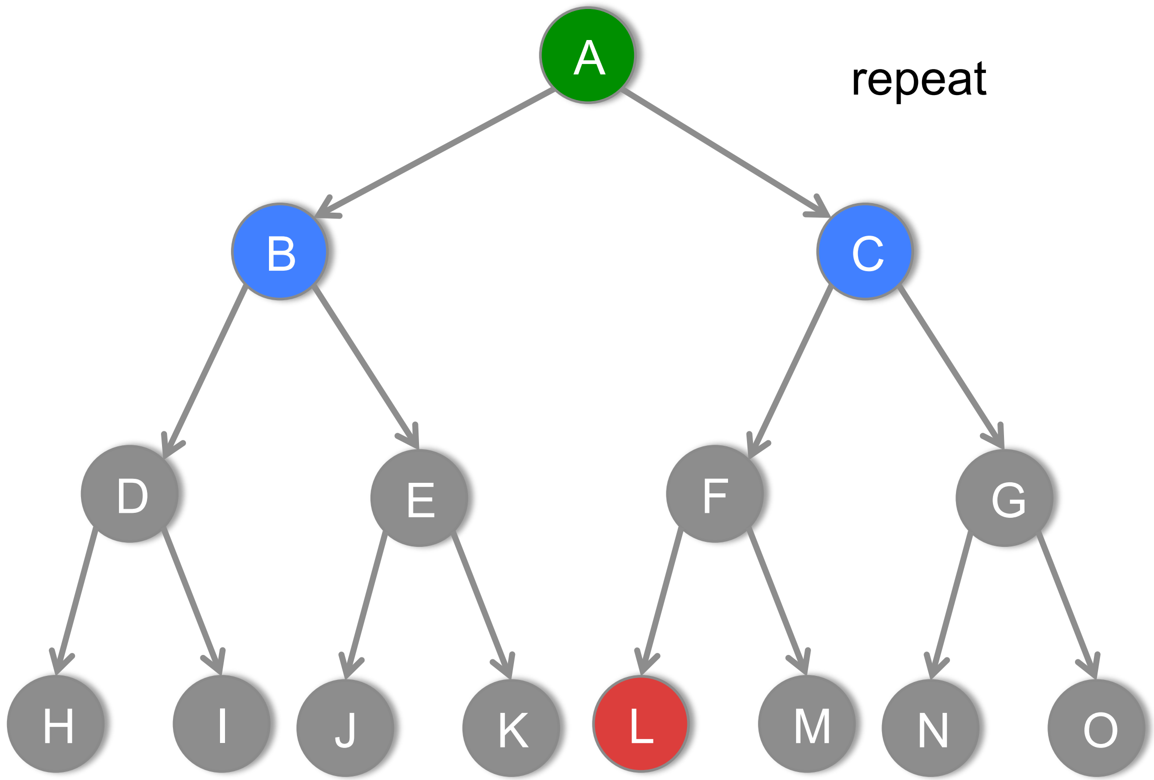
BREADTH FIRST SEARCH



enqueue the children

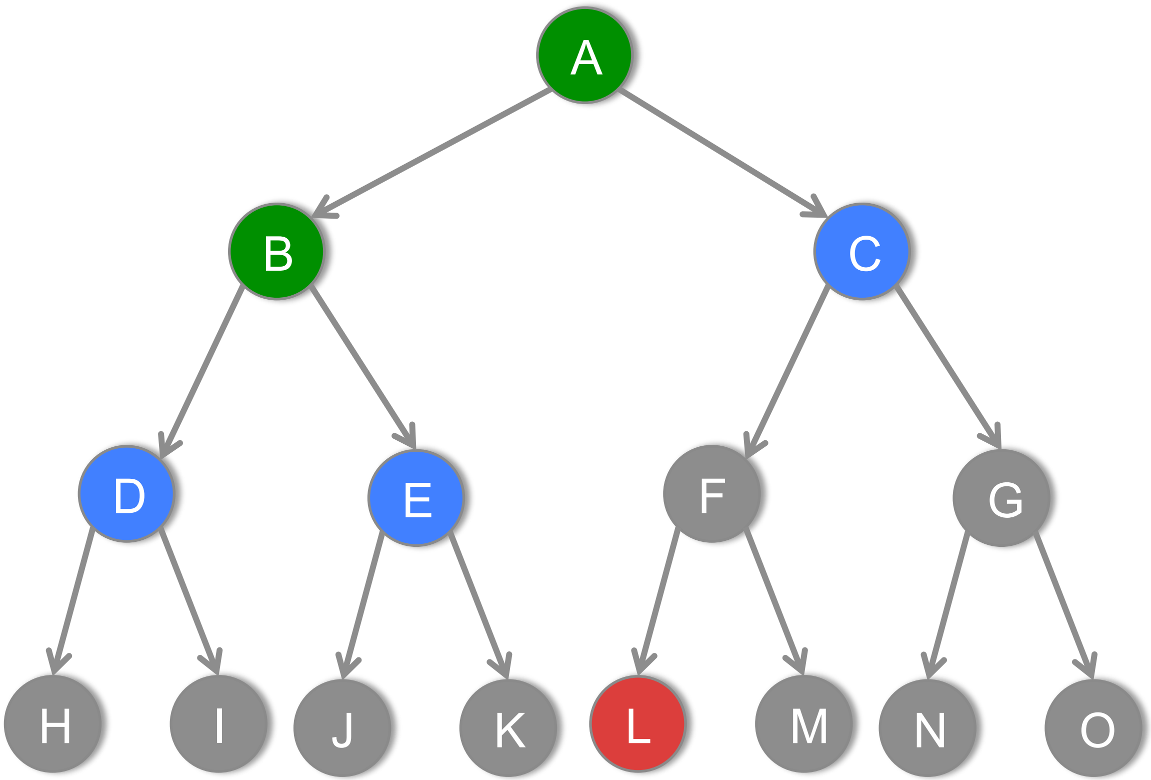
Queue: B | C |

BREADTH FIRST SEARCH



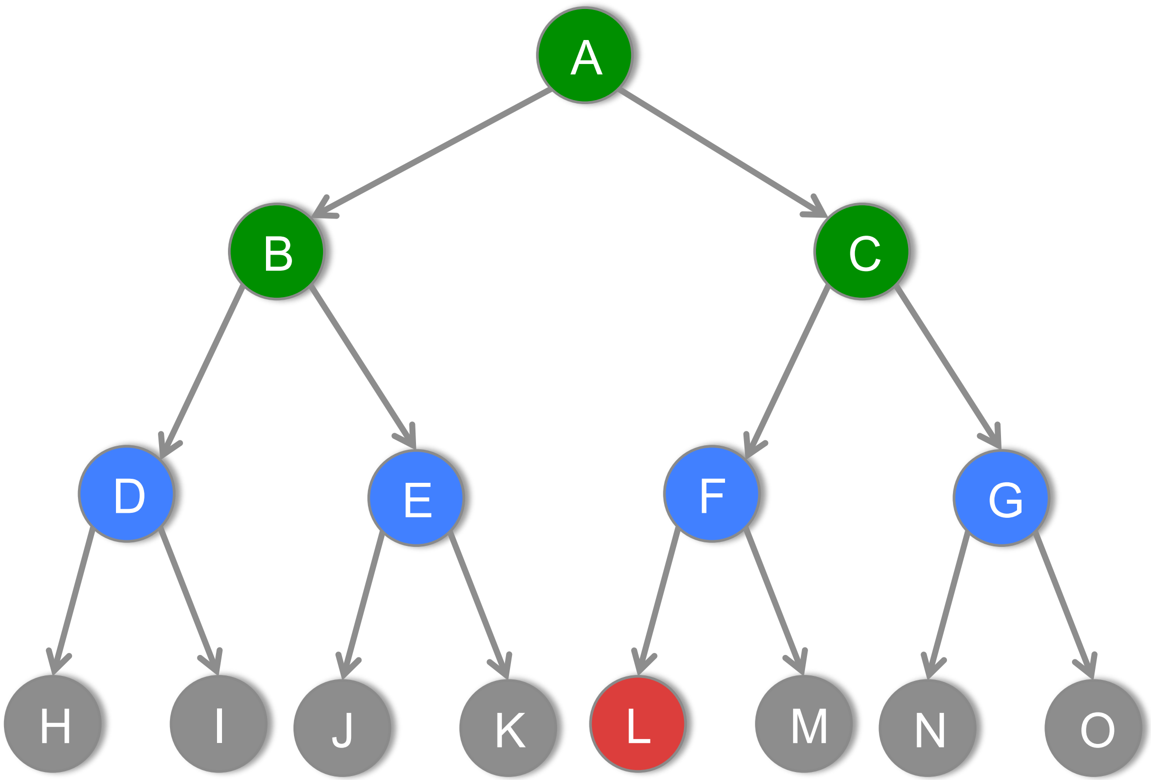
Queue: | B | C | |

BREADTH FIRST SEARCH



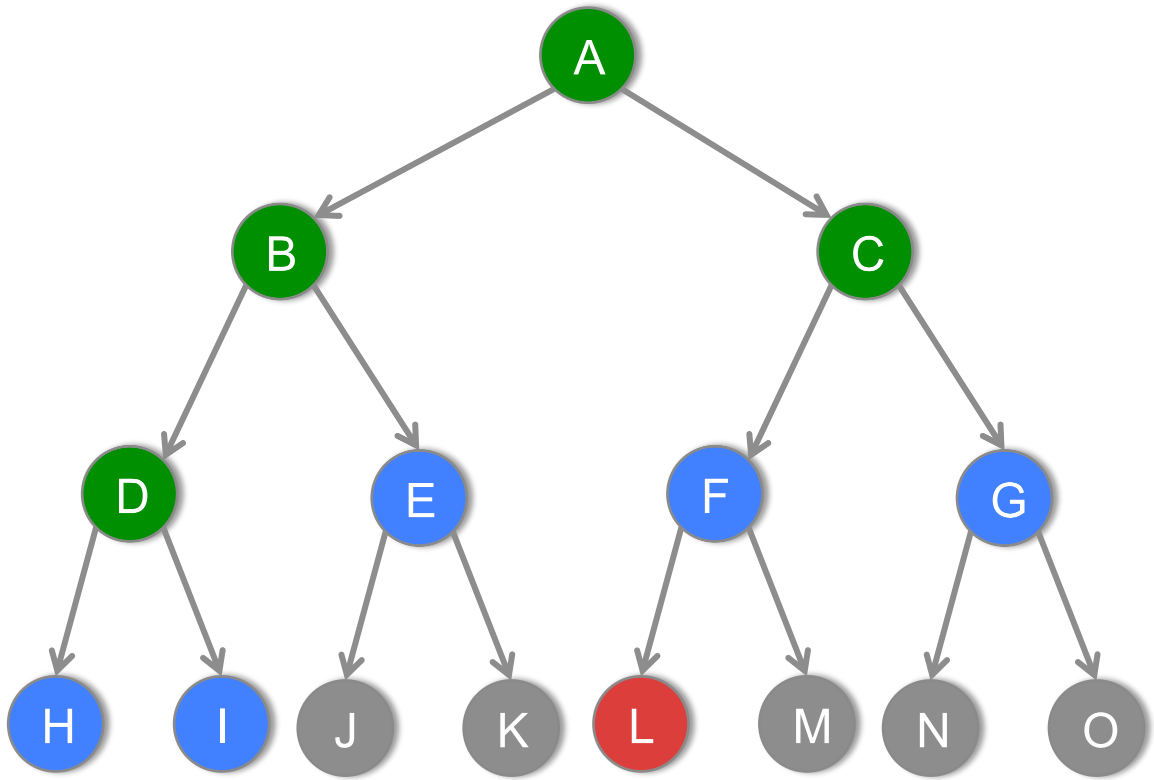
Queue: C | D | E |

BREADTH FIRST SEARCH



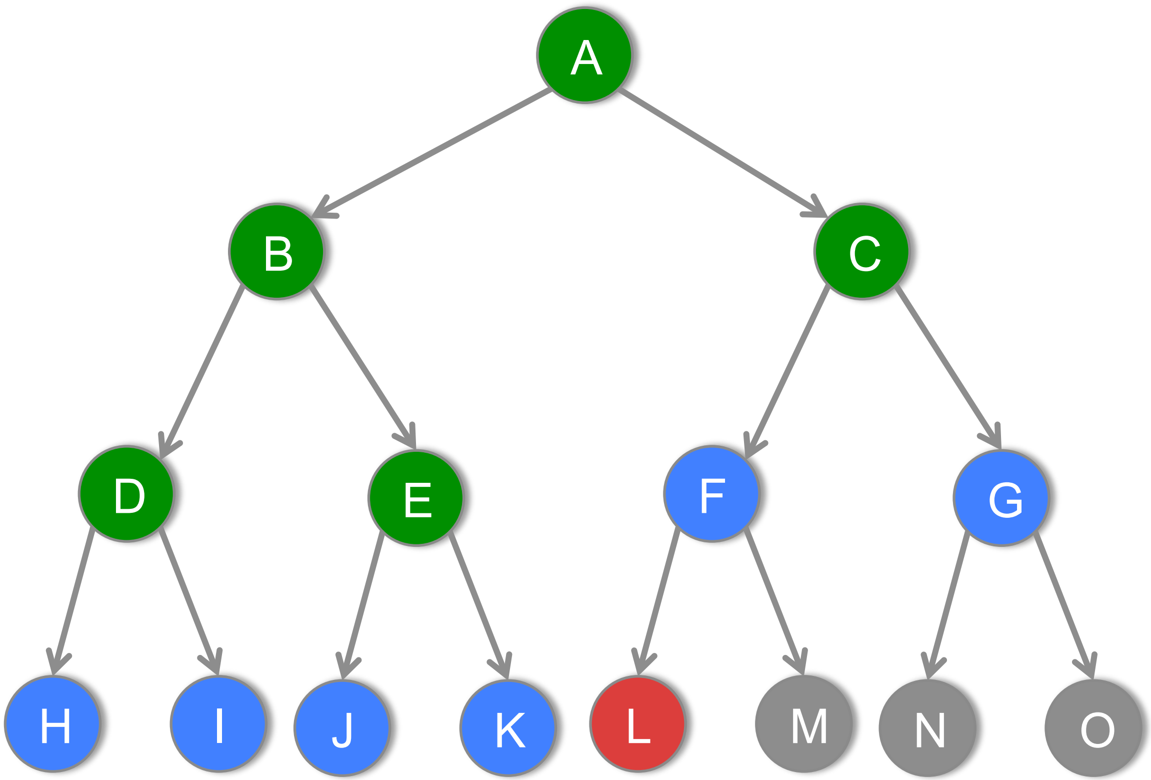
Queue: D | E | F | G |

BREADTH FIRST SEARCH



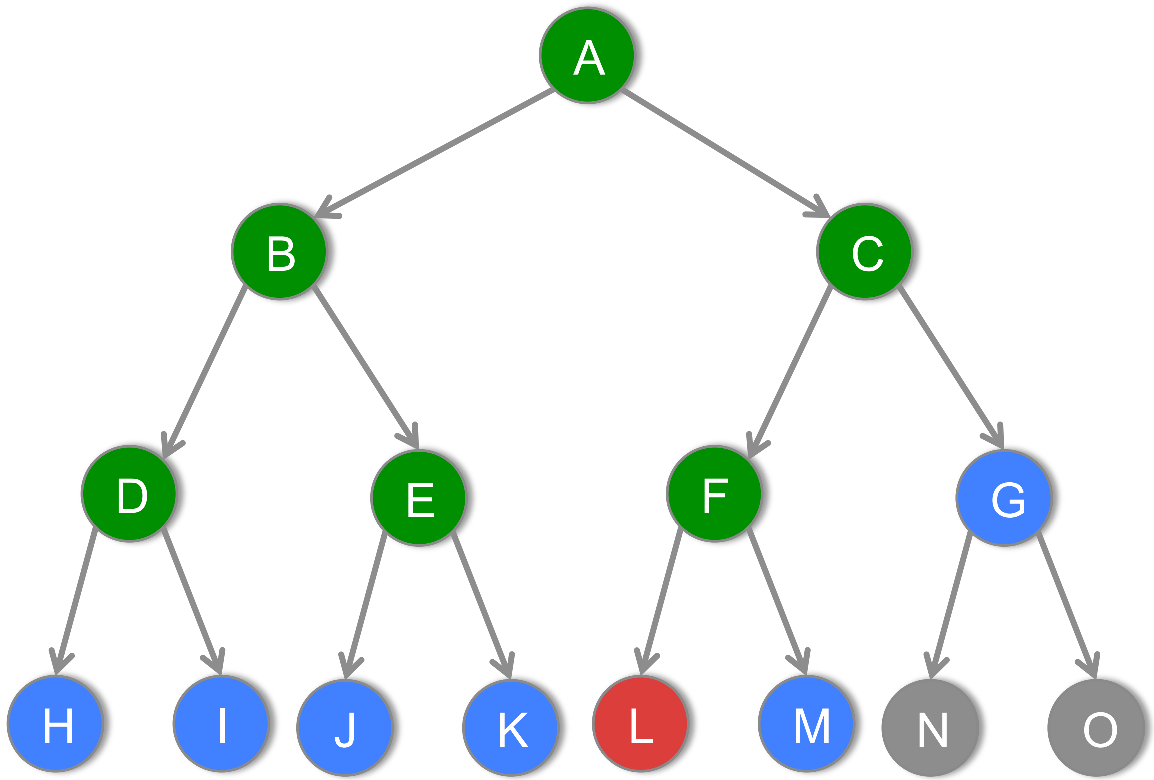
Queue: E | F | G | H | I |

BREADTH FIRST SEARCH



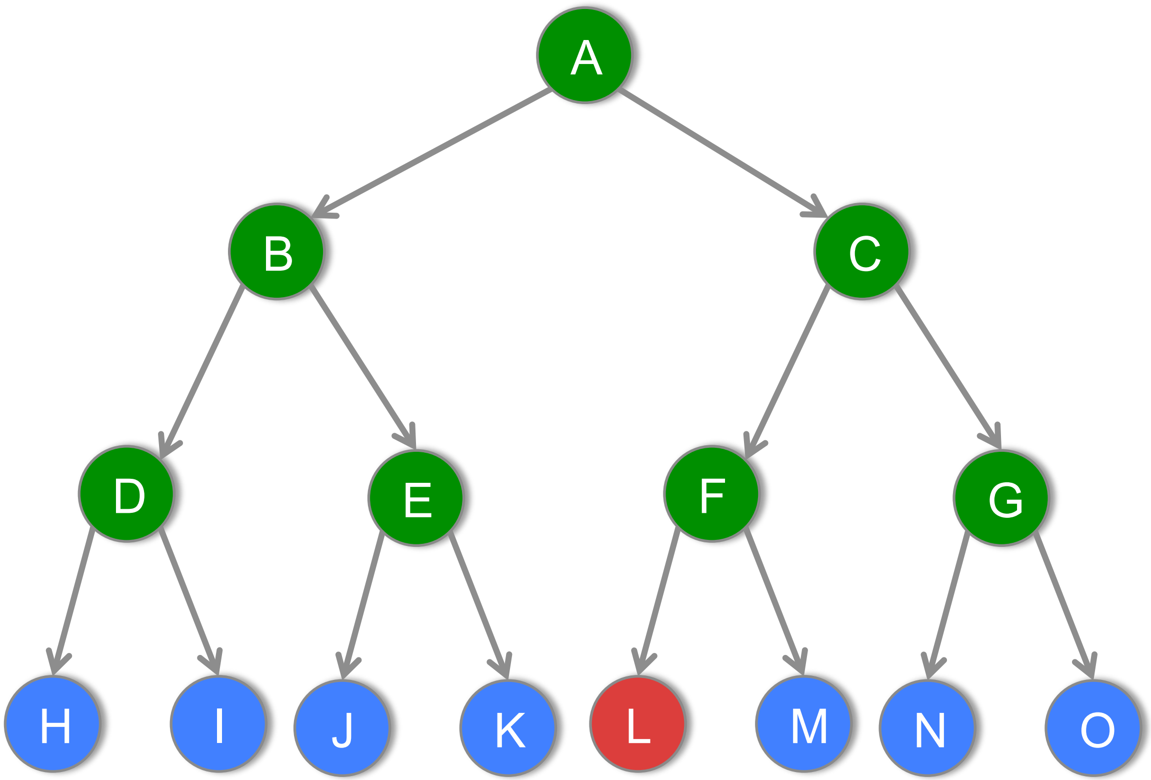
Queue: F | G | H | I | J | K |

BREADTH FIRST SEARCH



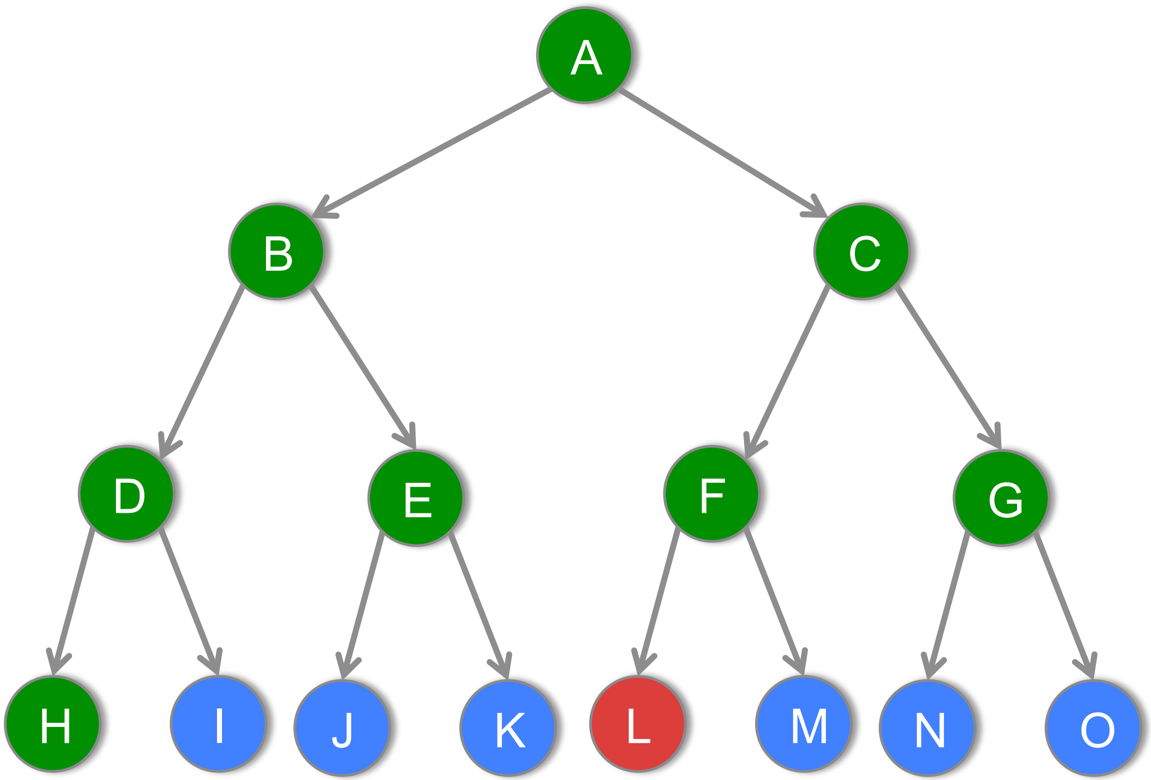
Queue: G | H | I | J | K | L | M

BREADTH FIRST SEARCH



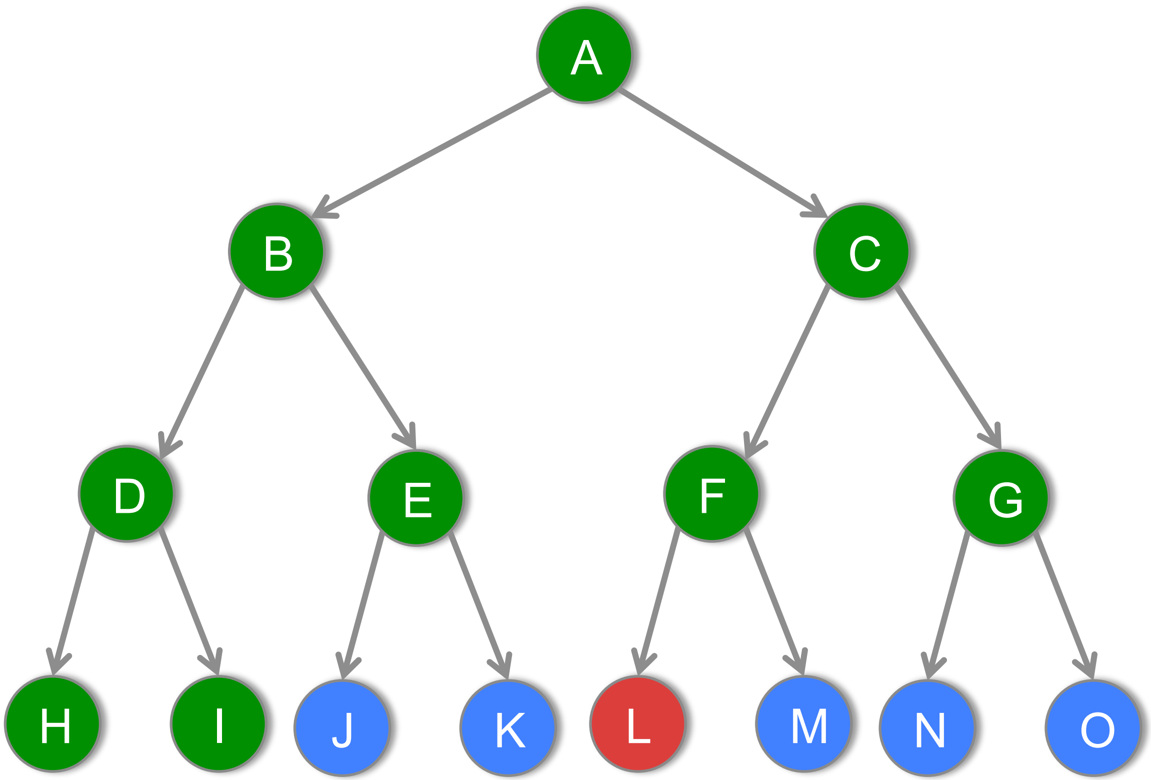
Queue: H | I | J | K | L | M | N | O

BREADTH FIRST SEARCH



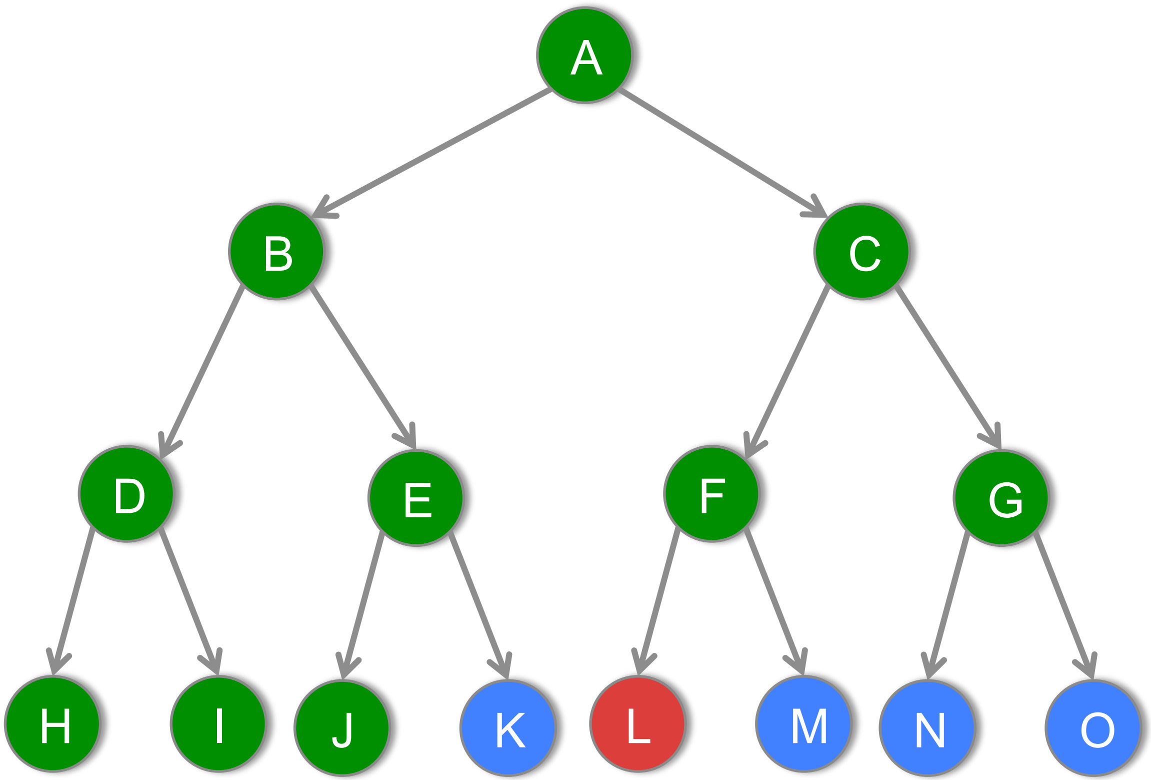
Queue: I | J | K | L | M | N | O

BREADTH FIRST SEARCH



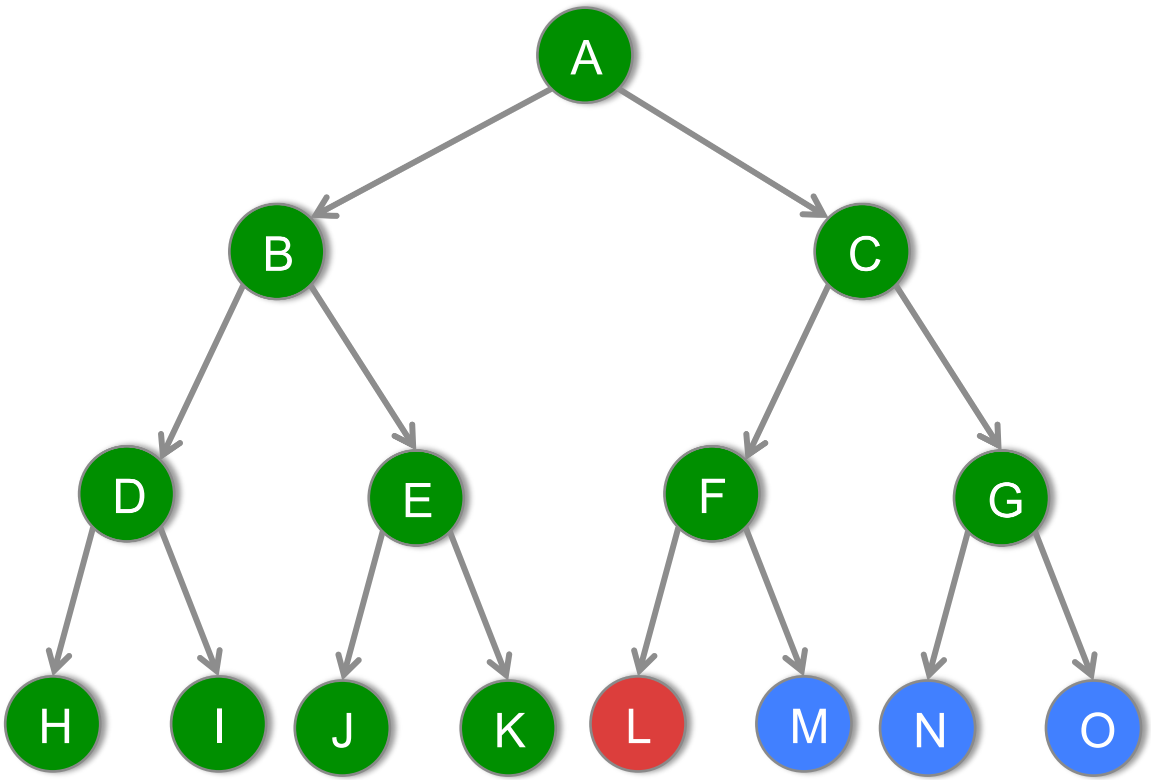
Queue: J | K | L | M | N | O

BREADTH FIRST SEARCH



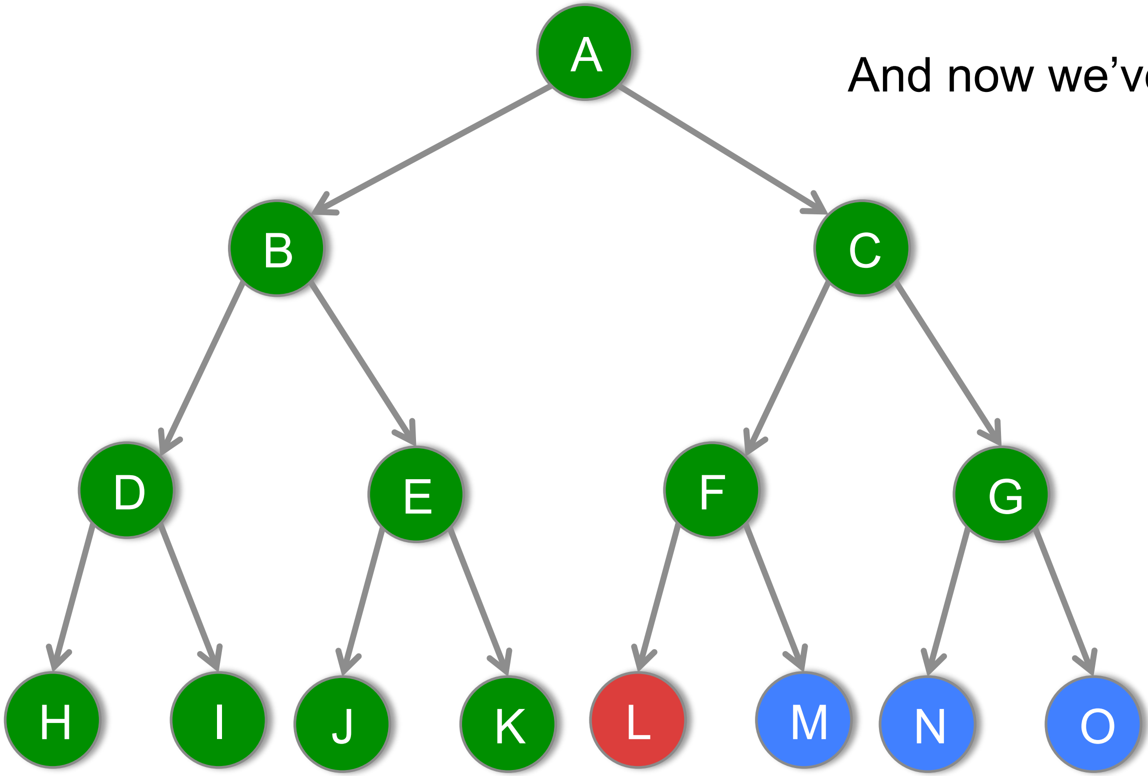
Queue: | K | L | M | N | O |

BREADTH FIRST SEARCH



Queue: | L | M | N | O |

BREADTH FIRST SEARCH



And now we've found it!

Queue: L | M | N | O

BREADTH FIRST SEARCH

- **Use a queue to keep track of the order**
  - *What happens if we use a stack?*
    - *Depth first search! These things are related!*
- **Next week**
  - Balance and the O(n) problem