Lecture 12: Introduction to Graphs and Trees

CS 5002: Discrete Math

Tamara Bonaci, Adrienne Slaughter

Northeastern University

November 29, 2018

1 Review: Proof Techniques

2 Some Graph and Tree Problems

3 Introduction to Trees

4 Special Trees

5 Tree Traversals

6 Introduction to Graphs

7 Graph Representations

8 Graph Traversals

9 Path Finding in a Graph

Section 1

Review: Proof Techniques

Proving Correctness

How to prove that an algorithm is correct?

Proof by:

- Counterexample (*indirect proof*)
- Induction (*direct proof*)
- Loop Invariant

Other approaches: proof by cases/enumeration, proof by chain of iffs, proof by contradiction, proof by contrapositive

Proof by Counterexample

Searching for counterexamples is the best way to disprove the correctness of some things.

- Identify a case for which something is NOT true
- If the proof seems hard or tricky, sometimes a counterexample works
- Sometimes a counterexample is just easy to see, and can shortcut a proof
- If a counterexample is hard to find, a proof might be easier

Proof by Induction

Failure to find a counterexample to a given algorithm does not mean “it is obvious” that the algorithm is correct.

Mathematical induction is a very useful method for proving the correctness of recursive algorithms.

1. Prove base case
2. Assume true for arbitrary value n
3. Prove true for case $n + 1$

Proof by Loop Invariant

- Built off proof by induction.
- Useful for algorithms that loop.

Formally: find loop invariant, then prove:

1. Define a Loop Invariant
2. Initialization
3. Maintenance
4. Termination

Informally:

1. Find p, a loop invariant
2. Show the base case for p
3. Use induction to show the rest.

Proof by Loop Invariant Is...

Invariant: something that is always true

After finding a candidate loop invariant, we prove:

1. ***Initialization***: How does the invariant get initialized?
2. ***Loop Maintenance***: How does the invariant change at each pass through the loop?
3. ***Termination***: Does the loop stop? When?

Steps to Loop Invariant Proof

After finding your loop invariant:

- Initialization
  - Prior to the loop initiating, does the property hold?
- Maintenance
  - After each loop iteration, does the property still hold, given the initialization properties?
- Termination
  - After the loop terminates, does the property still hold? And for what data?

Things to remember

- ***Algorithm termination*** is necessary for proving correctness of the entire algorithm.
- ***Loop invariant termination*** is necessary for proving the behavior of the given loop.

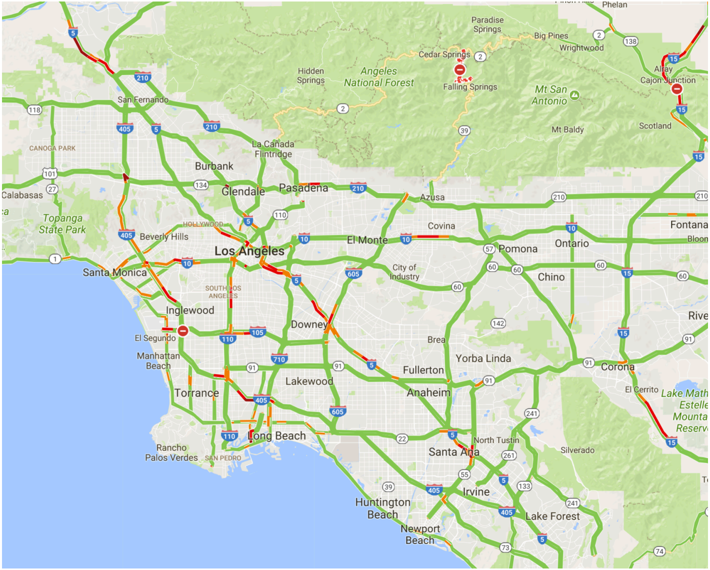
Summary

- Approaches to proving algorithms correct
- Counterexamples
  - Helpful for greedy algorithms, heuristics
- Induction
  - Based on mathematical induction
  - Once we prove a theorem, can use it to build an algorithm
- Loop Invariant
  - Based on induction
  - Key is finding an invariant
- Lots of examples

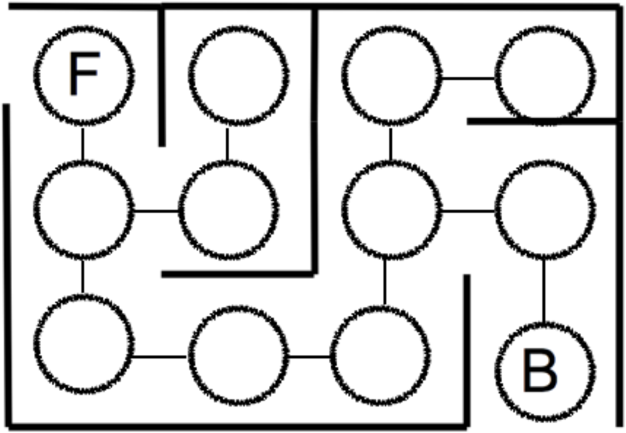
Section 2

Some Graph and Tree Problems

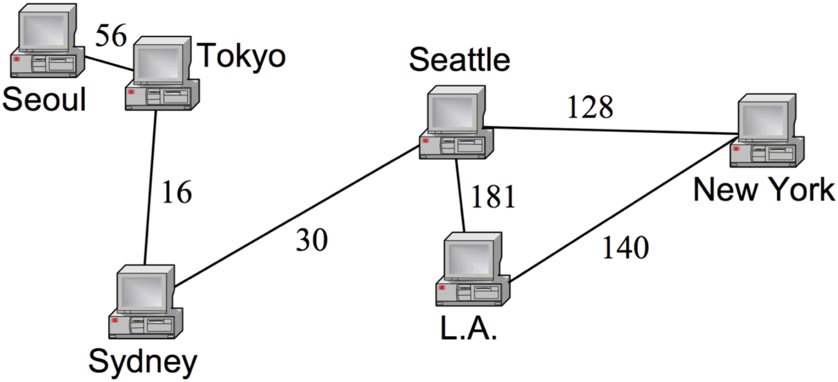
Outdoors Navigation



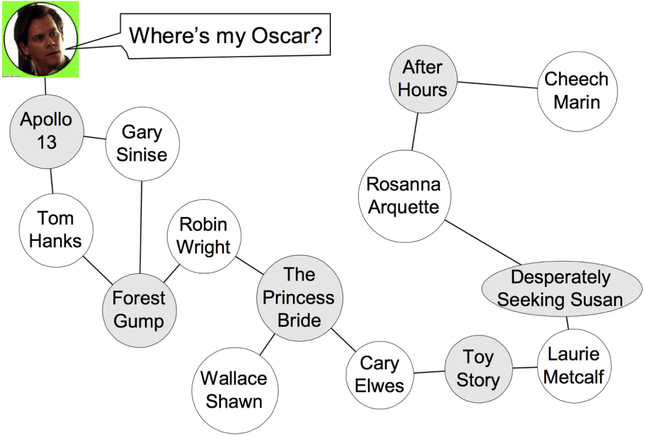
Indoors Navigation



Telecommunication Networks



Social Networks



Section 3

Introduction to Trees

What is a Tree?

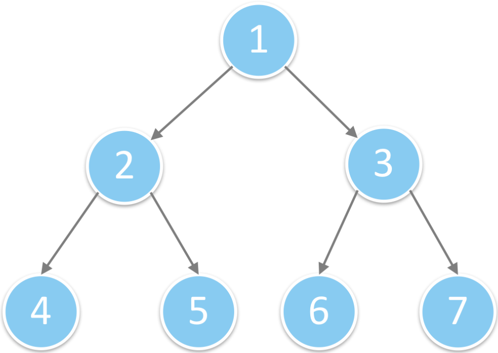
  

Tree - a directed, acyclic structure of linked nodes

- **Directed** - one-way links between nodes (no cycles)
- **Acyclic** - no path wraps back around to the same node twice (typically represents hierarchical data)

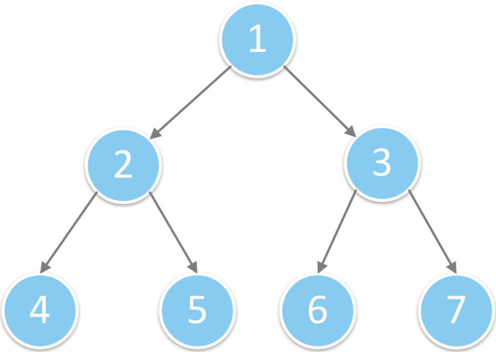
Tree Terminology: Nodes

- **Tree** - a directed, acyclic structure of linked nodes
- **Node** - an object containing a data value and links to other nodes
  - All the blue circles



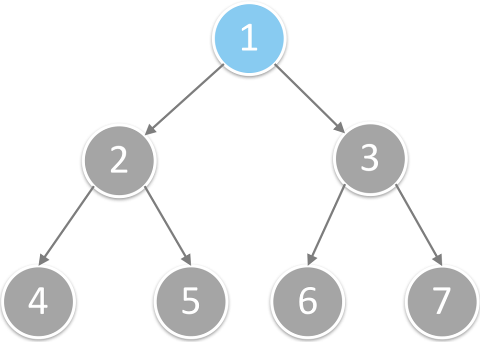
Tree Terminology: Edges

- **Tree** - a directed, acyclic structure of linked nodes
- **Edge** - directed link, representing relationships between nodes
  - All the grey lines



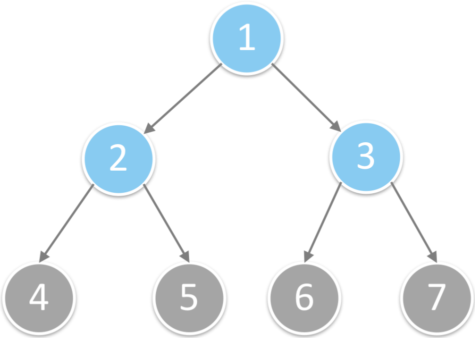
Root Node

- **Tree** - a directed, acyclic structure of linked nodes
- **Root** - the start of the tree tree)
  - The top-most node in the tree
  - Node without parents



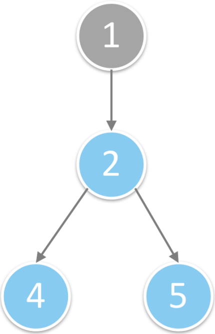
Parent Nodes

- **Tree** - a directed, acyclic structure of linked nodes
- **Parent (ancestor)** - any node with at least one child
  - The blue nodes



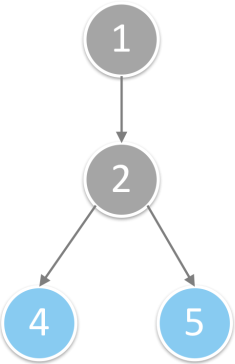
Children Nodes

- **Tree** - a directed, acyclic structure of linked nodes
- **Child (descendant)** - any node with a parent
  - The blue nodes



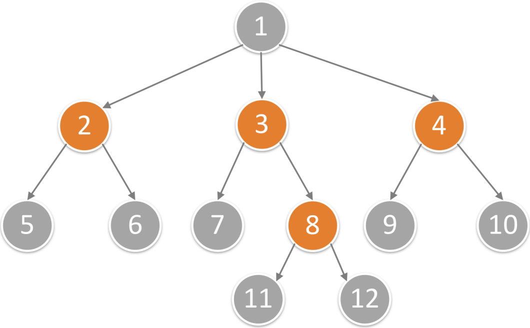
Sibling Nodes

- **Tree** - a directed, acyclic structure of linked nodes
- **Siblings** - all nodes on the same level
    - The blue nodes



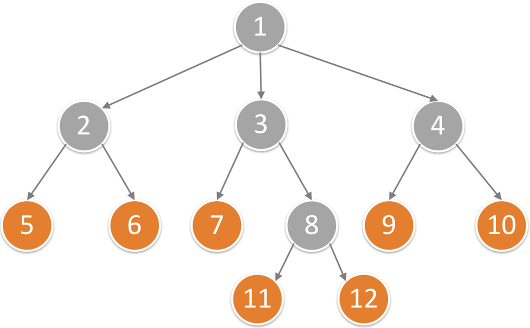
Internal Nodes

- **Tree** - a directed, acyclic structure of linked nodes
- **Internal node** - a node with at least one children (except root)
  - All the orange nodes



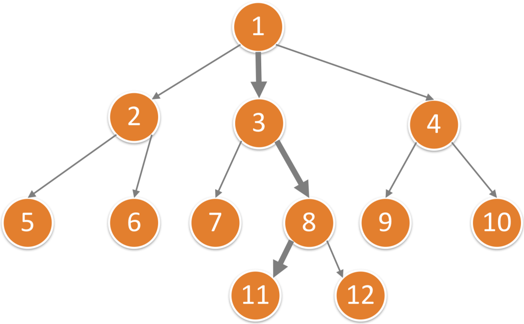
Leaf (External) Nodes

- **Tree** - a directed, acyclic structure of linked nodes
- **External node** - a node without children
  - All the orange nodes



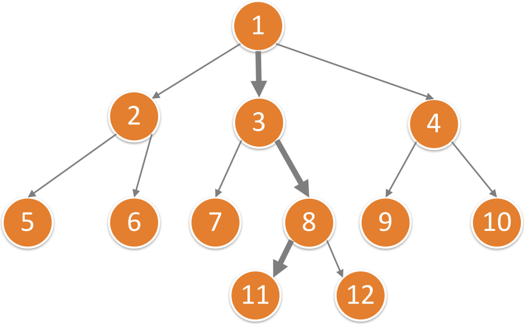
Tree Path

- **Tree** - a directed, acyclic structure of linked nodes
- **Path** - a sequence of edges that connects two nodes
  - All the orange nodes



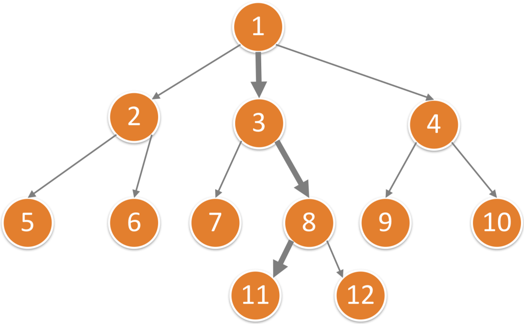
Node Level

- **Node level** - 1 + [the number of connections between the node and the root]
  - The level of node 1 is 1
  - The level of node 11 is 4



Node Height

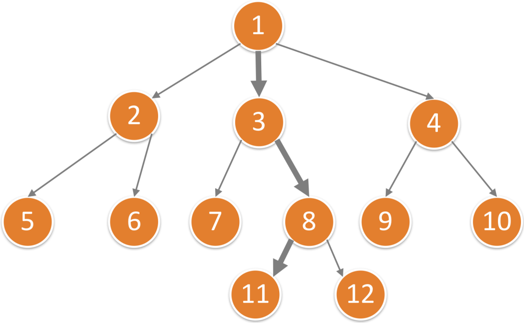
- **Node height** - the length of the longest path from the node to some leaf
  - **The height of any leaf node is 0**
  - The height of node 8 is 1
  - The height of node 1 is 3
  - The height of node 11 is 0



Tree Height

Tree height

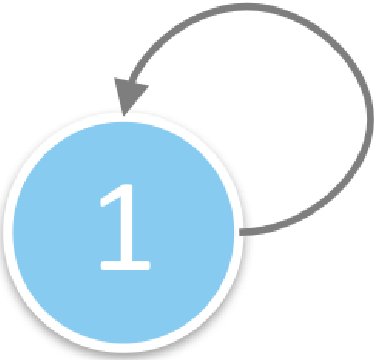
- The height of the root of the tree, or
- The number of levels of a tree -1.
  - The height of the given tree is 3.



What is Not a Tree?

Problems:

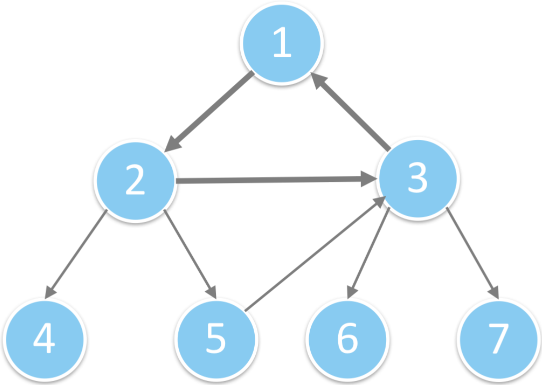
- **Cycles**: the only node has a cycle
- **No root:** the only node has a parent (itself, because of the cycle), so there is no root



What is Not a Tree?

Problems:

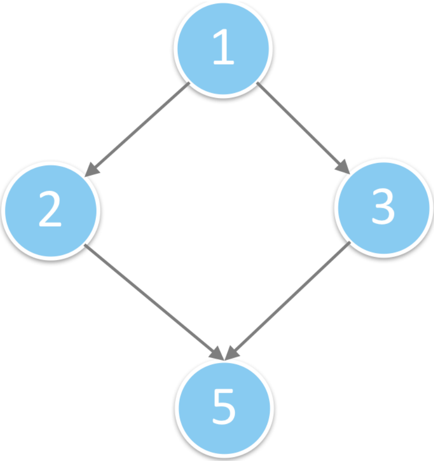
- **Cycles**: there is a cycle in the tree
- **Multiple parents:** node 3 has multiple parents on different levels



What is Not a Tree?

Problems:

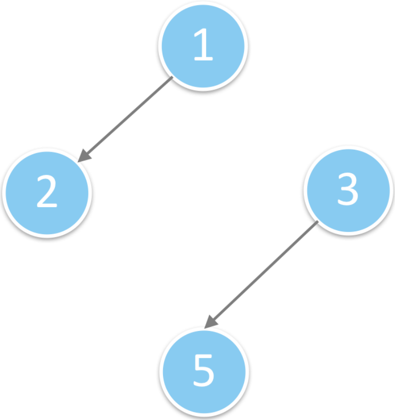
- **Cycles**: there is an undirected cycle in the tree
- **Multiple parents:** node 5 has multiple parents on different levels



What is Not a Tree?

Problems:

- **Disconnected components:** there are two unconnected groups of nodes



Summary: What is a Tree?

- A tree is a set of **nodes**, and that set can be empty
- If the tree is not empty, there exists a special node called a **root**
- The root can have multiple children, each of which can be the root of a **subtree**

Section 4

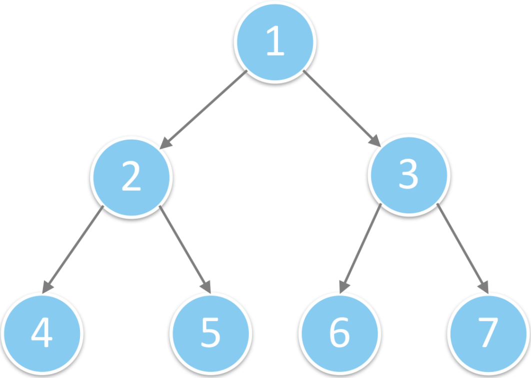
Special Trees

Special Trees

- Binary Tree
- Binary Search Tree
- Balanced Tree
- Binary Heap/Priority Queue
- Red-Black Tree

Binary Trees

Binary tree - a tree where every node has at most two children

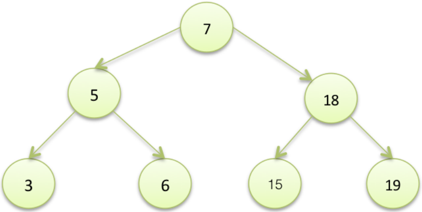


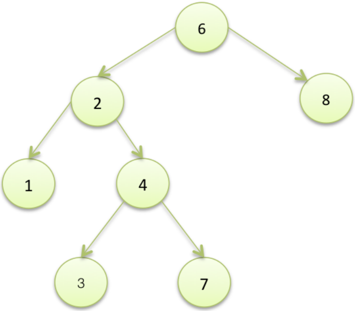
Binary Search Trees

- **Binary search tree (BST)** - a tree where nodes are organized in a **sorted** order to make it easier to search
- At every node, you are guaranteed:
- All nodes rooted at the left child are smaller than the current node value
- All nodes rooted at the right child are smaller than the current node value

Example: Binary Search Trees?

Binary search tree (BST) - a tree where nodes are organized in a **sorted** order to make it easier to search

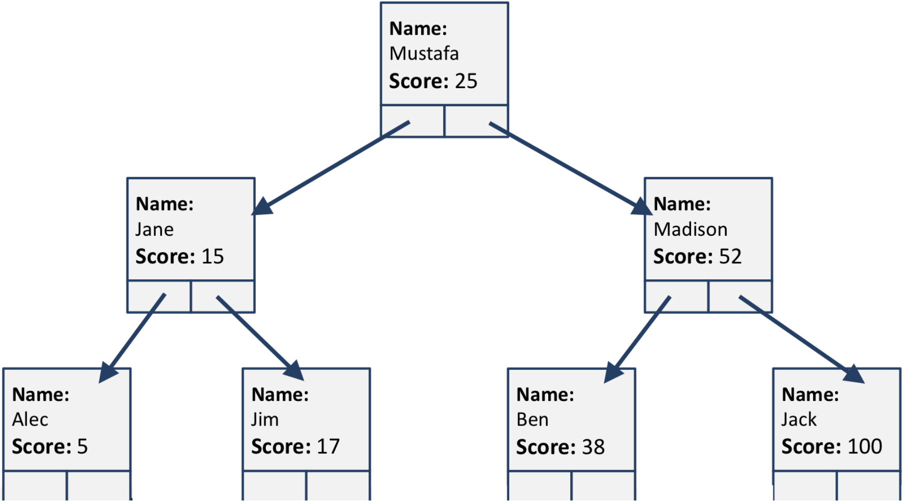




- Left tree is a BST
- Right tree is not a BST - node 7 is on the left hand-side of the root node, and yet it is greater than it

Example: Using BSTs

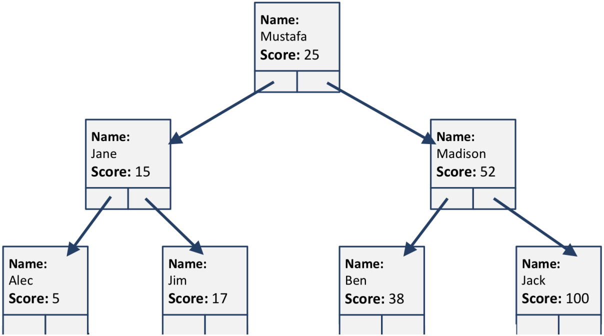
Suppose we want to find who has the score of 15...



Example: Using BSTs

Suppose we want to find who has the score of 15:

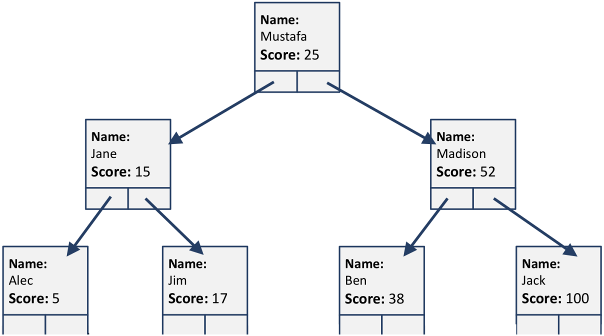
- Start at the root



Example: Using BSTs

Suppose we want to find who has the score of 15:

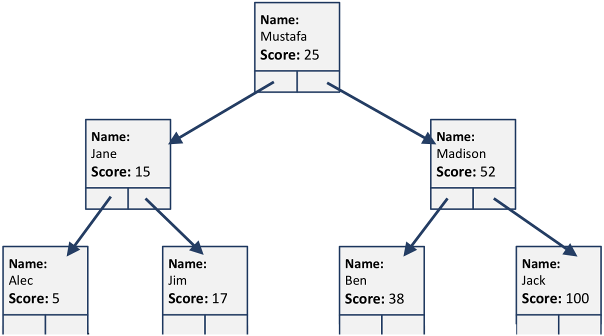
- Start at the root
- If the score is > 15, go to the left



Example: Using BSTs

Suppose we want to find who has the score of 15:

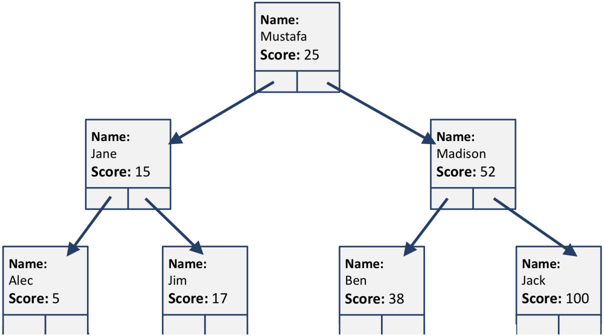
- Start at the root
- If the score is > 15, go to the left
- If the score is < 15, go to the right



Example: Using BSTs

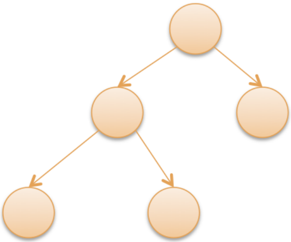
Suppose we want to find who has the score of 15:

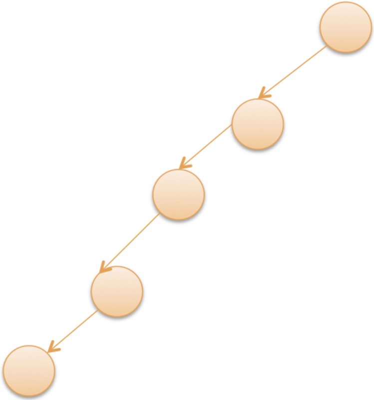
- Start at the root
- If the score is > 15, go to the left
- If the score is < 15, go to the right
- If the score $== 15$, stop



Balanced Trees

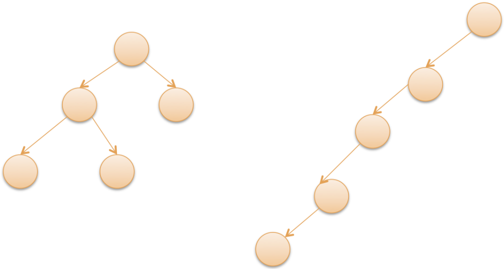
Consider the following two trees. Which tree would it make it easier for us to search for an element?





Balanced Trees

Consider the following two trees. Which tree would it make it easier for us to search for an element?



Observation: height is often key for how fast functions on our trees are. So, if we can, we want to choose a **balanced tree**.

How do we define balance?

- If the heights of the left and right trees are balanced, the tree is balanced, so:

Tree Balance and Height

How do we define balance?

- If the heights of the left and right trees are balanced, the tree is balanced, so:

$$|(\text{height}(\text{left}) - \text{height}(\text{right}))|$$

Tree Balance and Height

How do we define balance?

- If the heights of the left and right trees are balanced, the tree is balanced, so:

  $$|(\text{height}(\text{left}) - \text{height}(\text{right}))|$$

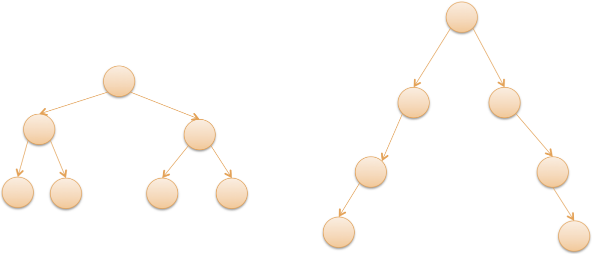
- Anything wrong with this approach?

How do we define balance?

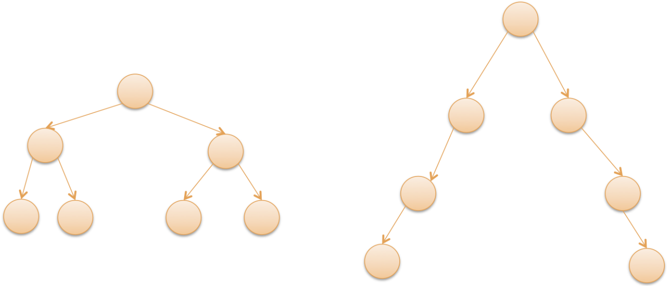
- If the heights of the left and right trees are balanced, the tree is balanced, so:

  $$|(\text{height}(\text{left}) - \text{height}(\text{right}))|$$

- Anything wrong with this approach?
- Are these trees balanced?

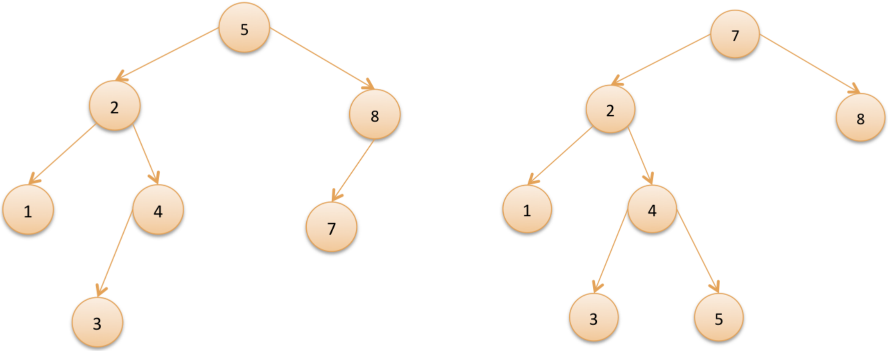


Tree Balance and Height



- **Observation:** it is not enough to balance only root, all nodes should be balanced.
- **The balancing condition**: the heights of all left and right subtrees differ by at most 1

Example: Balanced Trees?



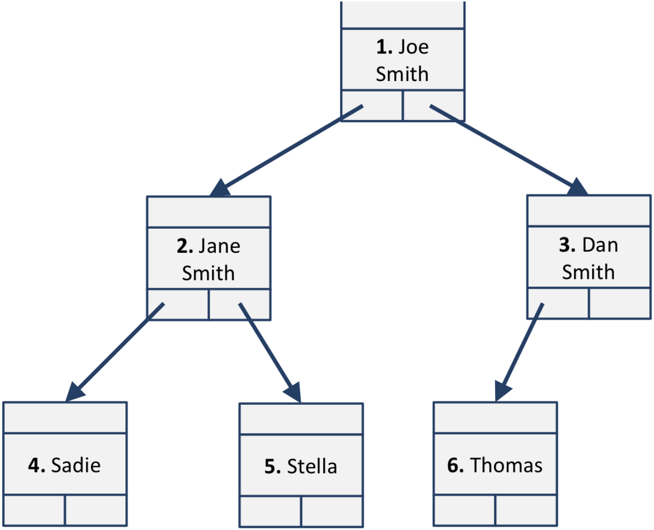
- The left tree is balanced.
- The right tree is not balanced. The height difference between nodes 2 and 8 is two.

Section 5

Tree Traversals

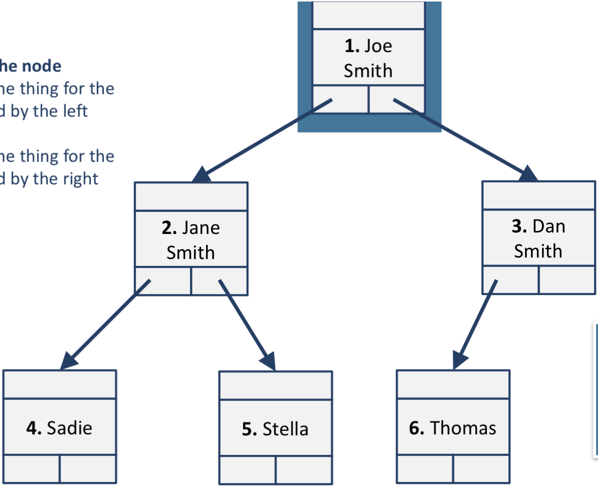
Tree Traversals

Challenge: write a recursive function that starts at the root, and prints out the data in each node of the tree below



Tree Traversals

1. **Print out the node**
2. Do the same thing for the tree rooted by the left child.
3. Do the same thing for the tree rooted by the right child.

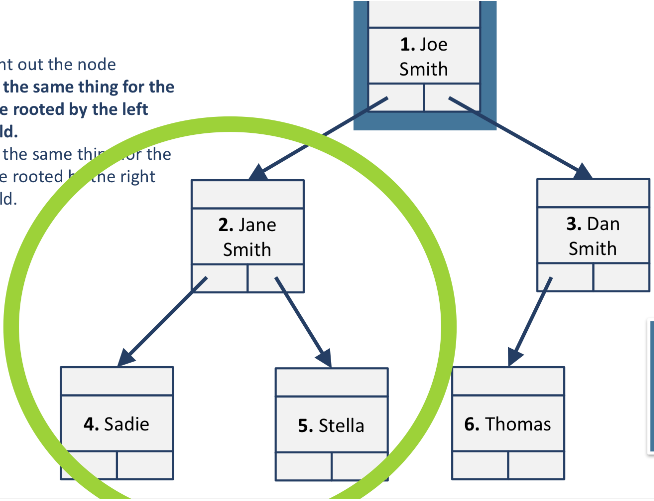



```
Joe Smith
```

Write a ***recursive*** function that starts at the root and prints out the data in each node of the tree.

Tree Traversals

1. Print out the node
2. **Do the same thing for the tree rooted by the left child.**
3. Do the same thing for the tree rooted by the right child.

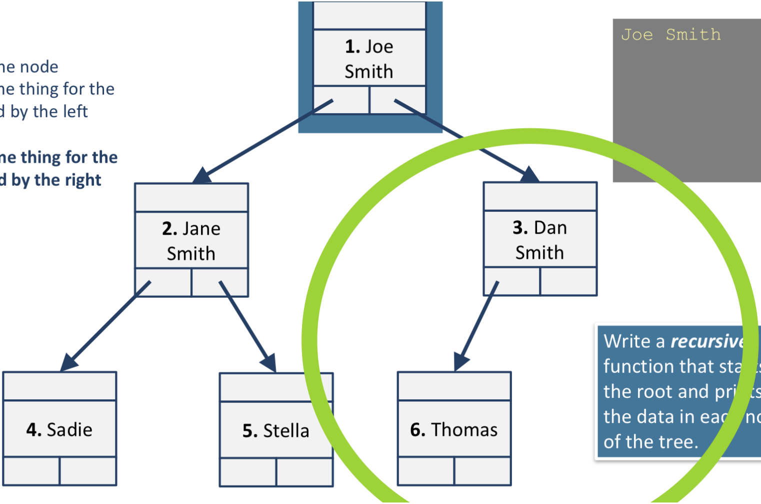



```
Joe Smith
```

Write a ***recursive*** function that starts at the root and prints out the data in each node of the tree.

Tree Traversals

1. Print out the node
2. Do the same thing for the tree rooted by the left child.
3. **Do the same thing for the tree rooted by the right child.**

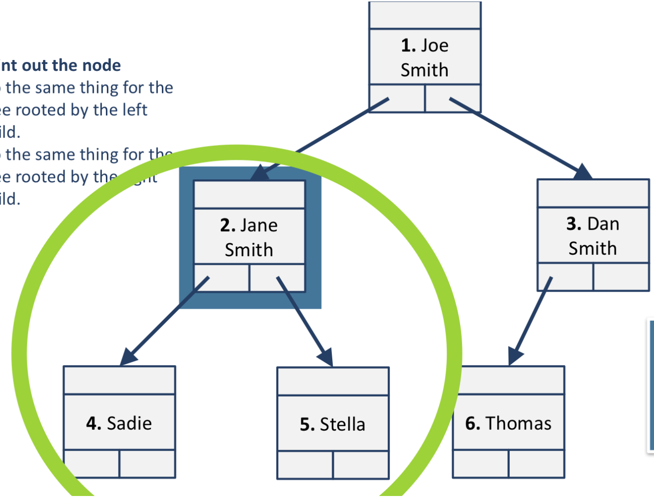



```
Joe Smith
```

Write a ***recursive*** function that sta[ts at] the root and pri[nts] out the data in eac[h] node of the tree.

Tree Traversals

1. **Print out the node**
2. Do the same thing for the tree rooted by the left child.
3. Do the same thing for the tree rooted by the right child.

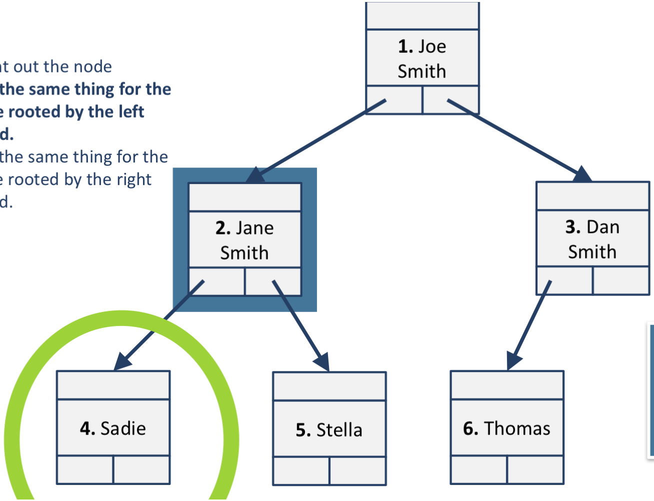



```
Joe Smith
Jane Smith
```

Write a ***recursive*** function that starts at the root and prints out the data in each node of the tree.

Tree Traversals

1. Print out the node
2. **Do the same thing for the tree rooted by the left child.**
3. Do the same thing for the tree rooted by the right child.

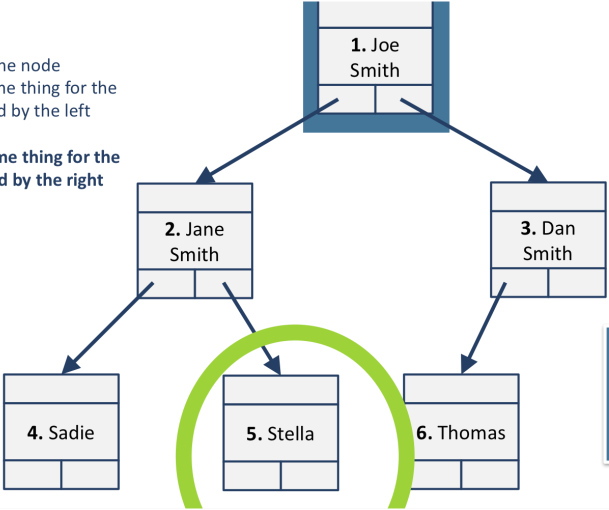



```
Joe Smith
Jane Smith
```

Write a ***recursive*** function that starts at the root and prints out the data in each node of the tree.

Tree Traversals

1. Print out the node
2. Do the same thing for the tree rooted by the left child.
3. **Do the same thing for the tree rooted by the right child.**

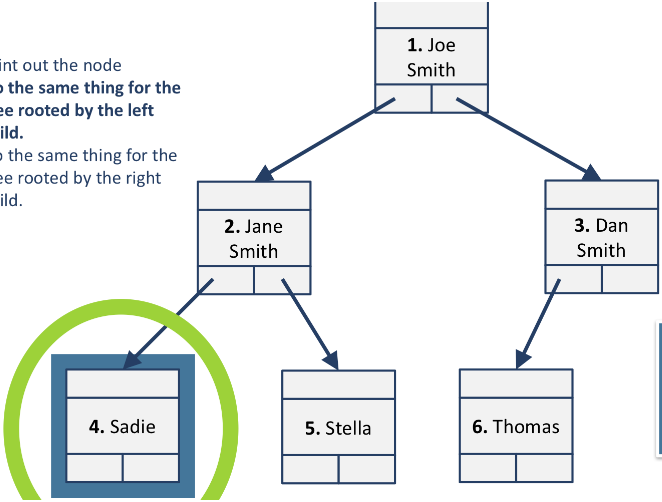



```
Joe Smith
Jane Smith
```

Write a ***recursive*** function that starts at the root and prints out the data in each node of the tree.

Tree Traversals

1. Print out the node
2. **Do the same thing for the tree rooted by the left child.**
3. Do the same thing for the tree rooted by the right child.

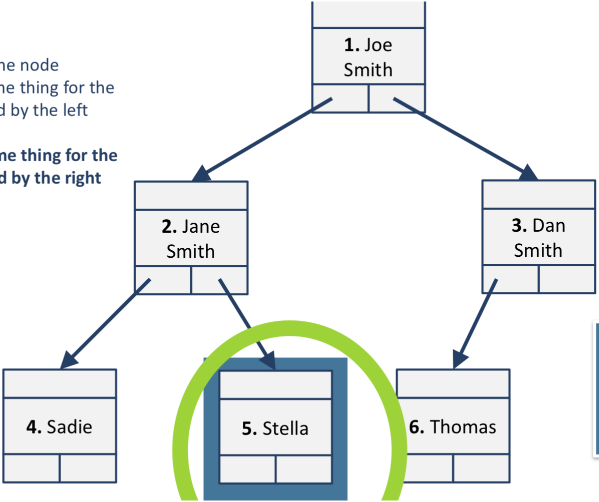



```
Joe Smith
Jane Smith
Sadie
```

Write a ***recursive*** function that starts at the root and prints out the data in each node of the tree.

Tree Traversals

1. Print out the node
2. Do the same thing for the tree rooted by the left child.
3. **Do the same thing for the tree rooted by the right child.**

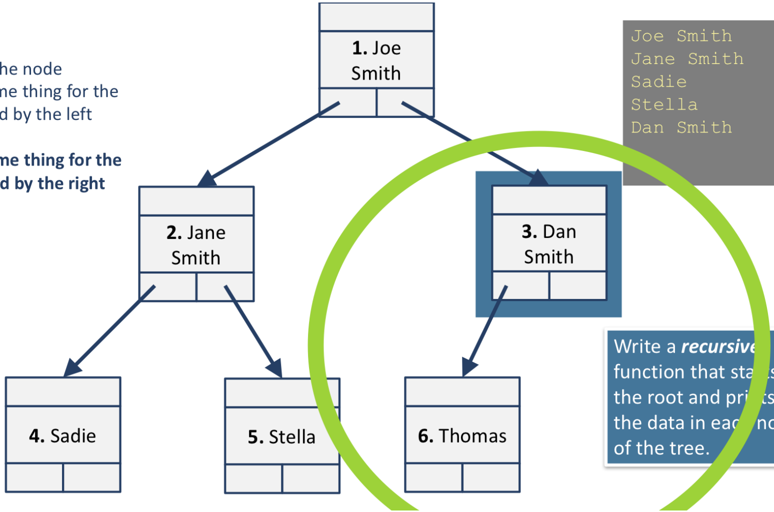



```
Joe Smith
Jane Smith
Sadie
Stella
```

Write a ***recursive*** function that starts at the root and prints out the data in each node of the tree.

Tree Traversals

1. Print out the node
2. Do the same thing for the tree rooted by the left child.
3. **Do the same thing for the tree rooted by the right child.**

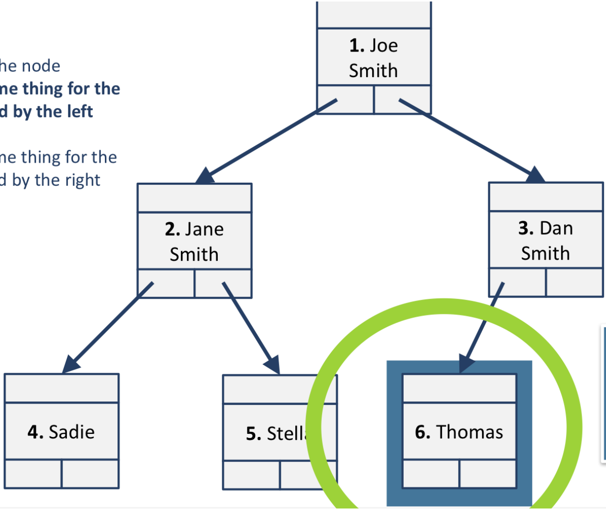



```
Joe Smith
Jane Smith
Sadie
Stella
Dan Smith
```

Write a ***recursive*** function that starts at the root and prints out the data in each node of the tree.

Tree Traversals

1. Print out the node
2. **Do the same thing for the tree rooted by the left child.**
3. Do the same thing for the tree rooted by the right child.




```
Joe Smith
Jane Smith
Sadie
Stella
Dan Smith
Thomas
```

Write a ***recursive*** function that starts at the root and prints out the data in each node of the tree.

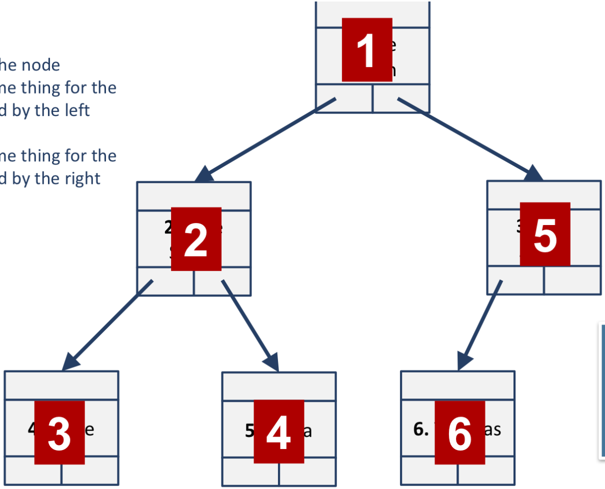
Tree Traversals

Summary:

```
void printTree(Node *root){
    printf("%s\n", root->data);
    printTree(root->leftChild);
    printTree(root->rightChild);
}
```

Tree Traversals

1. Print out the node
2. Do the same thing for the tree rooted by the left child.
3. Do the same thing for the tree rooted by the right child.

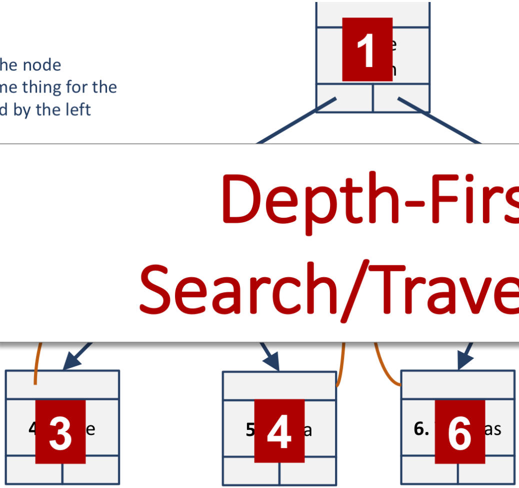



```
Joe Smith
Jane Smith
Sadie
Stella
Dan Smith
Thomas
```

Write a ***recursive*** function that starts at the root and prints out the data in each node of the tree.

Tree Traversals

1. Print out the node
2. Do the same thing for the tree rooted by the left child.
3. Do the sa tree roote child.



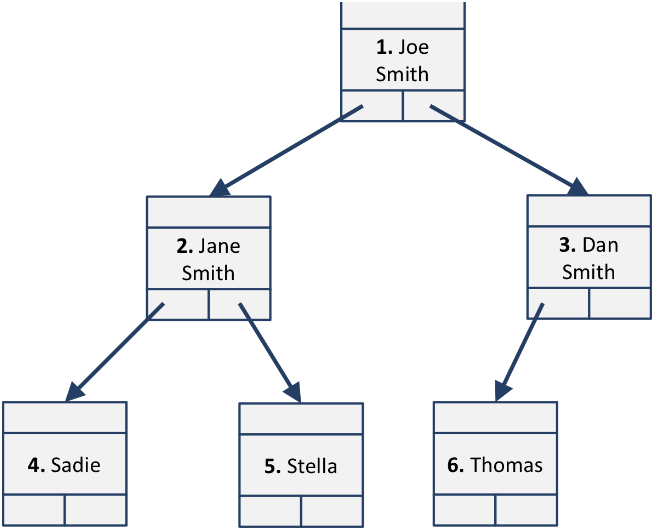
Joe Smith
Jane Smith
Sadie
Stella
Dan Smith

Depth-First Search/Traversal

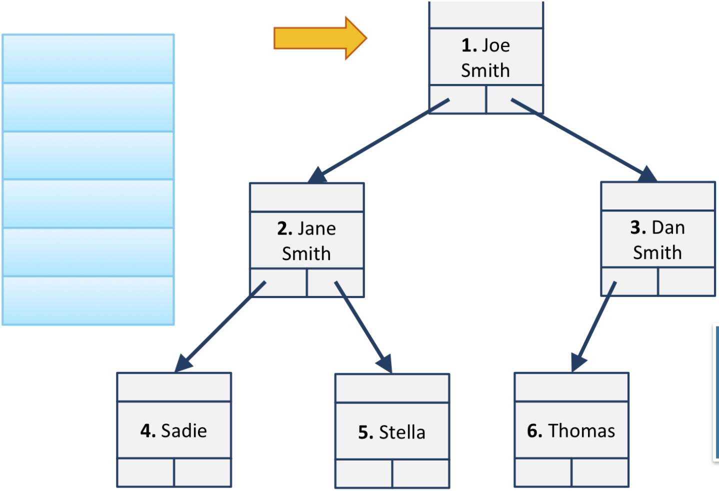
function that starts at the root and prints out the data in each node of the tree.

Tree Traversals

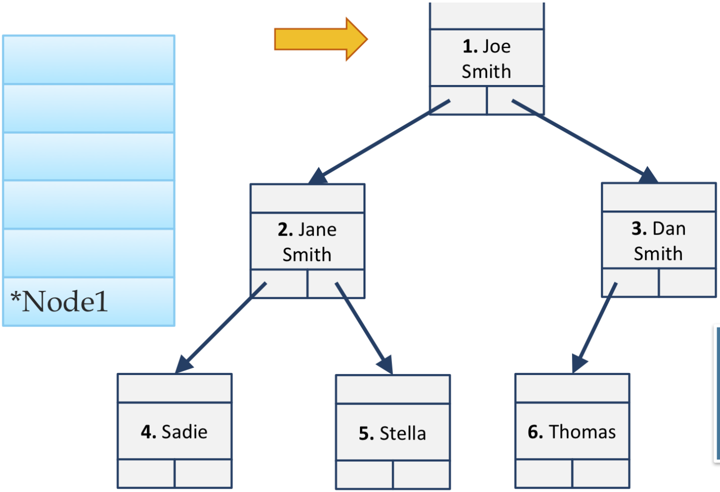
Challenge: write a non-recursive function that starts at the root, and prints out the data in each node of the tree below



Tree Traversals

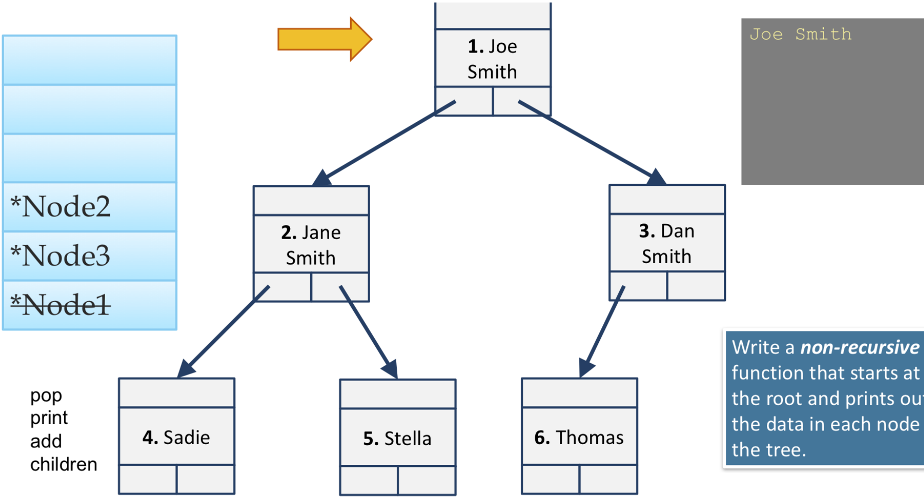


Write a ***non-recursive*** function that starts at the root and prints out the data in each node o the tree.



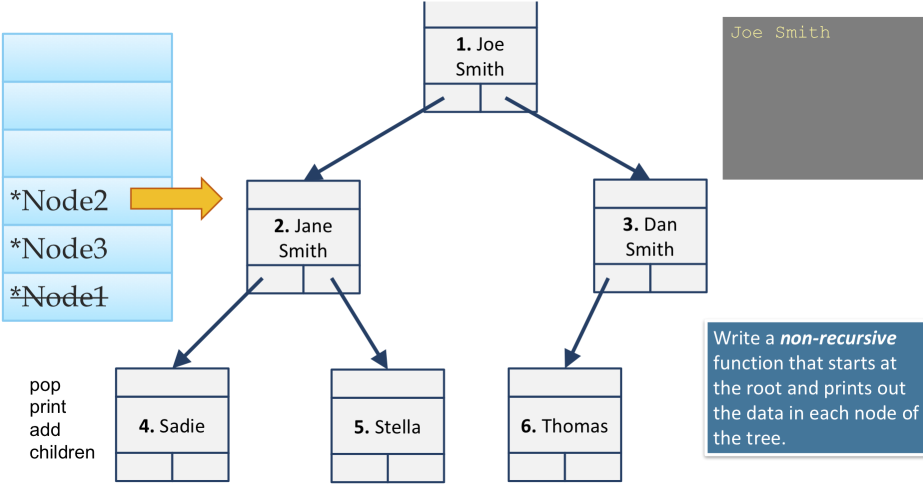
Write a ***non-recursive*** function that starts at the root and prints out the data in each node of the tree.

Tree Traversals



Write a ***non-recursive*** function that starts at the root and prints ou[t] the data in each node [of] the tree.

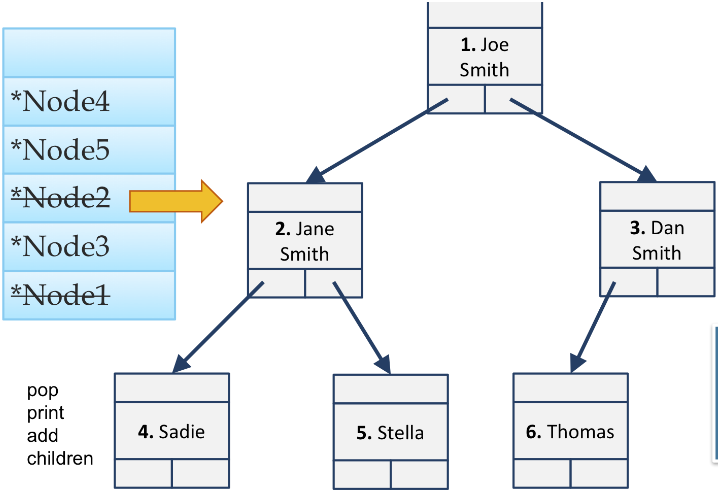
Tree Traversals



pop
print
add
children

Write a ***non-recursive*** function that starts at the root and prints out the data in each node of the tree.

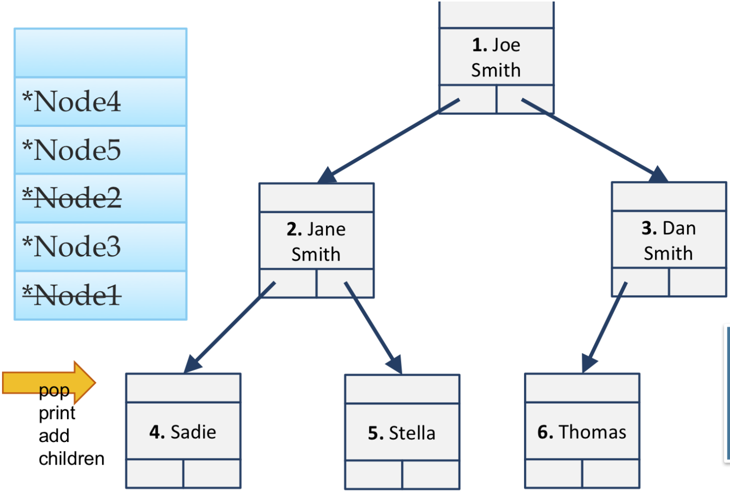
Tree Traversals




```
Joe Smith
Jane Smith
```

Write a ***non-recursive*** function that starts at the root and prints out the data in each node of the tree.

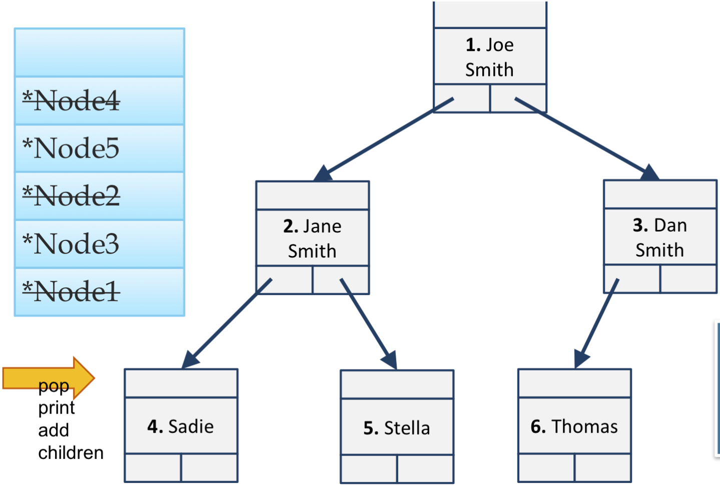
Tree Traversals




```
Joe Smith
Jane Smith
```

Write a ***non-recursive*** function that starts at the root and prints out the data in each node of the tree.

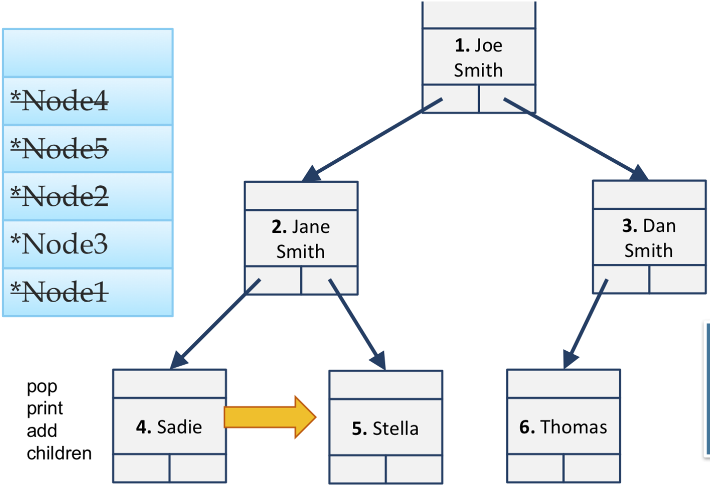
Tree Traversals




```
Joe Smith
Jane Smith
Sadie
```

Write a ***non-recursive*** function that starts at the root and prints out the data in each node of the tree.

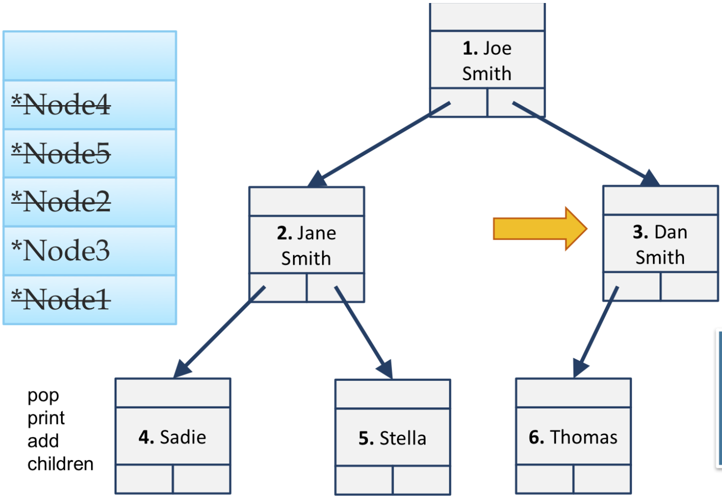
Tree Traversals




```
Joe Smith
Jane Smith
Sadie
Stella
```

Write a ***non-recursive*** function that starts at the root and prints out the data in each node of the tree.

Tree Traversals

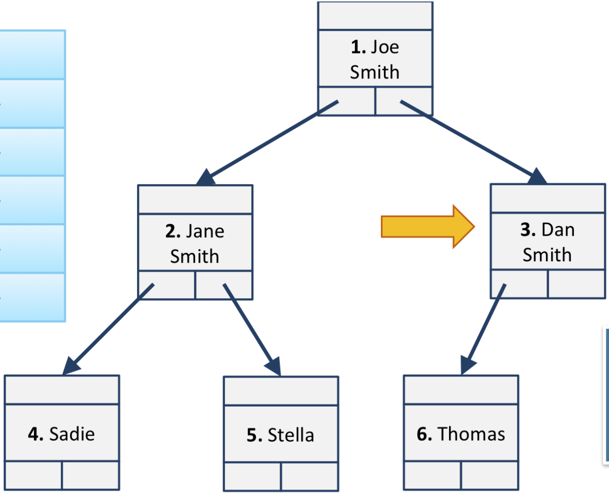


Joe Smith
Jane Smith
Sadie
Stella

Write a ***non-recursive*** function that starts at the root and prints out the data in each node of the tree.

Tree Traversals

- *Node6
- ~~*Node4~~
- ~~*Node5~~
- ~~*Node2~~
- ~~*Node3~~
- ~~*Node1~~

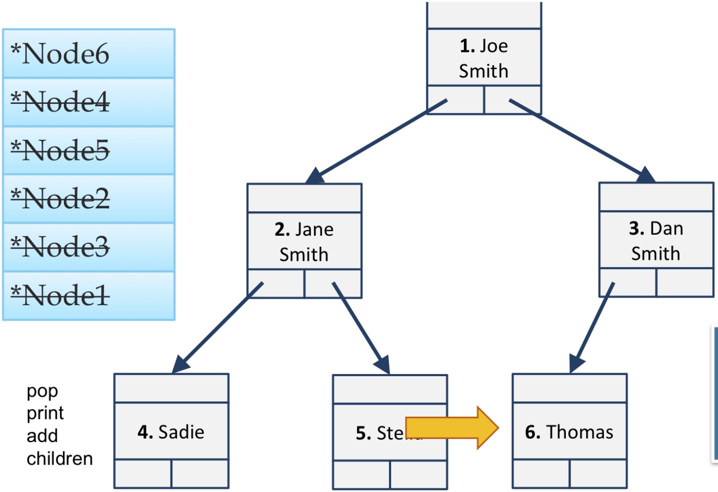


pop
print
add
children

```
Joe Smith
Jane Smith
Sadie
Stella
Dan Smith
```

Write a ***non-recursive*** function that starts at the root and prints out the data in each node of the tree.

Tree Traversals




```
Joe Smith
Jane Smith
Sadie
Stella
Dan Smith
Thomas
```

Write a ***non-recursive*** function that starts at the root and prints out the data in each node of the tree.

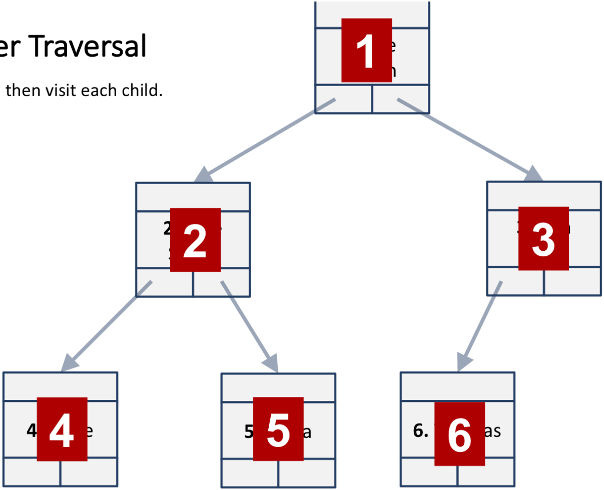
Tree Traversals

```
stack s;
push(s, root)
Node* curNode;
while (!isEmpty(s)){
        curNode = pop(s);
        print(curNode);
        push(s, curNode->right);
        push(s, curNode->left);
}
```

Tree Traversals

Pre-order Traversal

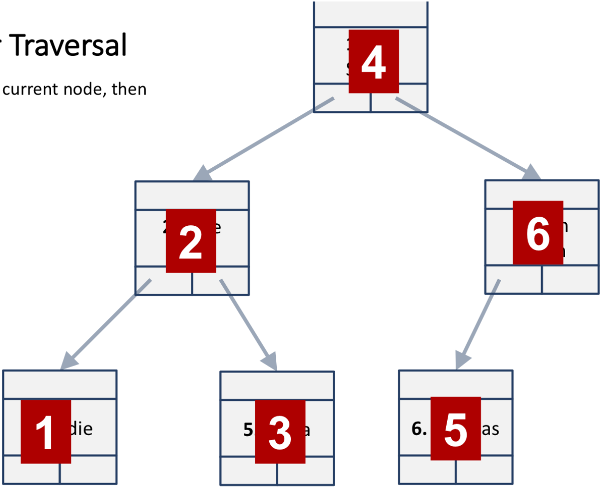
Visit the node, then visit each child.



Tree Traversals

In-order Traversal

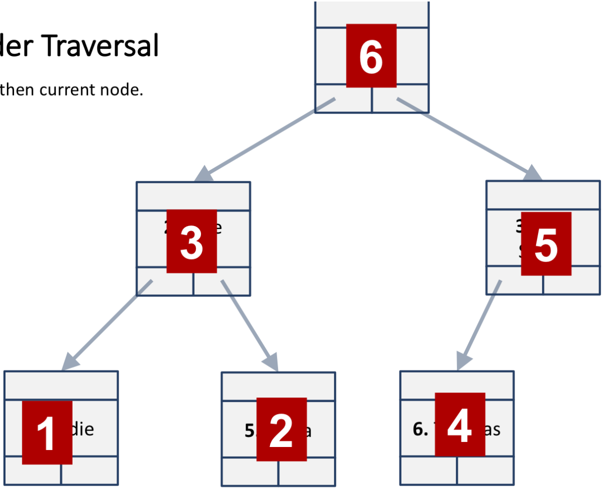
Visit left child, current node, then right child.



Tree Traversals

Post-order Traversal

Visit children, then current node.

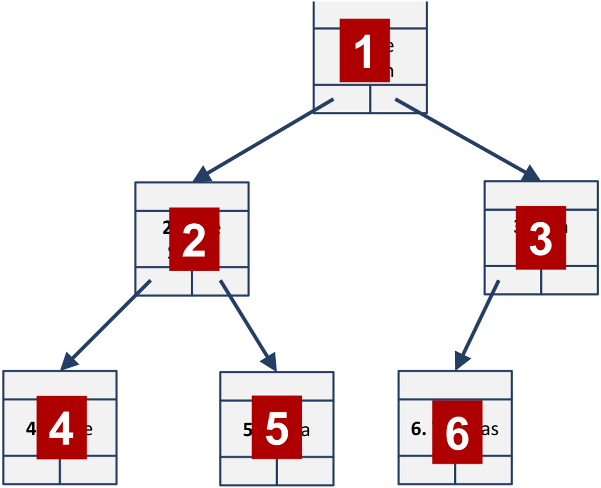


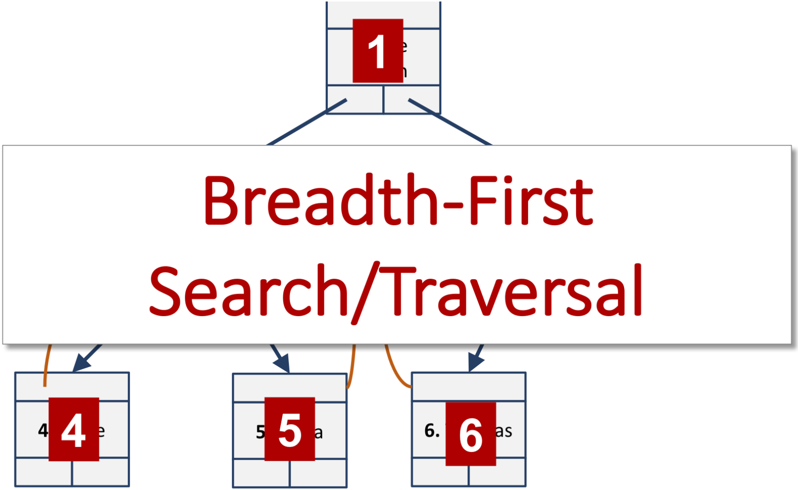
Challenge:

Consider how to modify this algorithm to produce a post-order printing of the nodes.

HINT: You might need to add a helper variable somewhere.

Tree Traversals

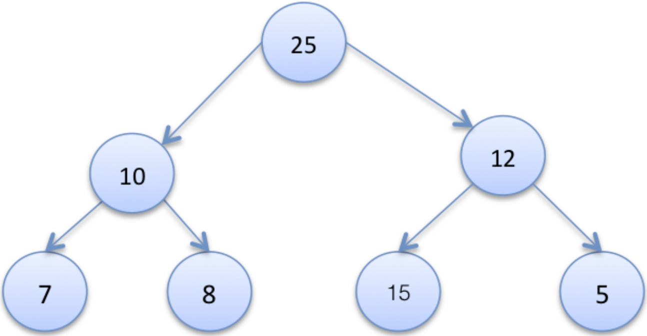



1
Breadth-First
Search/Traversal
4
5
6

BFS Example

Find element with value 15 in the tree below.

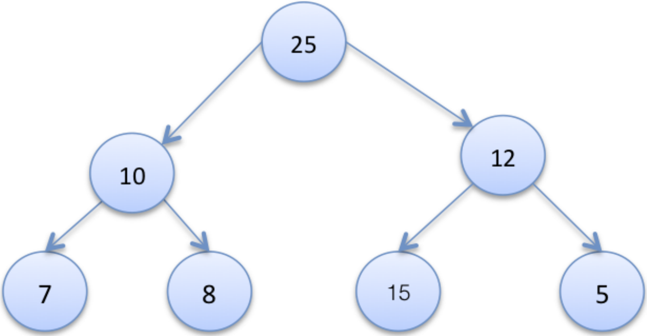
- BFS: traverse all of the nodes on the same level first, and then move on to the next (lower) level



BFS Example

Find element with value 15 in the tree below using BFS.

- BFS: traverse all of the nodes on the same level first, and then move on to the next (lower) level

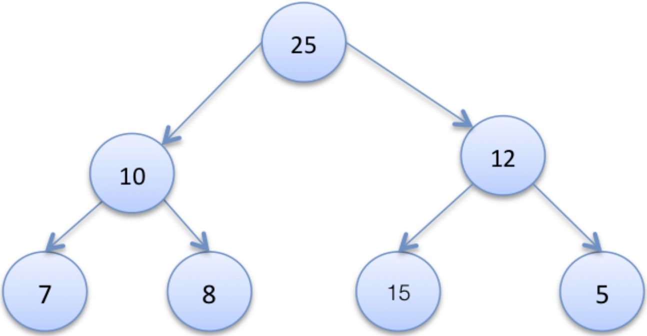


25 – 10 – 12 – 7 – 8 – 15 – 5

DFS Example

Find element with value 15 in the tree below using DFS.

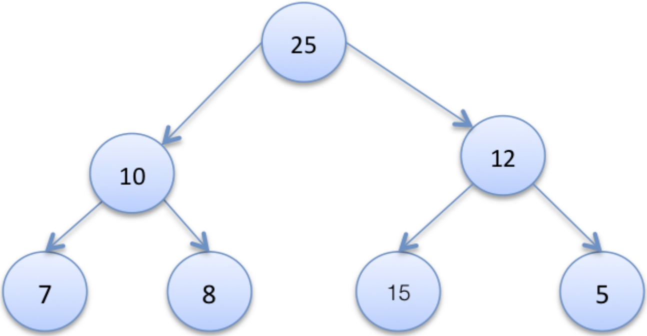
- DFS: traverse one side of the tree all the way to the leaves, followed by the other side



DFS Example

Find element with value 15 in the tree below using DFS.

- DFS: traverse one side of the tree all the way to the leaves, followed by the other side

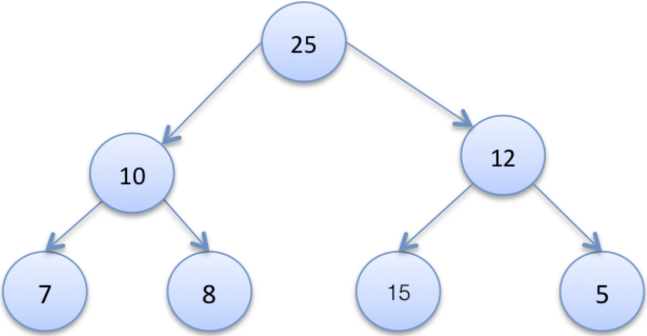


25 – 10 – 7 –8 – 12 – 15 – 5

Tree Traversals Example

Traverse the tree below, using:

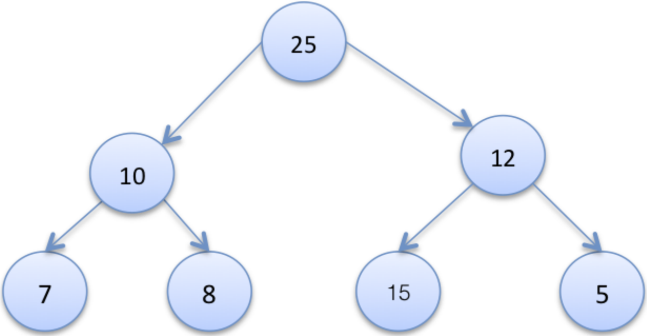
- Pre-order traversal: 25 – 10 – 7 – 8 – 12 – 15 – 5



Tree Traversals Example

Traverse the tree below, using:

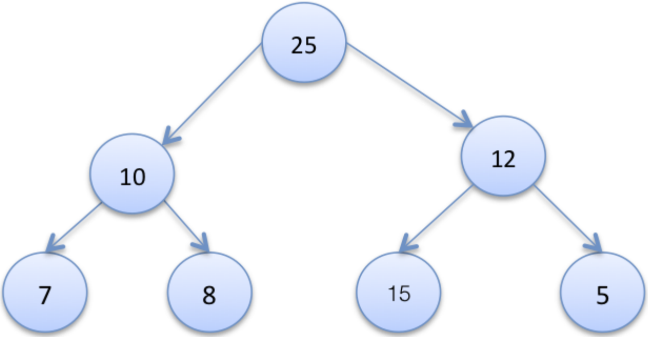
- Pre-order traversal: 25 – 10 – 7 – 8 – 12 – 15 – 5
- In-order traversal: 7 – 10 – 8 – 25 – 15 – 12 – 5



Tree Traversals Example

Traverse the tree below, using:

- Pre-order traversal: 25 – 10 – 7 – 8 – 12 – 15 – 5
- In-order traversal: 7 – 10 – 8 – 25 – 15 – 12 – 5
- Post-order traversal: 7 – 8 –10 – 15 –5 – 12 – 25



Section 6

Introduction to Graphs

What is a Graph?

Formal Definition:

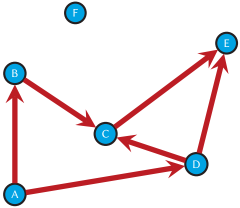
- A graph G is a pair (V, E) where
- V is a set of vertices or nodes
- E is a set of edges that connect vertices

Simply put:

- A graph is a collection of nodes (vertices) and edges
- Linked lists, trees, and heaps are all special cases of graphs

Here is a graph $G = (V, E)$

- Each edge is a pair (v_1, v_2), where v_1, v_2 are vertices in V
  - $V = \{A, B, C, D, E, F\}$
  - $E = \{(A, B), (A, D), (B, C), (C, D), (C, E), (D, E)\}$

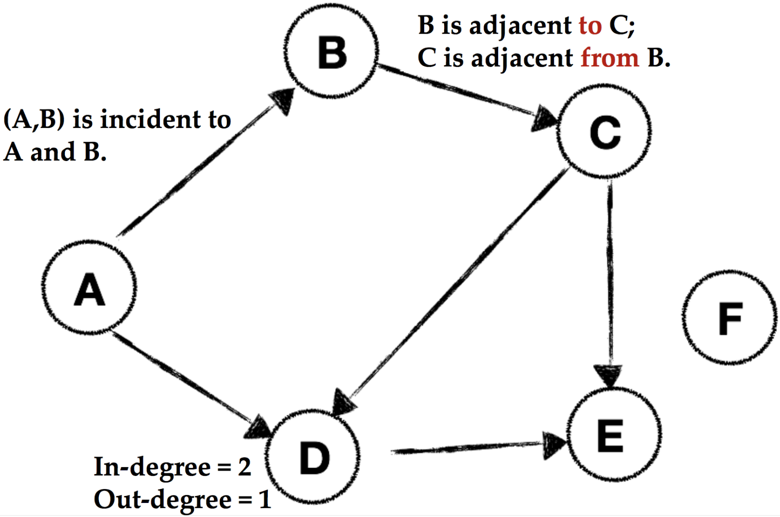


Terminology: Undirected Graph

- Two vertices u and v are ***adjacent*** in an undirected graph G if $\{u, v\}$ is an edge in G
  - edge $e = \{u, v\}$ is ***incident*** with vertex u and vertex v
- The ***degree*** of a vertex in an undirected graph is the number of edges incident with it
  - a self-loop counts twice (both ends count)
  - denoted with $deg(v)$

Terminology: Directed Graph

- Vertex u is ***adjacent to*** vertex v in a directed graph G if (u, v) is an edge in G
  - vertex u is the initial vertex of (u, v)
- Vertex v is ***adjacent from*** vertex u
  - vertex v is the ***terminal*** (or end) vertex of (u, v)
- Degree
  - ***in-degree*** is the number of edges with the vertex as the terminal vertex
  - ***out-degree*** is the number of edges with the vertex as the initial vertex

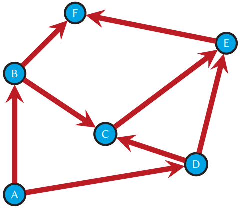

B is adjacent to C;
C is adjacent from B.
(A,B) is incident to
A and B.
In-degree = 2
Out-degree = 1
A
B
C
D
E
F

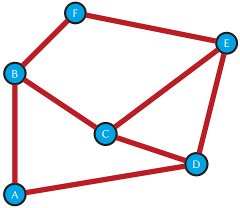
Kinds of Graphs

- directed vs undirected
- weighted vs unweighted
- simple vs non-simple
- sparse vs dense
- cyclic vs acyclic
- labeled vs unlabeled

Directed vs Undirected

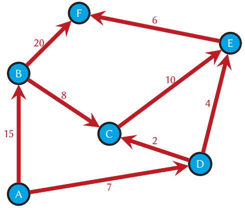
- Undirected if edge (x, y) implies edge (y, x).
  - otherwise directed
- Roads between cities are usually undirected (go both ways)
- Streets in cities tend to be directed (one-way)





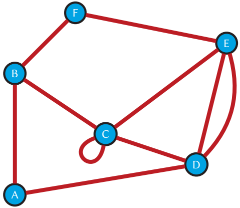
Weighted vs Unweighted

- Each edge or vertex is assigned a numerical value (weight).
- A road network might be labeled with:
  - length
  - drive-time
  - speed-limit
- In an unweighted graph, there is no distinction between edges.



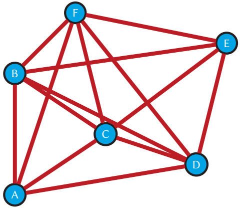
Simple vs Not simple

- Some kinds of edges make working with graphs complicated
- A ***self-loop*** is an edge (x, x) (one vertex).
- An edge (x, y) is a ***multiedge*** if it occurs more than once in a graph.



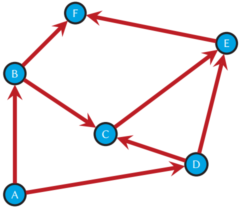
Sparse vs Dense

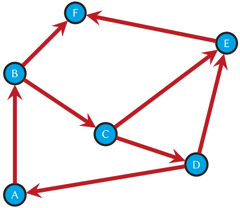
- Graphs are sparse when a small fraction of vertex pairs have edges between them
- Graphs are dense when a large fraction of vertex pairs have edges
- There's no formal distinction between sparse and dense



Cyclic vs Acyclic

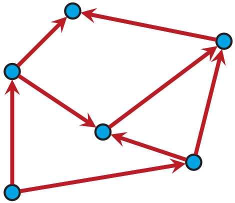
- An ***acyclic*** graph contains no cycles
- A ***cyclic*** graph contains a cycle
- Trees are *connected, acyclic, undirected* graphs
- Directed acyclic graphs are called ***DAGs***





Labeled vs Unlabeled

- Each vertex is assigned a unique name or identifier in a ***labeled*** graph
  - In an unlabeled graph, there are no named nodes
- Graphs usually have names—e.g., city names in a transportation network
- We might ignore names in graphs to determine if they are isomorphic (similar in structure)



Section 7

Graph Representations

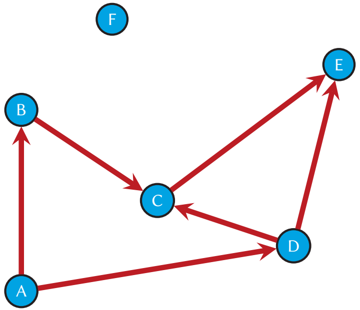
Graph Representations

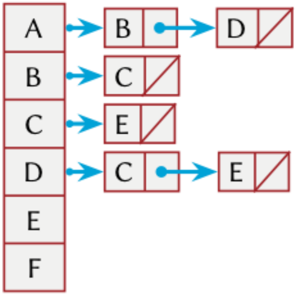
Two ways to represent a graph in code:

- **Adjacency List**
  - A list of nodes
  - Every node has a list of adjacent nodes
- **Adjacency Matrix**
  - A matrix has a column and a row to represent every node
  - All entries are 0 by default
  - An entry $G[u, v]$ is 1 if there is an edge from node u to v

Adjacency List

For each v in V, $L(v)$ = list of w such that (v, w) is in E:





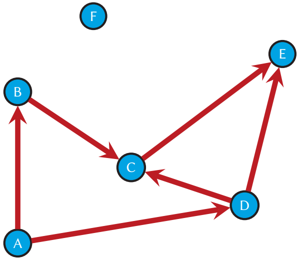
Storage space:

$$a|V| + b|E|$$

$$a = \text{sizeof}(node)$$

$$b = \text{sizeof}(\text{ linked list element})$$

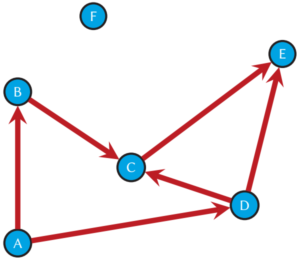
Adjacency Matrix



$$\begin{array}{c|cccccc} & A & B & C & D & E & F \\ \hline A & 0 & 1 & 0 & 1 & 0 & 0 \\ B & 1 & 0 & 1 & 0 & 0 & 0 \\ C & 0 & 1 & 0 & 1 & 1 & 0 \\ D & 1 & 0 & 1 & 0 & 1 & 0 \\ E & 0 & 0 & 1 & 1 & 0 & 0 \\ F & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Storage space: $|V|^2$

Adjacency Matrix



$$\begin{array}{cc} & \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \\ E \\ F \end{array} & \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Storage space: $|V|^2$

Does this matrix represent a directed or undirected graph?

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?

1. adjacency matrix

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?

1. adjacency matrix

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?

1. adjacency matrix
2. adjacency list

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?

1. adjacency matrix
2. adjacency list

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?
4. Less memory on big graphs?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?
4. Less memory on big graphs?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)
4. adjacency matrices (a little)

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?
4. Less memory on big graphs?
5. Edge insertion or deletion?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)
4. adjacency matrices (a little)

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?
4. Less memory on big graphs?
5. Edge insertion or deletion?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)
4. adjacency matrices (a little)
5. adjacency matrices O(1) vs O(d)

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?
4. Less memory on big graphs?
5. Edge insertion or deletion?
6. Faster to traverse the graph?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)
4. adjacency matrices (a little)
5. adjacency matrices O(1) vs O(d)

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?
4. Less memory on big graphs?
5. Edge insertion or deletion?
6. Faster to traverse the graph?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)
4. adjacency matrices (a little)
5. adjacency matrices O(1) vs O(d)
6. adjacency list

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?
4. Less memory on big graphs?
5. Edge insertion or deletion?
6. Faster to traverse the graph?
7. Better for most problems?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)
4. adjacency matrices (a little)
5. adjacency matrices O(1) vs O(d)
6. adjacency list

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?
4. Less memory on big graphs?
5. Edge insertion or deletion?
6. Faster to traverse the graph?
7. Better for most problems?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)
4. adjacency matrices (a little)
5. adjacency matrices O(1) vs O(d)
6. adjacency list
7. adjacency list

Analyzing Graph Algorithms

- Space and time are analyzed in terms of:
  - Number of vertices $m = |V|$
  - Number of edges $n = |E|$
- Aim for polynomial running times.
- But: is $O(m^2)$ or $O(n^3)$ a better running time?
  - depends on what the relation is between n and m
  - the number of edges m can be at most $n^2 \leq n^2$.
  - connected graphs have at least $m \geq n - 1$ edges
- Stil do not know which of two running times (such as m^2 and n^3) are better,
- Goal: implement the basic graph search algorithms in time $O(m + n)$.
  - This is linear time, since it takes $O(m + n)$ time simply to read the input.
- Note that when we work with connected graphs, a running time of $O(m + n)$ is the same as $O(m)$, since $m \geq n - 1$.

Section 8

Graph Traversals

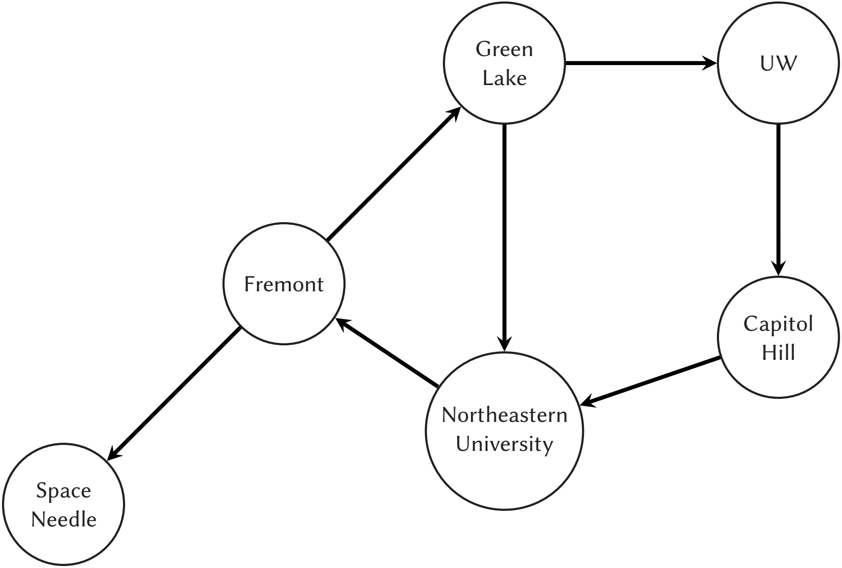
Graph Traversals

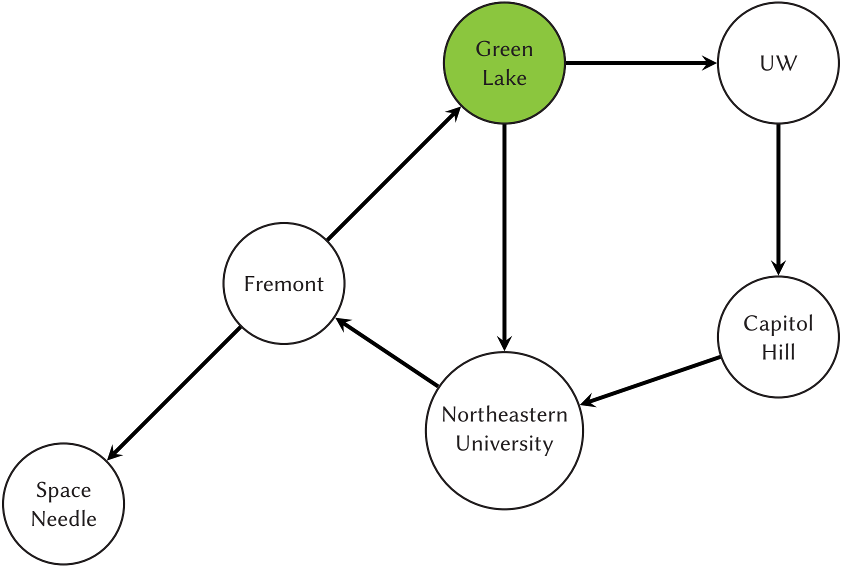
Two basic traversals:

- Breadth First Search (BFS)
- Depth First Search (DFS)

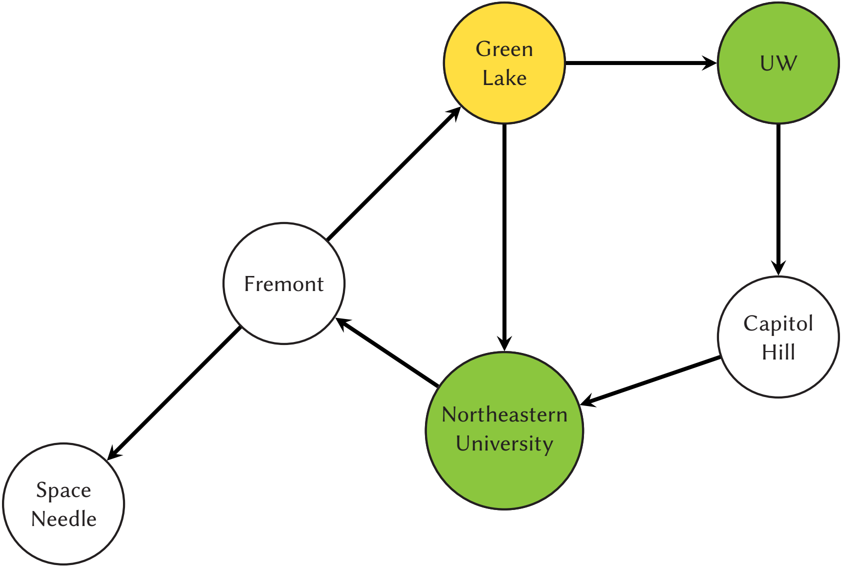
BFS

Example...

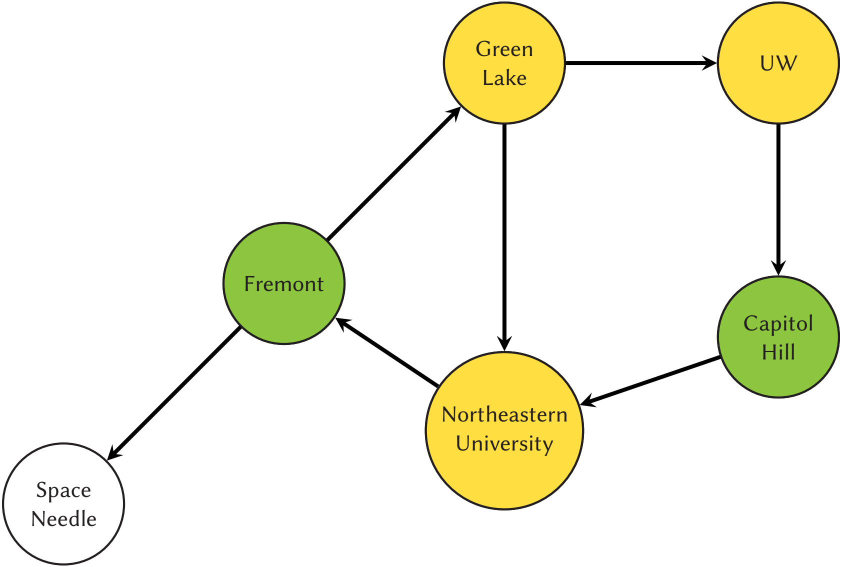

Green Lake
UW
Fremont
Capitol Hill
Northeastern University
Space Needle



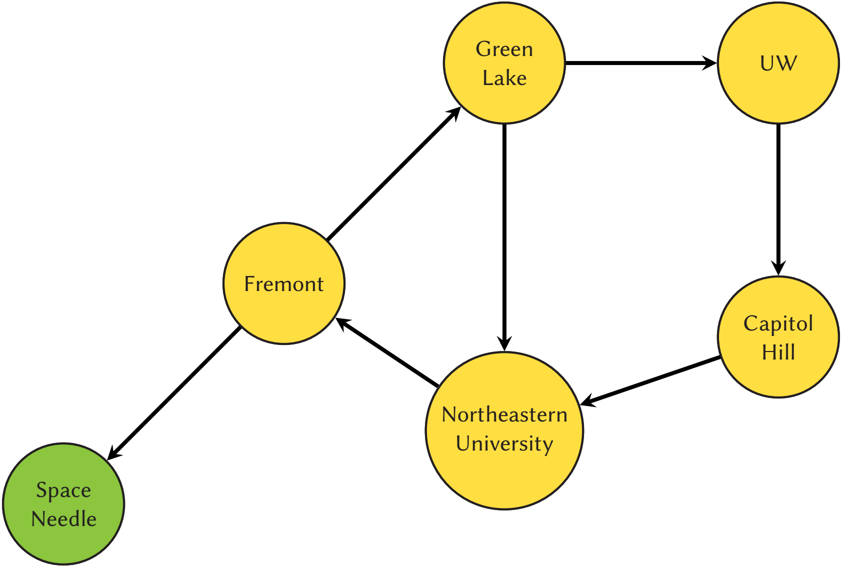
What's the best way for me to get from Green Lake to Space Needle?



What's the best way for me to get from Green Lake to Space Needle?



What's the best way for me to get from Green Lake to Space Needle?



What's the best way for me to get from Green Lake to Space Needle?

BFS: The Algorithm

- Start at the start.
- Look at all the neighbors. Are any of them the destination?
- If no:
  - Look at all the neighbors of the neighbors. Are any of them the destination?
  - Look at all the neighbors of the neighbors of the neighbors. Are any of them the destination?

BFS: Runtime

- If you search the entire network, you traverse each edge at least once: $O(|E|)$
  - That is, O(number of edges)
- Keeping a queue of who to visit in order.
  - Add single node to queue: $O(1)$
  - For all nodes: O(number of nodes)
  - $O(|V|)$
- Together, it's $O(V + E)$

Using

- Assuming we can add and remove from our "pending" DS in $O(1)$ time, the entire traversal is $O(|E|)$
- Traversal order depends on what we use for our pending DS.
  - Stack : DFS
  - Queue: BFS
- These are the main traversal techniques in CS, but there are others!

- Depth first search needs to check which nodes have been output or else it can get stuck in loops.
- In a connected graph, a BFS will print all nodes, but it will repeat if there are cycles and may not terminate
- As an aside, in-order, pre-order and postorder traversals only make sense in binary trees, so they aren't important for graphs. However, we do need some way to order our out-vertices (left and right in BST).

- Breadth-first always finds shortest length paths, i.e., “optimal solutions”
- Better for “what is the shortest path from x to y”
  - But depth-first can use less space in finding a path
- If longest path in the graph is p and highest out- degree is d then DFS stack never has more than $d * p$ elements
- But a queue for BFS may hold $O(|V|)$ nodes

BFS vs DFS: Problems

BFS Applications

- Connected components
- Two-coloring graphs

DFS Applications

- Finding cycles
- Topological Sorting
- Strongly Connected Components

Section 9

Path Finding in a Graph

Single-Source Shortest Path

Input Directed graph with non-negative weighted edges, a starting node s and a destination node d

Problem Starting at the given node s, find the path with the lowest total edge weight to node d

Example A map with cities as nodes and the edges are distances between the cities. Find the shortest distance between city 1 and city 2.

Djikstra's Algorithm: Overview

- Find the "cheapest" node— the node you can get to in the shortest amount of time.
- Update the costs of the neighbors of this node.
- Repeat until you've done this for each node.
- Calculate the final path.

Djikstra's Algorithm: Formally

DJIKSTRA(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u =$ EXTRACT-MIN(Q)
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX (u, v, w)

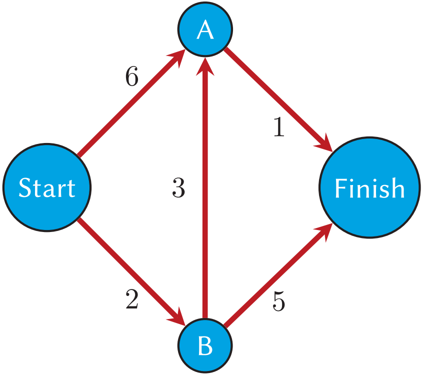
```
DJIKSTRA(G, w, s)
  ▷ G is a graph
  ▷ w is the weighting function such that w(u, v) returns the weight of the
  ▷ s is the starting node
  for each vertex u ∈ G
      u.d = w(s, u) ▷ where w(s, u) = ∞ if there is no edge (s, u).
  S = ∅ ▷ Nodes we know the distance to
  Q = G.V ▷ min-PriorityQueue starting with all our nodes, ordered by dis
  while Q ≠ ∅
      u = EXTRACT-MIN(Q) ▷ Greedy step: get the closest node
      S = S ∪ {u} ▷ Set of nodes that have shortest-path-distance found
      for each vertex v ∈ G.Adj[u]
          RELAX (u, v, w)
```

```
RELAX(u, v, w)
  ▷ u is the start node
  ▷ v is the destination node
  ▷ w is the weight function
```

Djikstra's: A walkthrough

- Find the "cheapest" node— the node you can get to in the shortest amount of time.
- Update the costs of the neighbors of this node.
- Repeat until you've done this for each node.
- Calculate the final path.

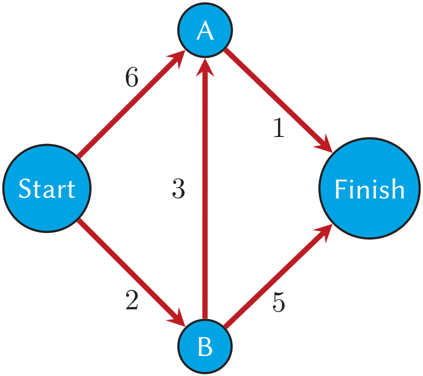
Breadth First Search: distance = 7



Step 1: Find the cheapest node

1. Should we go to A or B?
   - Make a table of how long it takes to get to each node from this node.
   - We don't know how long it takes to get to Finish, so we just say infinity for now.

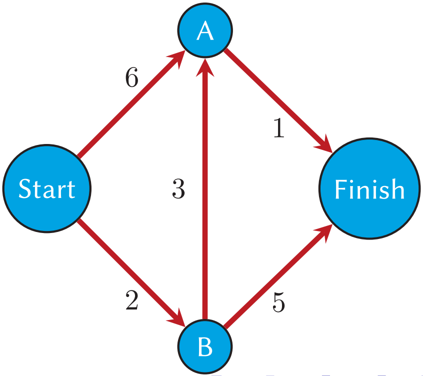
Node	Time to Node
A	6
B	2
Finish	∞



Step 2: Take the next step

1. Calculate how long it takes to get (from Start) to B's neighbors by following an edge from B
   - We chose B because it's the fastest to get to.
   - Assume we started at Start, went to B, and then now we're updating Time to Nodes.

Node	Time to Node
A	$\cancel{6}5$
B	2
Finish	$\cancel{\infty}$ 7



Step 3: Repeat!

1. Find the node that takes the least amount of time to get to.
   - We already did B, so let's do A.
   - Update the costs of A's neighbors
     - Takes 5 to get to A; 1 more to get to Finish

Node	Time to Node
A	$\cancel{6}5$
B	2
Finish	$\cancel{7}6$