C, Algorithms and Systems

Lecture 5: Trees and Graphs

CS 5006/7
Spring 2019

Seattle

Adrienne Slaughter, Ph.D.
ahslaughter@northeastern.edu

Joe Buck
j.buck@northeastern.edu

Today

- A few notes on Strings
- List your questions
- Quick review of trees, graphs terminology
- Implementing trees
- Using Valgrind

AM I RECORDING????

Coming Up

- **Feb 11:** More trees/graphs
- **Feb 18:** Intro to C Libraries, fun sorts
- **Feb 25:** 5006 Final

A note on Grades

- The intention was to grade 5006 & 5007 as one course.
  - 70% assignments
  - 15% midterm
  - 15% final project
- Due to forces out of our control, you get a different grade for 5006 and 5007
  - We need to assign grades after the midterm
  - 5006 grade based on Assignments 1-5 and the midterm
  - 5007 grade based on Assignments 6-8 and final project
- Yes, it's unfair and we don't like it.

A note on Grades

- 5006 Grade:
  - 75% assignments
  - 25% midterm/"final"
- 5007 Grade:
  - 75% assignments
  - 25% final project

Assignments

- From now on, Assignments go out on Thursdays officially, and all due the following Friday.
- PUSH YOUR CODE. NOW.

EMACS TIP OF THE DAY

Adding line numbers to your buffer

- `M-x linum-mode`
- OR
- `M-x global-linum-mode`

- To enable it ALL THE TIME:
  - Open your .emacs file:
    - `emacs ~/.emacs`
  - Add the following line to the file:
    - `(global-linum-mode)`

A NOTE ON STRINGS

Difference between chars and strings

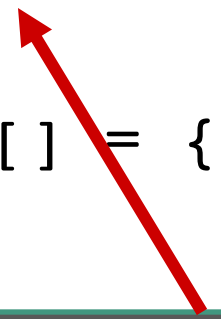
- ‘a’, ‘b’
  - Chars have single quotes
  - Represented by an **`int`** value
  - Includes letters and special chars
    - Special chars: things like ‘\n’ (newline), ‘\0’ (null), ’\t’ (tab)
- “a”, “b”
  - Strings have double quotes
  - Represented by an array of chars ending with a NULL character (‘\0’)

Strings

- Strings are arrays of chars terminated by a \0.
- When you initialize a string, you get back a pointer to a char array
- Unlike regular arrays, you can use the existence of the \0 to find the end of the array
- An array of strings turns out to be an array of pointers to arrays of chars.
- Because an array is just a pointer to the first element of the array, an array of strings is a pointer to a pointer.

Declaring Strings

```
// creates an array of chars [‘b’, ‘l’, ‘u’, ‘e’, ‘\0’]
char color[] = “blue”;
char *colorPtr = “blue”;  // creates a variable of type char pointer
char color[] = {‘b’, ‘l’, ‘u’, ‘e’, ‘\0’}
```



Note:

Declaring a string this way can make it read-only. That means if you try to modify it later, you can’t do it by modifying the characters in the underlying array.
(see the demo code)

Declaring an array to eventually hold a string:

** Make sure it’s long enough to contain all chars AND the null character!!**

Demo with `string.c`

- Round 1:
  - Declare 3 strings as noted in slide
  - Print all three out
- Round 2:
  - Change str[0] for each
  - Run, and see segfault for first
- Round 3:
  - Print out the rest
- Round 4:
  - string2 can point to any other string

TREES

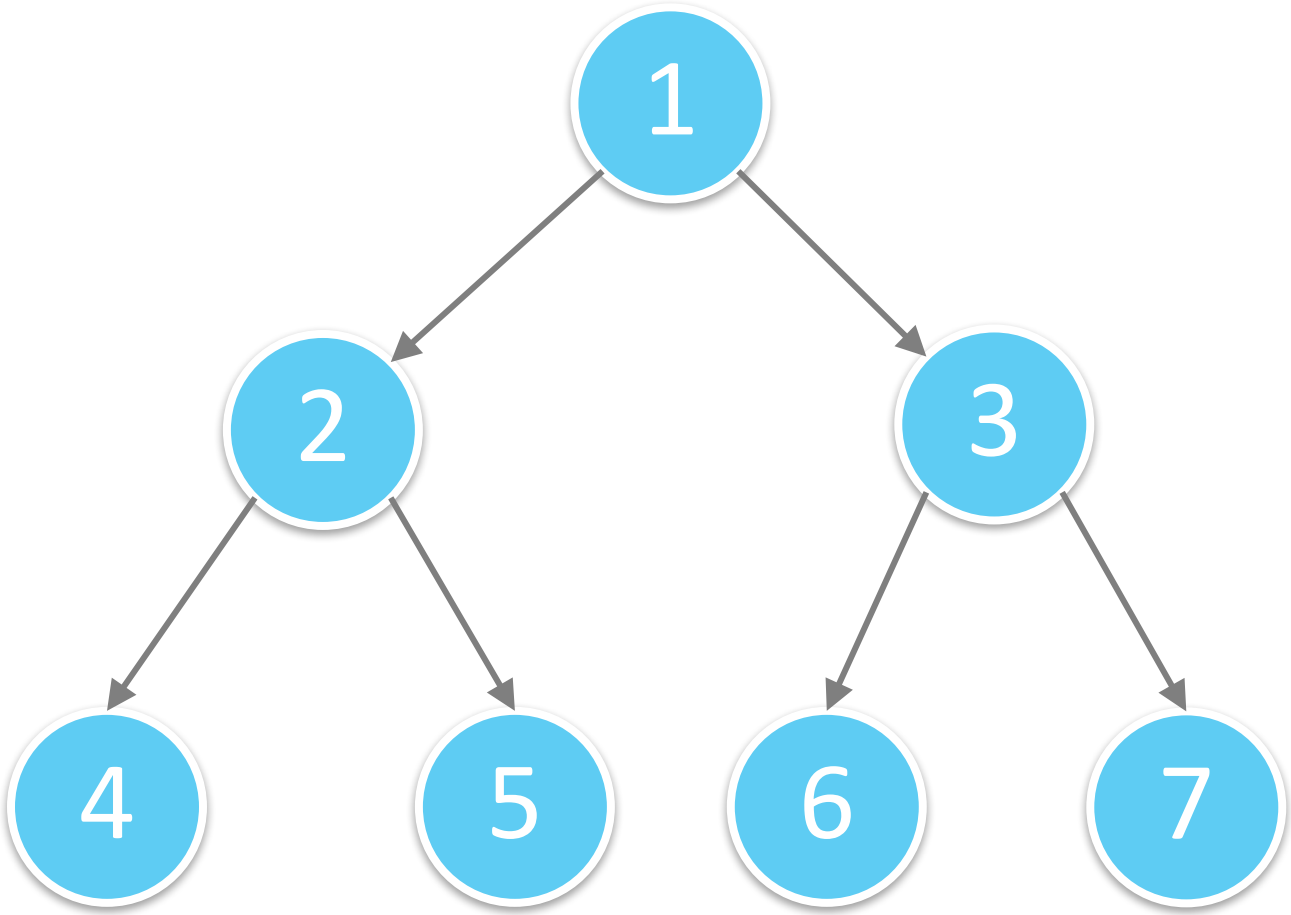
Sorted arrays provide fast search.

Doubly-linked list provides fast/flexible update (insert/delete).

Can something provide **fast search** *AND* **flexible update?**

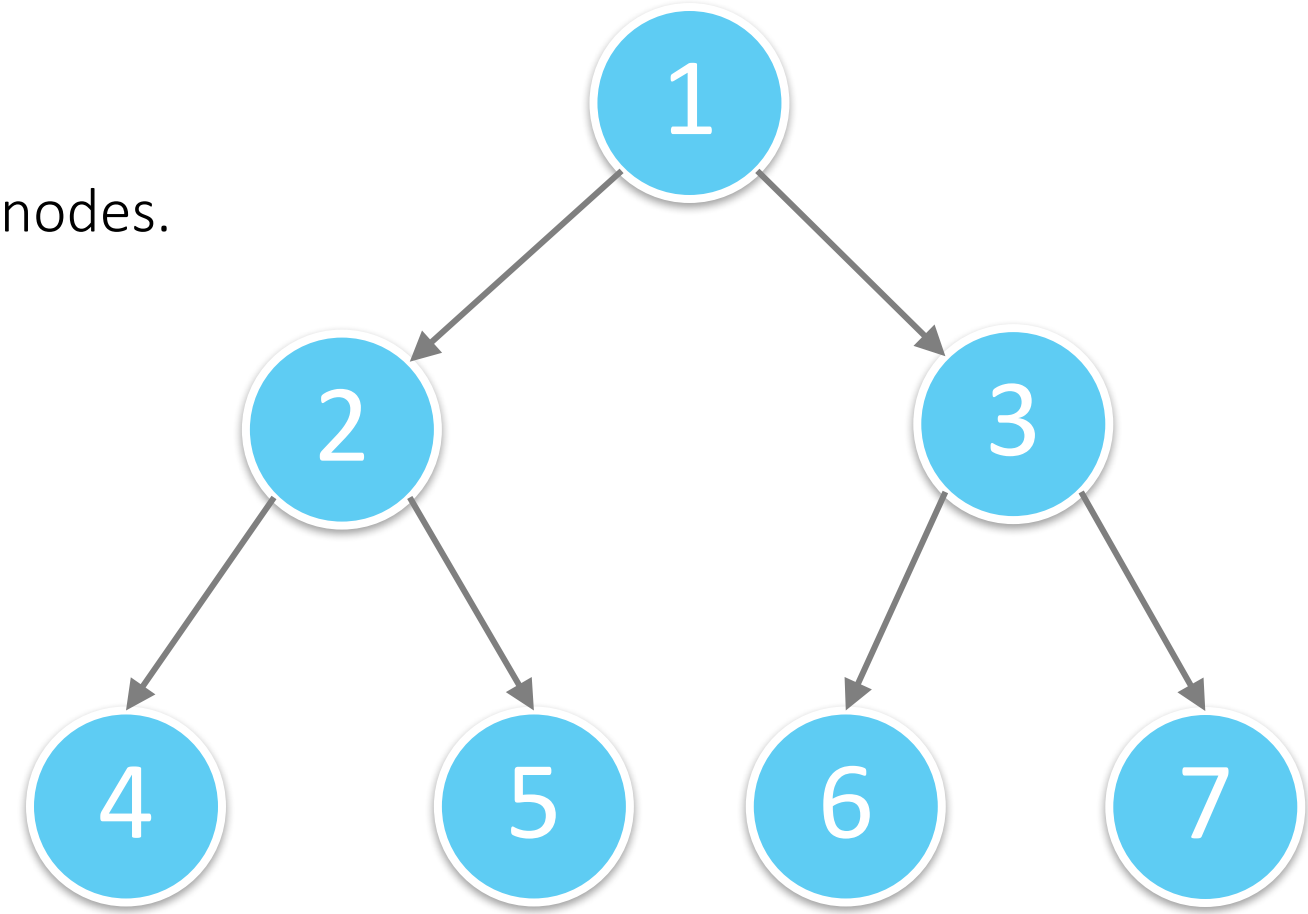
Nodes:

All the blue circles



Edges:

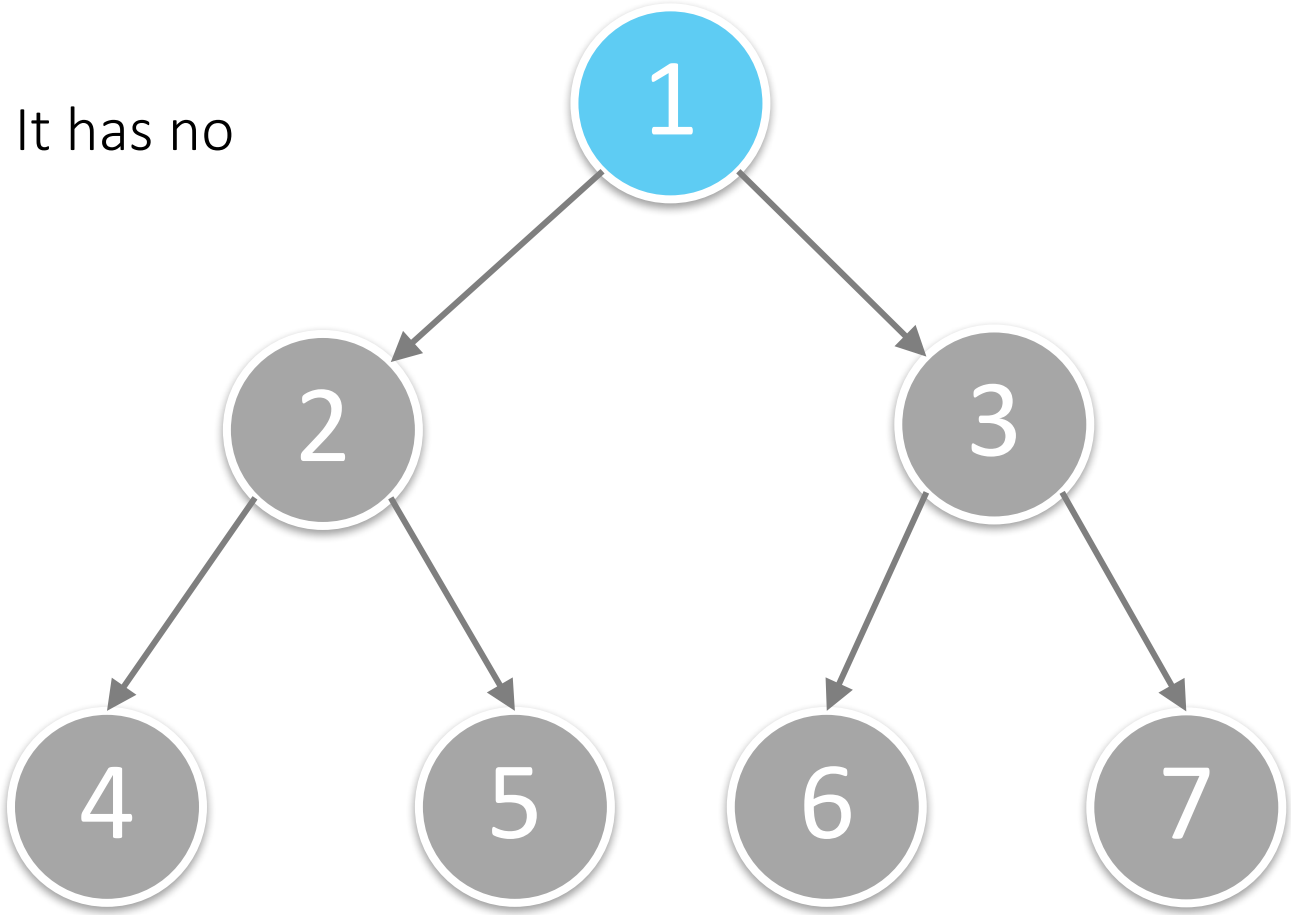
All the grey lines, representing the relationship between nodes.



Root:

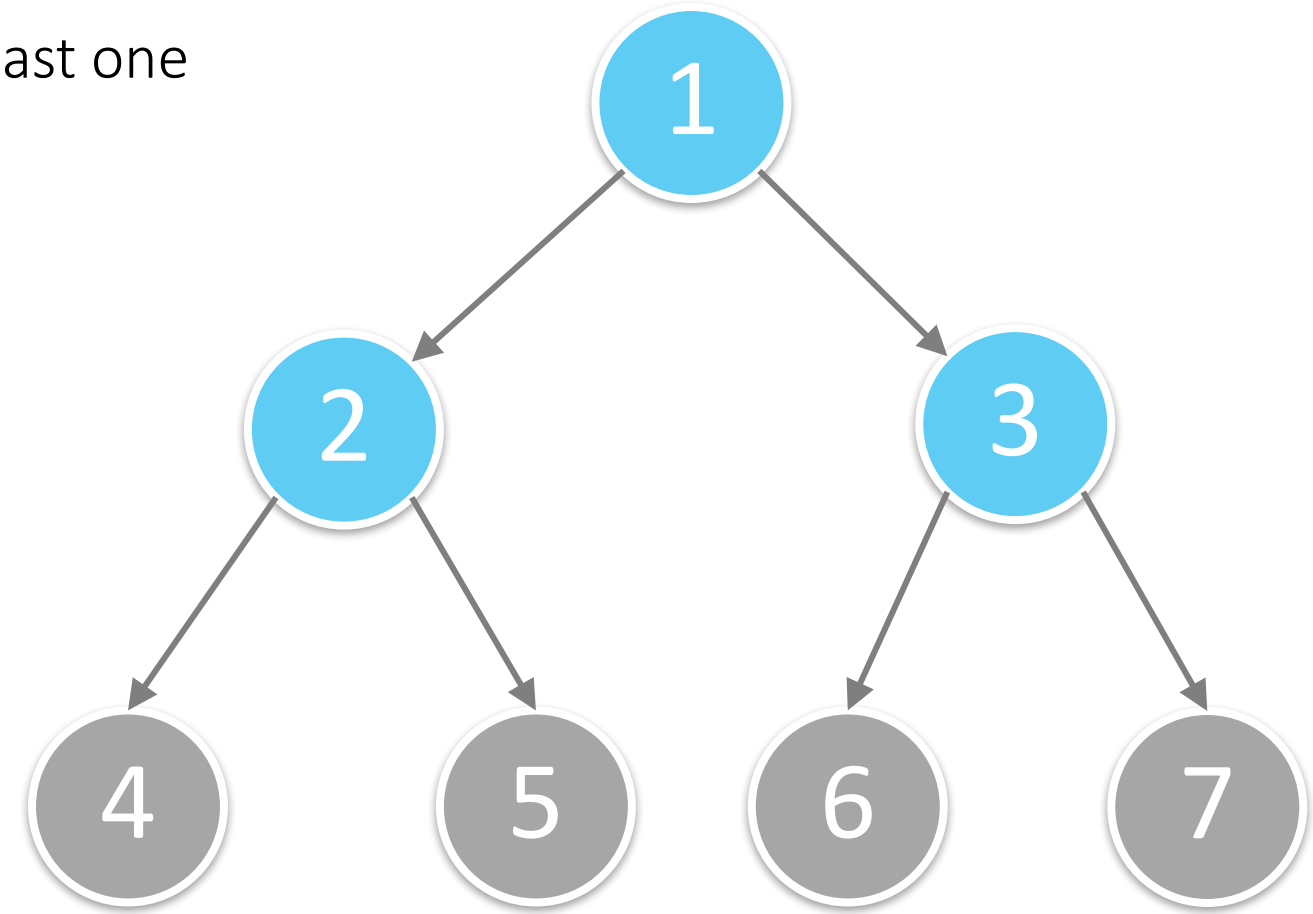
The blue circle.

The start of the tree. It has no parents.



Parent or Ancestor

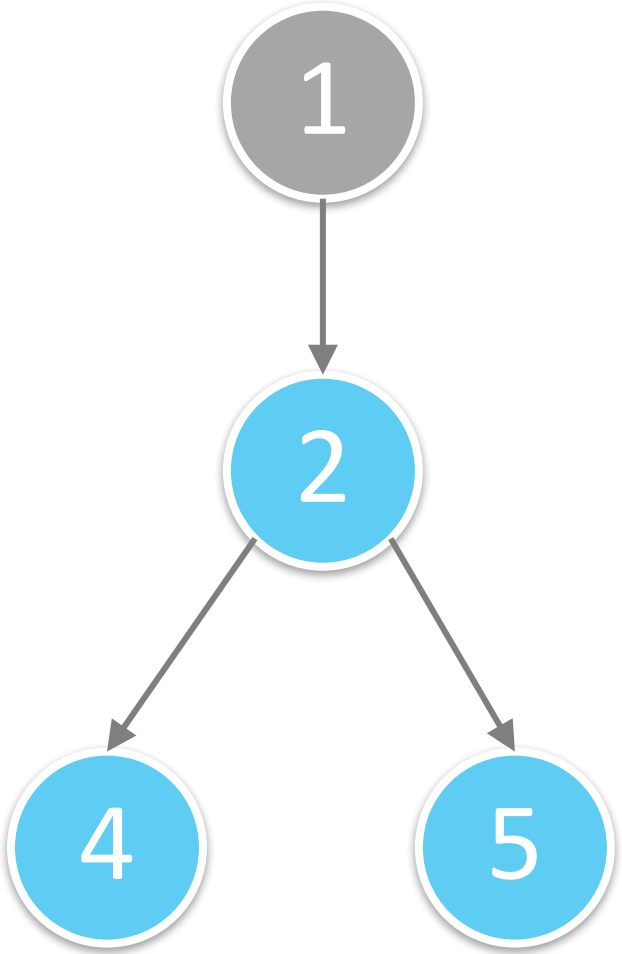
Any node with at least one child node.
(The blue nodes)



Children or Descendant

The blue nodes.

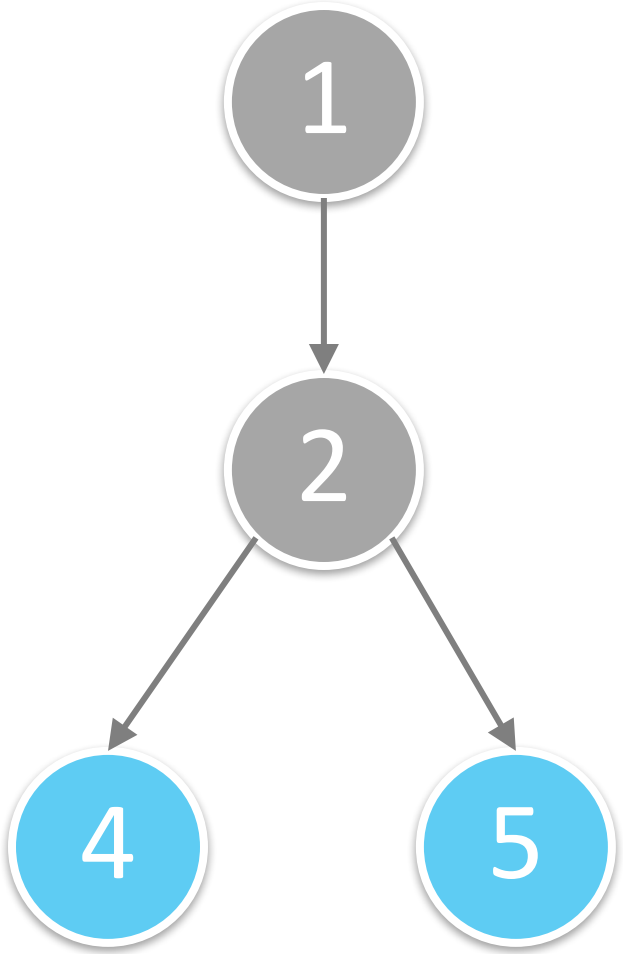
Any node with a parent.



Siblings

The blue nodes.

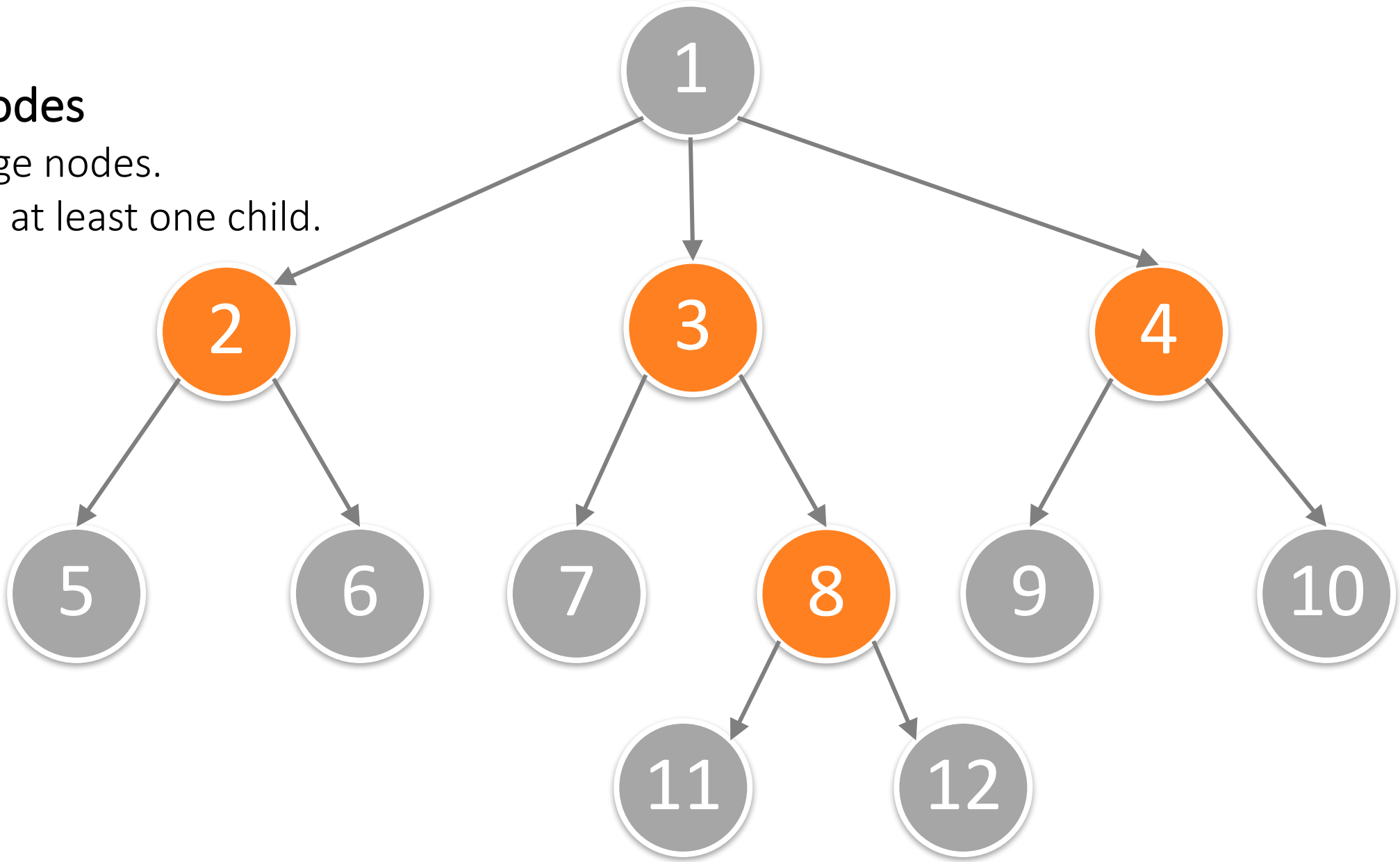
All nodes on the same level.



Internal nodes

All the orange nodes.

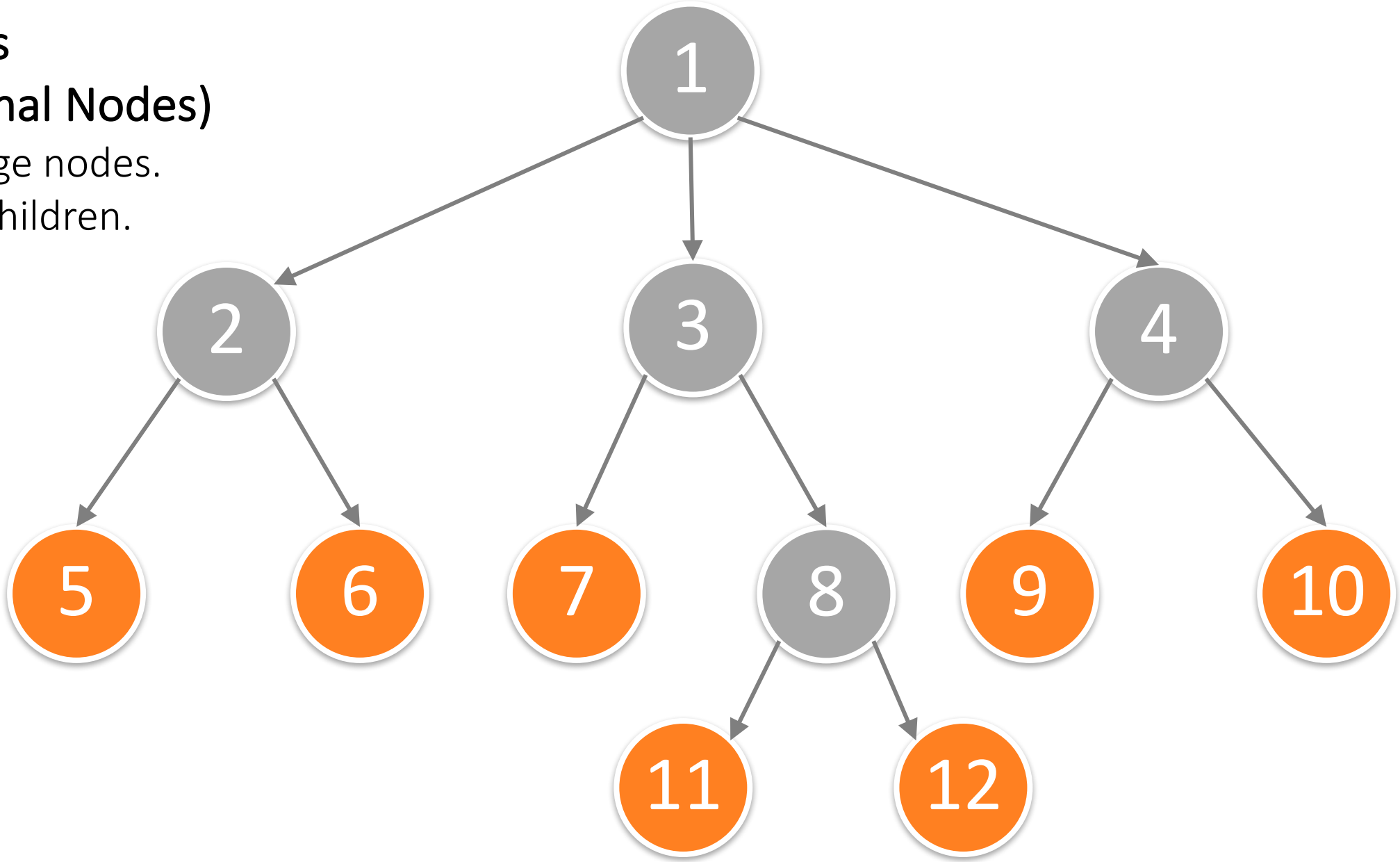
A node with at least one child.



Leaf nodes
(aka External Nodes)

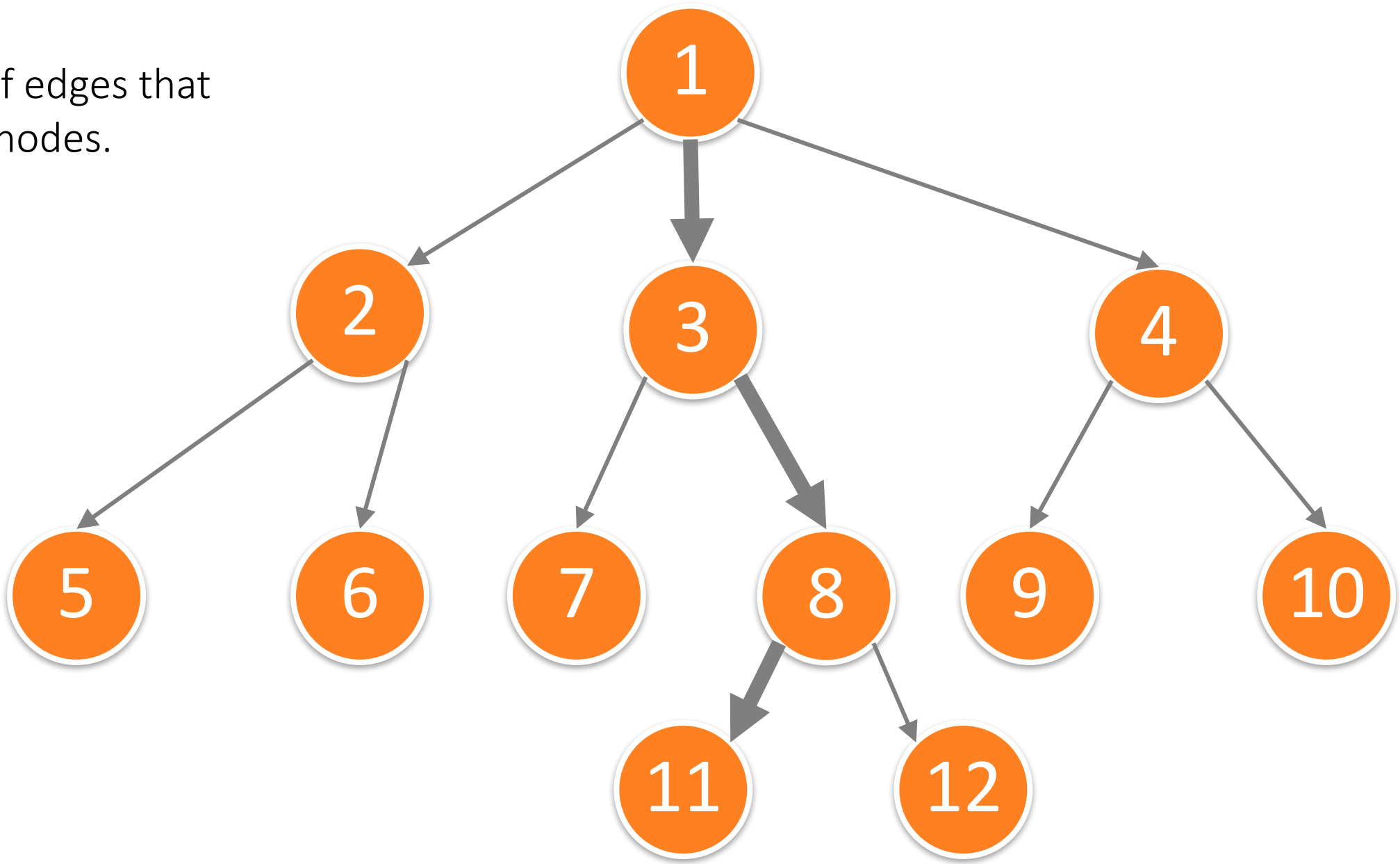
All the orange nodes.

A node no children.



Path

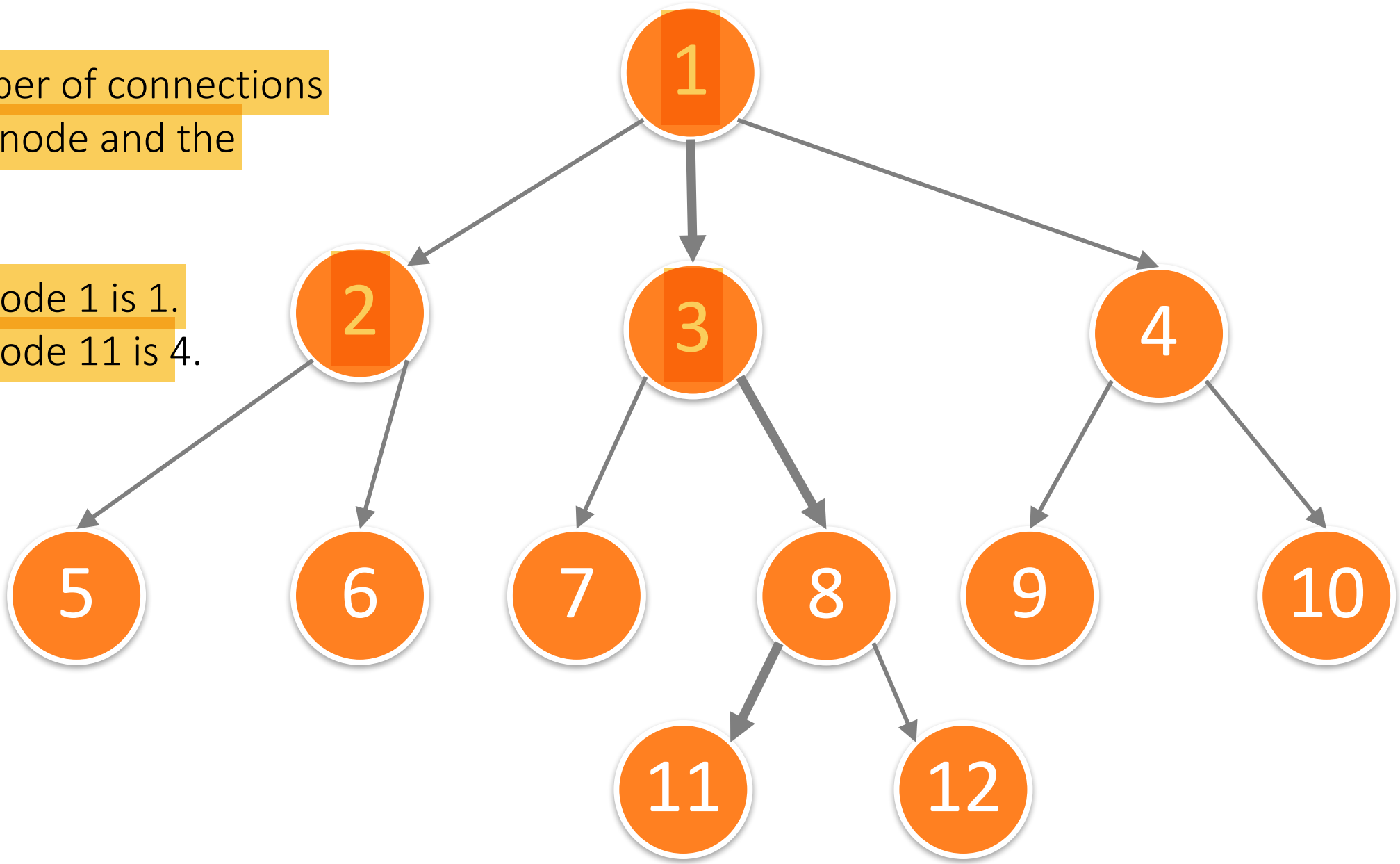
A sequence of edges that connect two nodes.



Level

1 + [the number of connections between the node and the root].

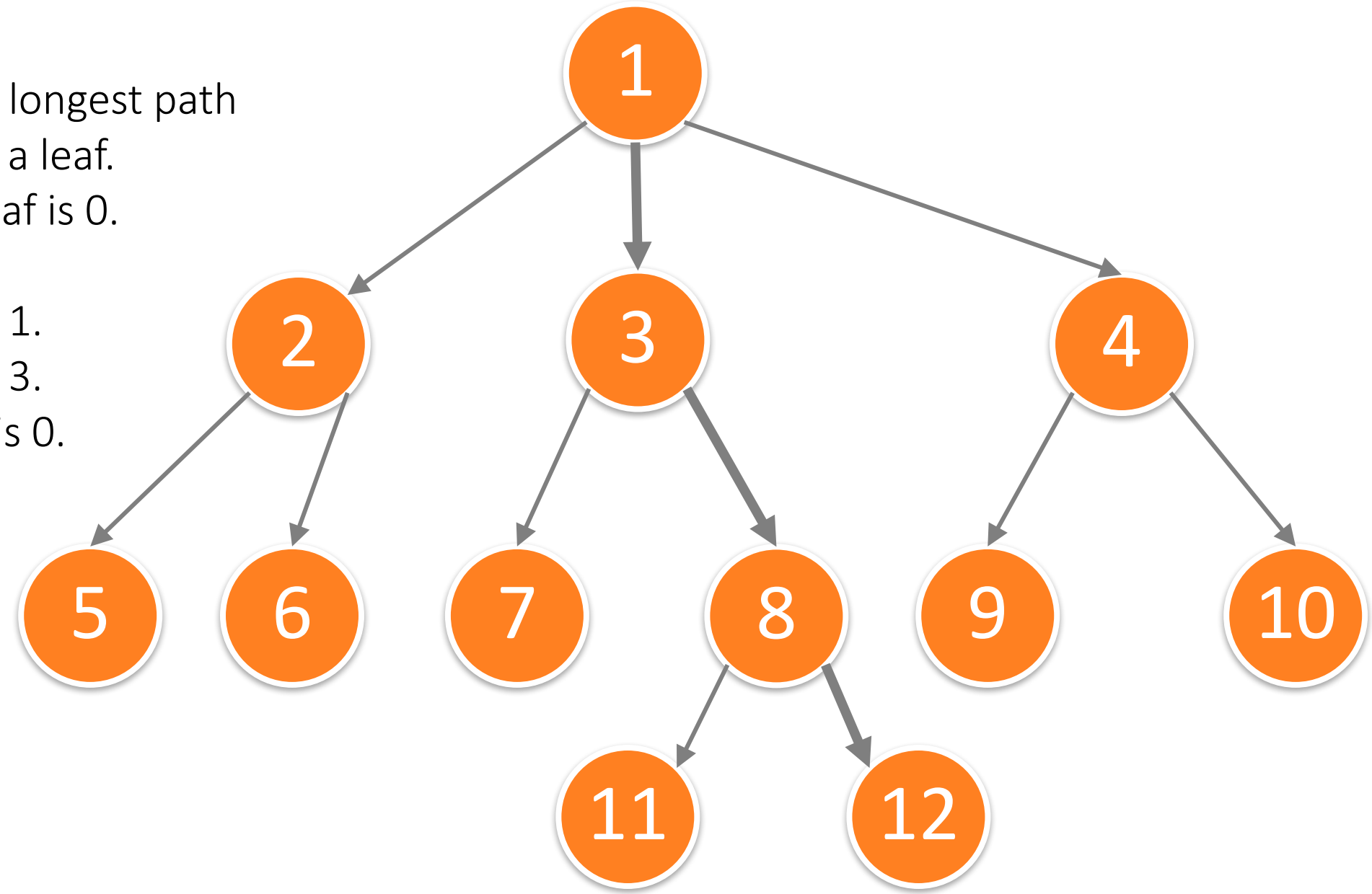
The level of node 1 is 1.
The level of node 11 is 4.



Height (node)

The length of the longest path from the node to a leaf.
The height of a leaf is 0.

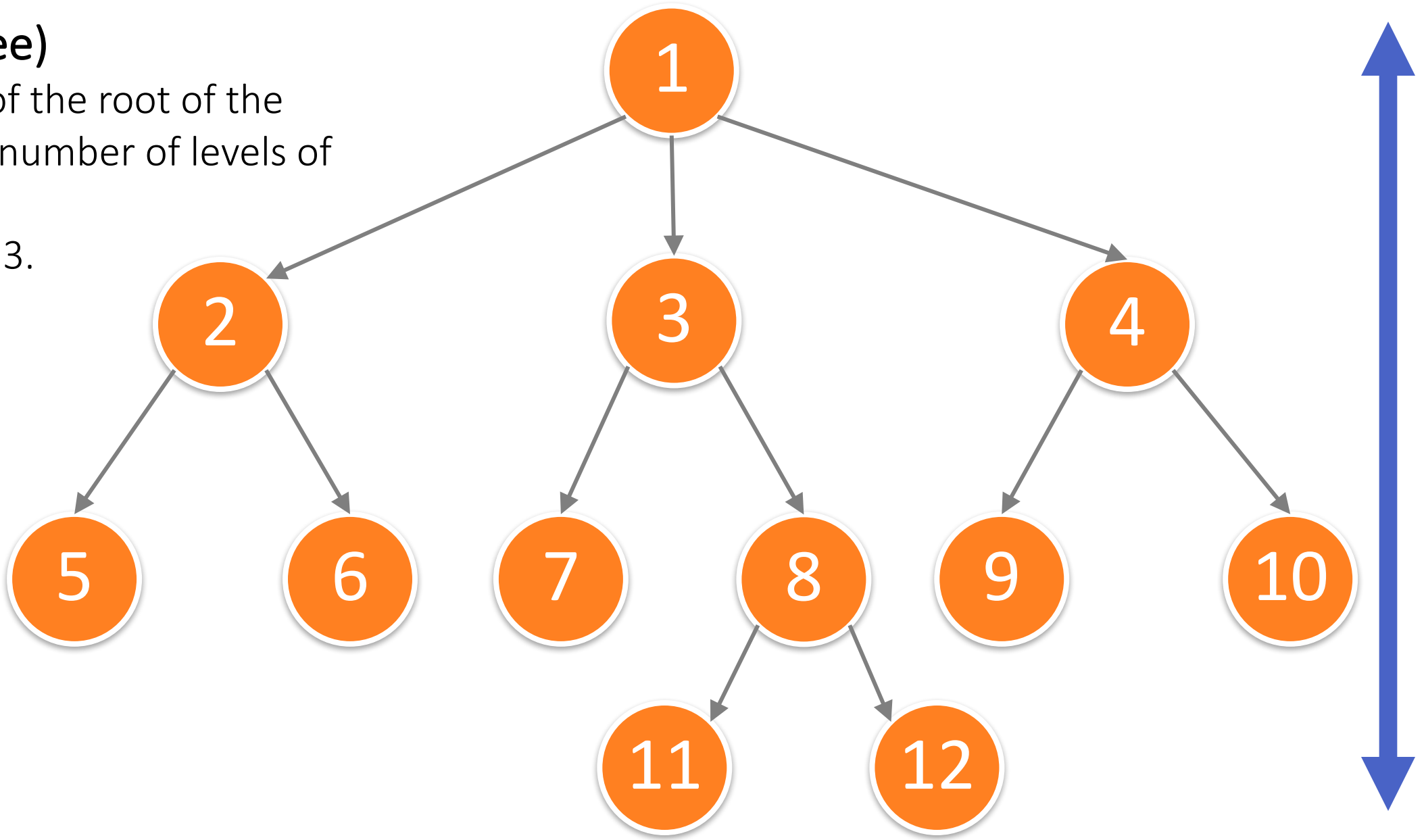
The height of 8 is 1.
The height of 1 is 3.
The height of 11 is 0.



Height (tree)

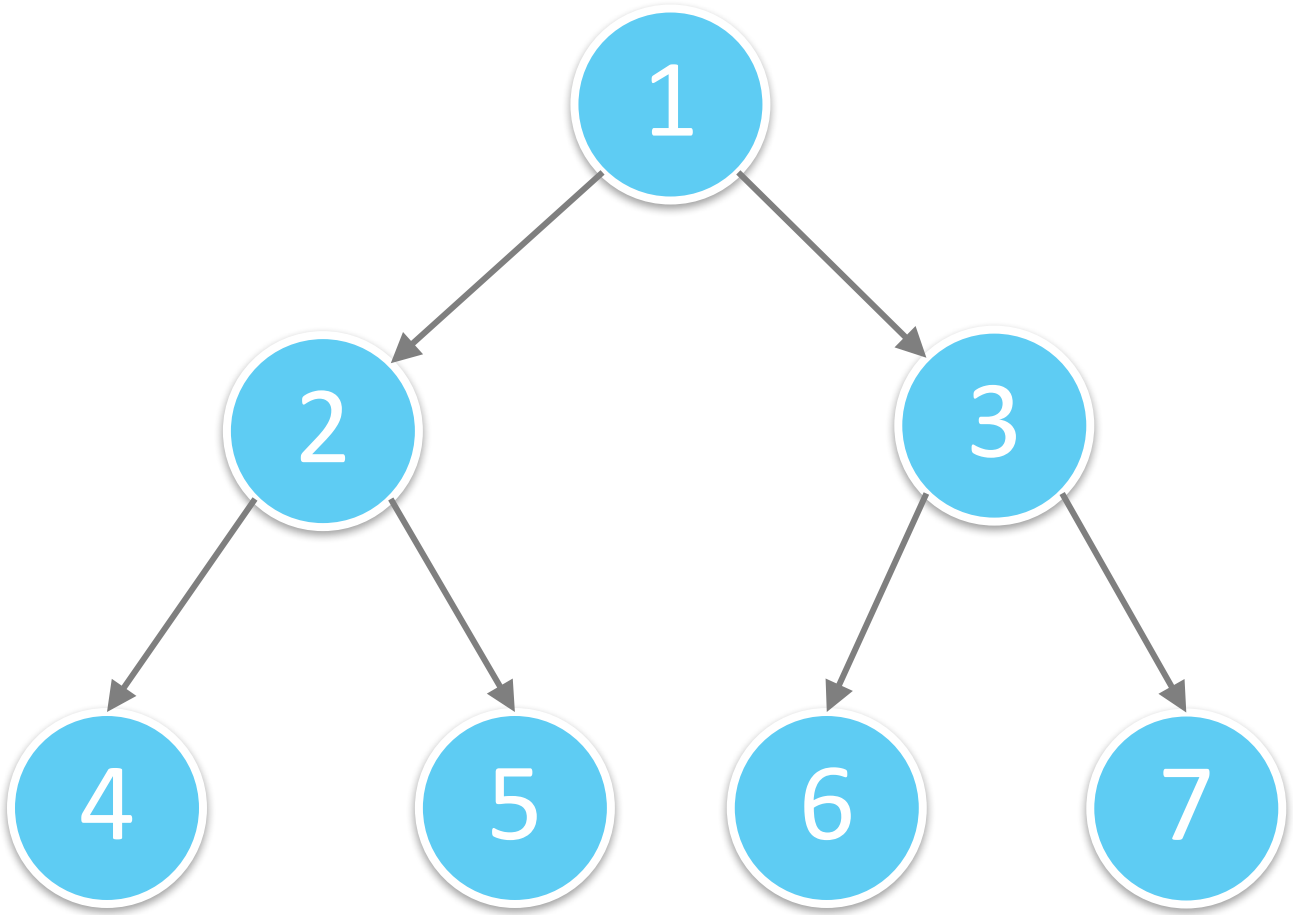
The height of the root of the tree, or the number of levels of a tree -1.

In this case, 3.



Binary Tree

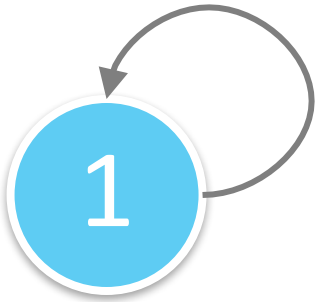
Every node has at most 2 children.



What is not a tree

Not a Tree

There is a cycle, and the root also has a parent (itself), so there is no root.

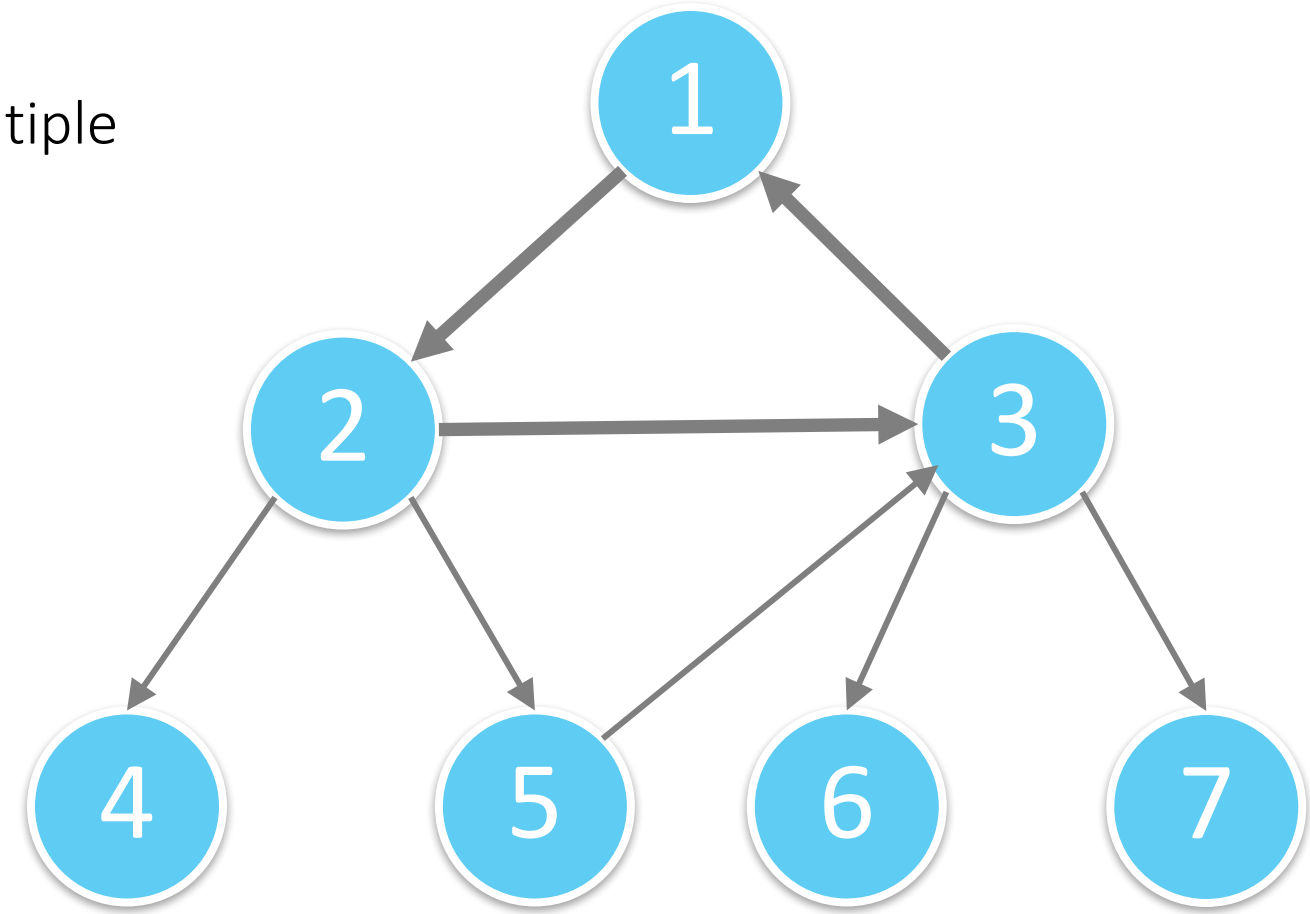


What is not a tree

Not a Tree

There is a cycle.

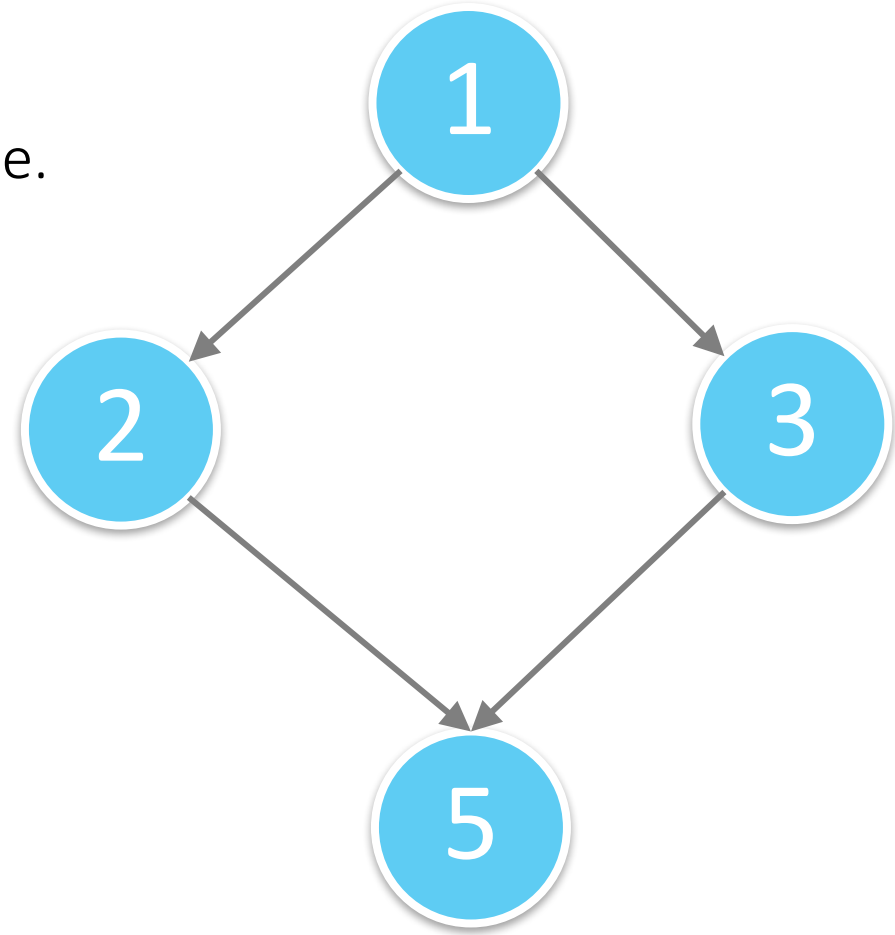
Also, node 3 has multiple parents.



What is not a tree

Not a Tree

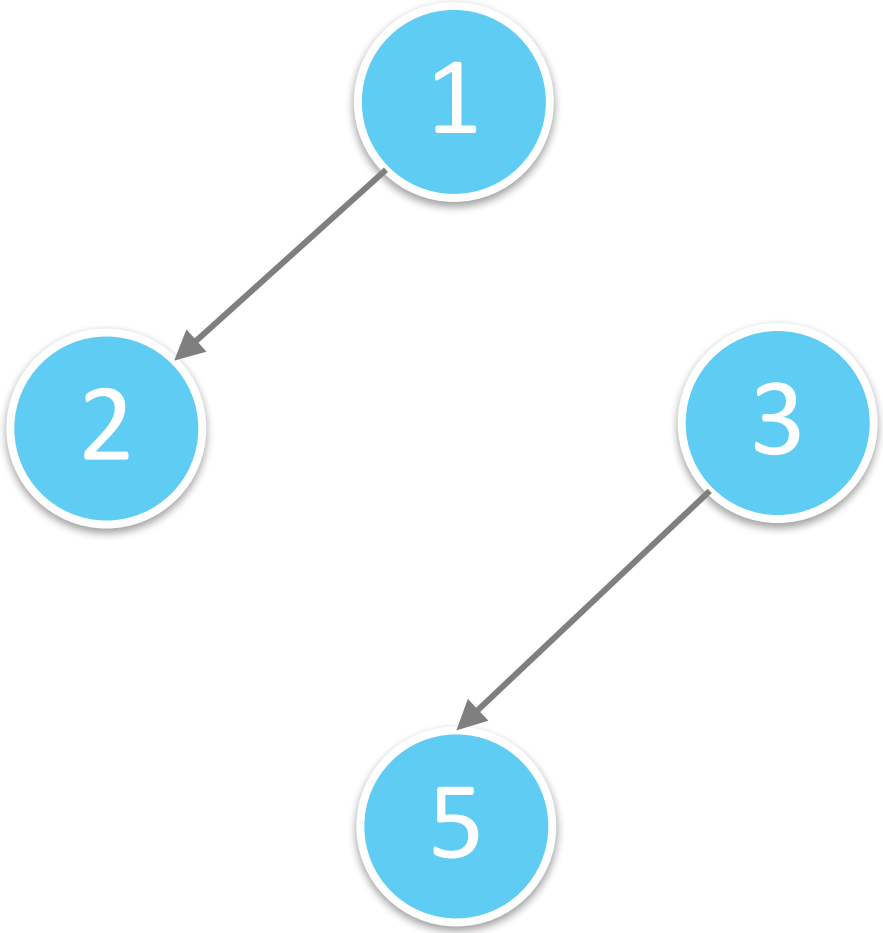
There is an undirected cycle.
Also, node 5 has multiple parents.



What is not a tree

Not a Tree

There are 2 unconnected groups.



What is a tree?

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

Special Trees

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

Special Trees

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

Binary Tree
A tree where each node has at most 2 children.

Special Trees

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

Binary Search Tree Nodes are in a special order to make it easier to search.

Special Trees

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

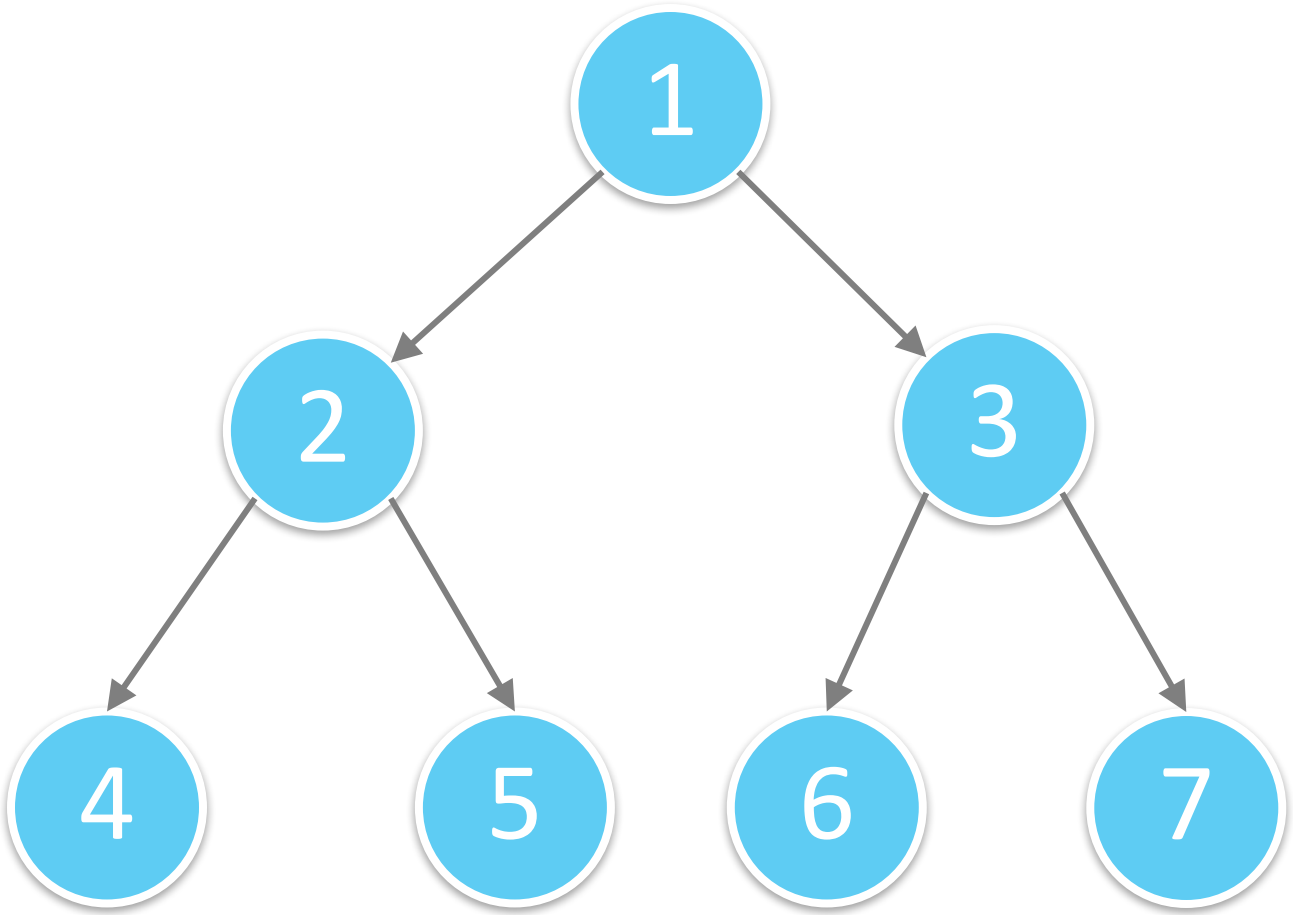
Heap/Priority Queue Nodes are in a special order to make it easier to search.

Special Trees

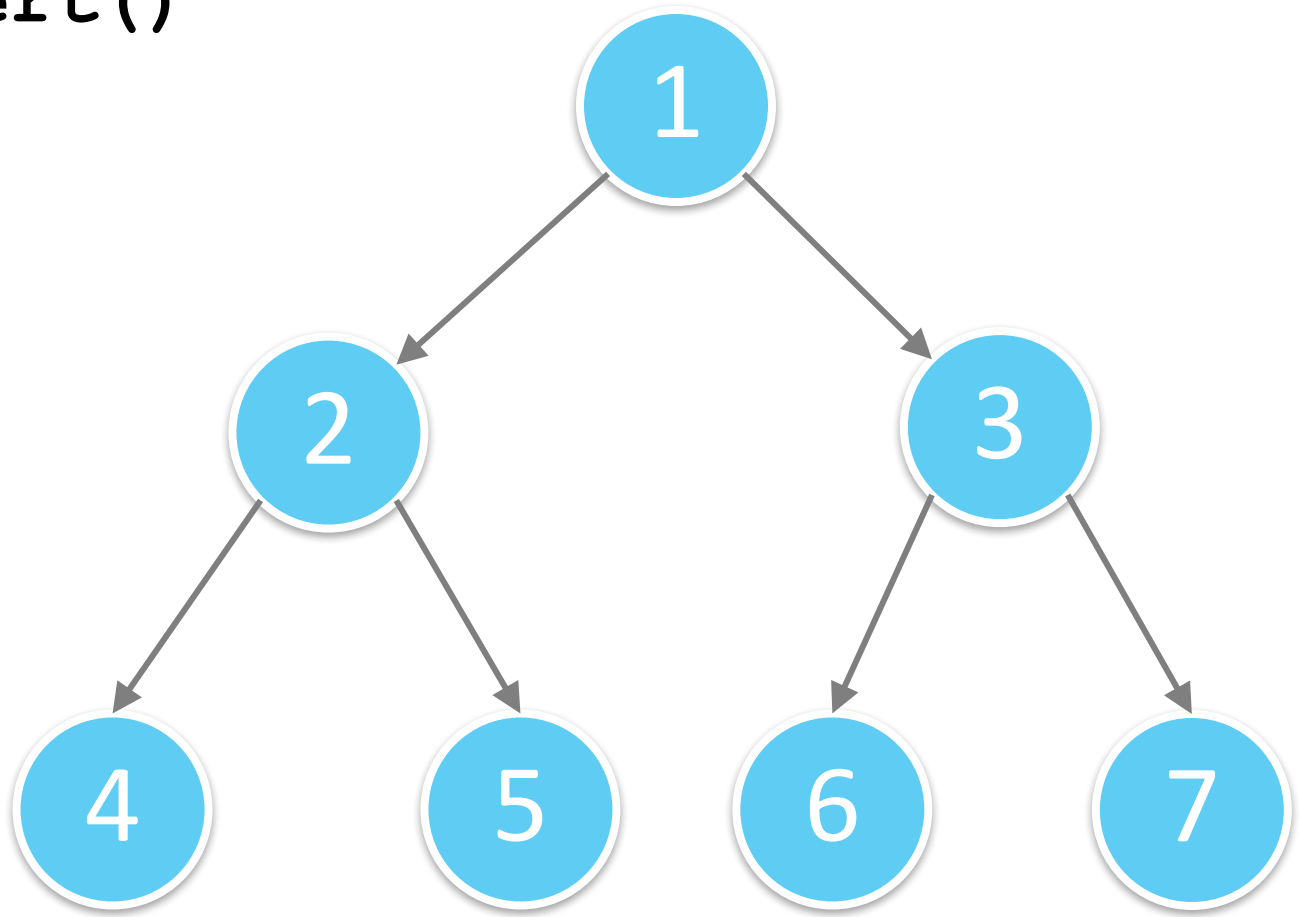
- Binary Tree
- Binary Search Tree
- Binary Heap/Priority Queue
- Red-Black Tree

Red-Black Tree
A special kind of balanced binary search tree.

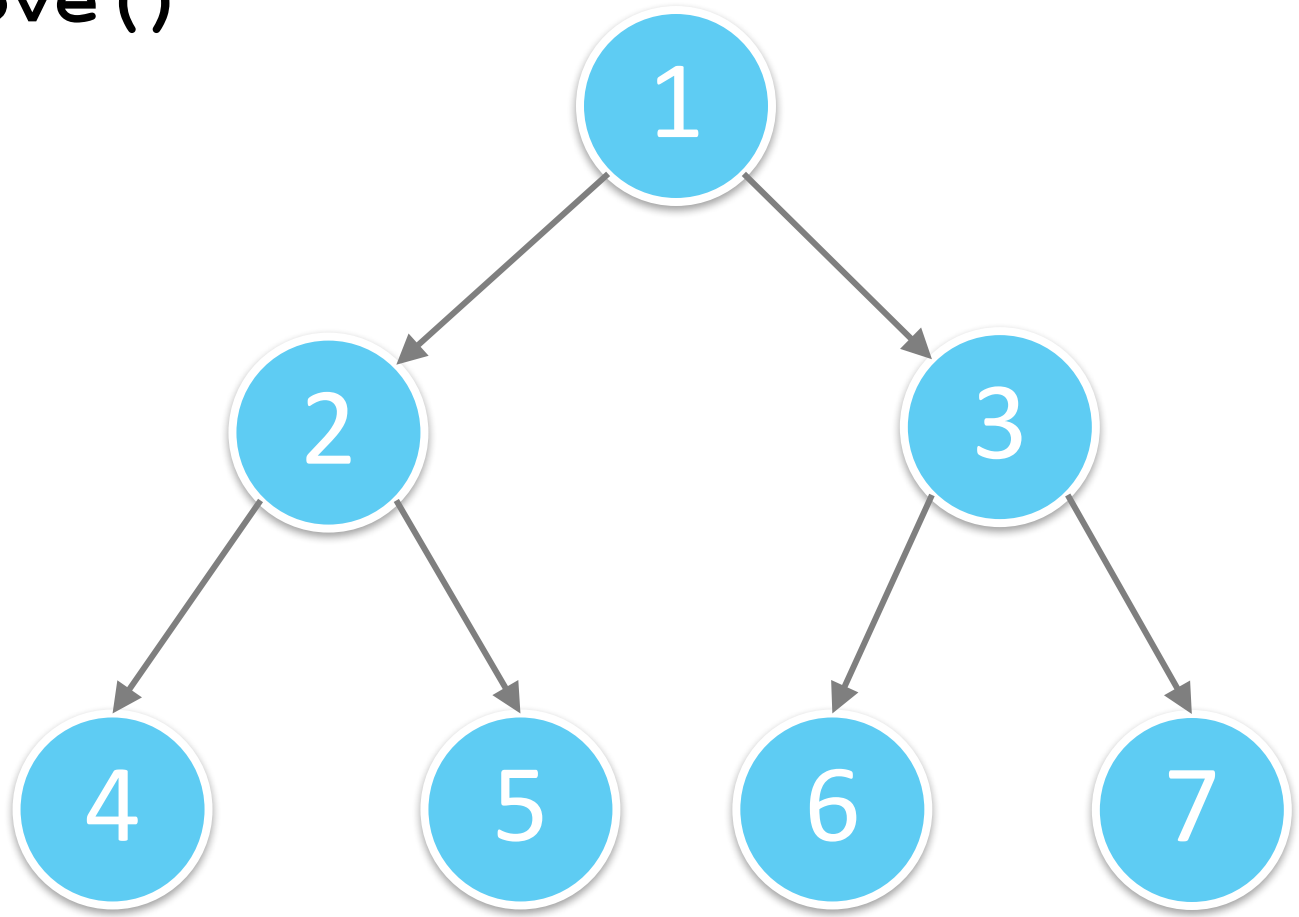
BINARY TREES



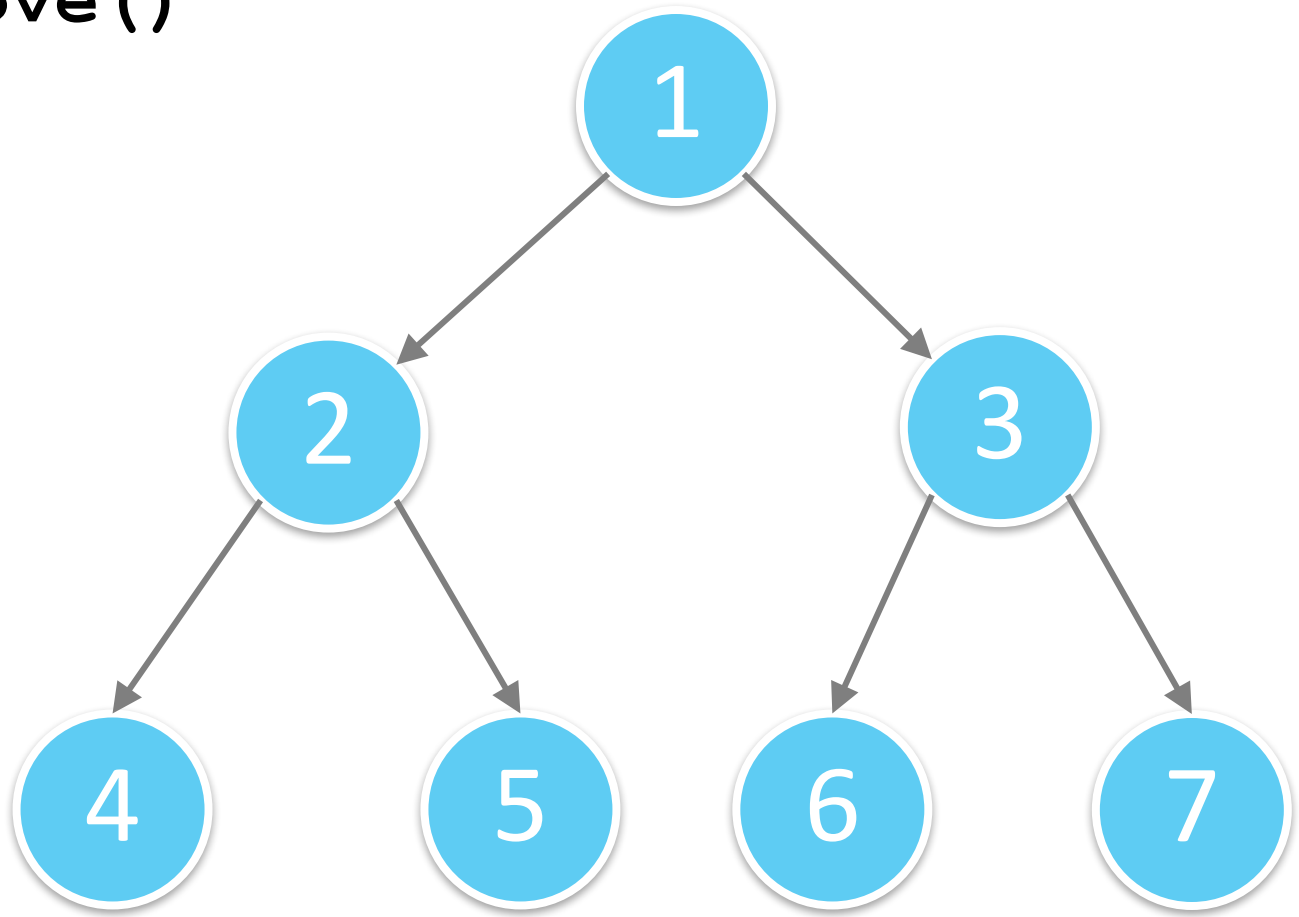
`insert()`

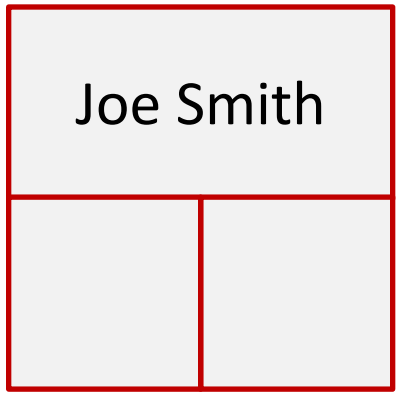


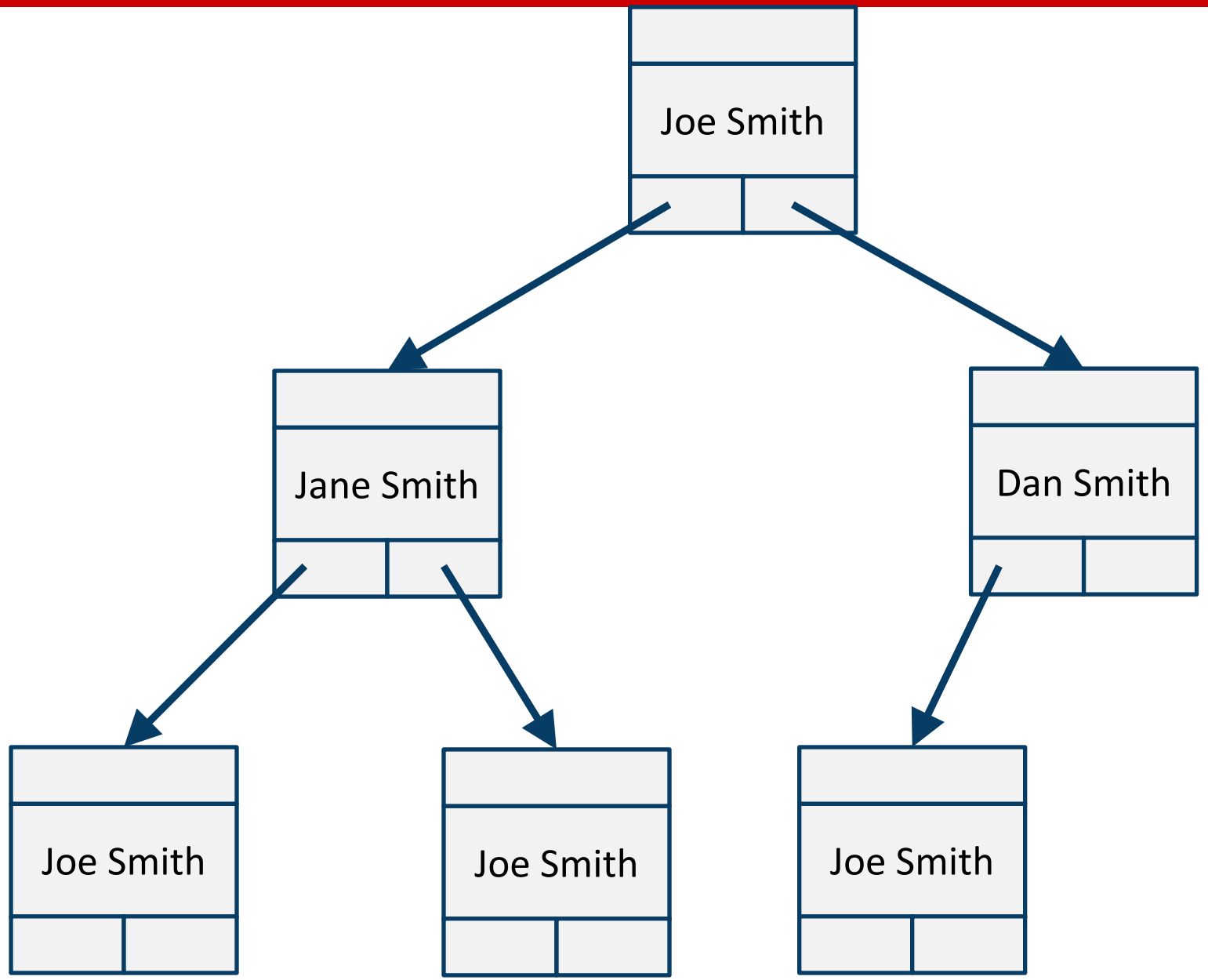
`remove()`

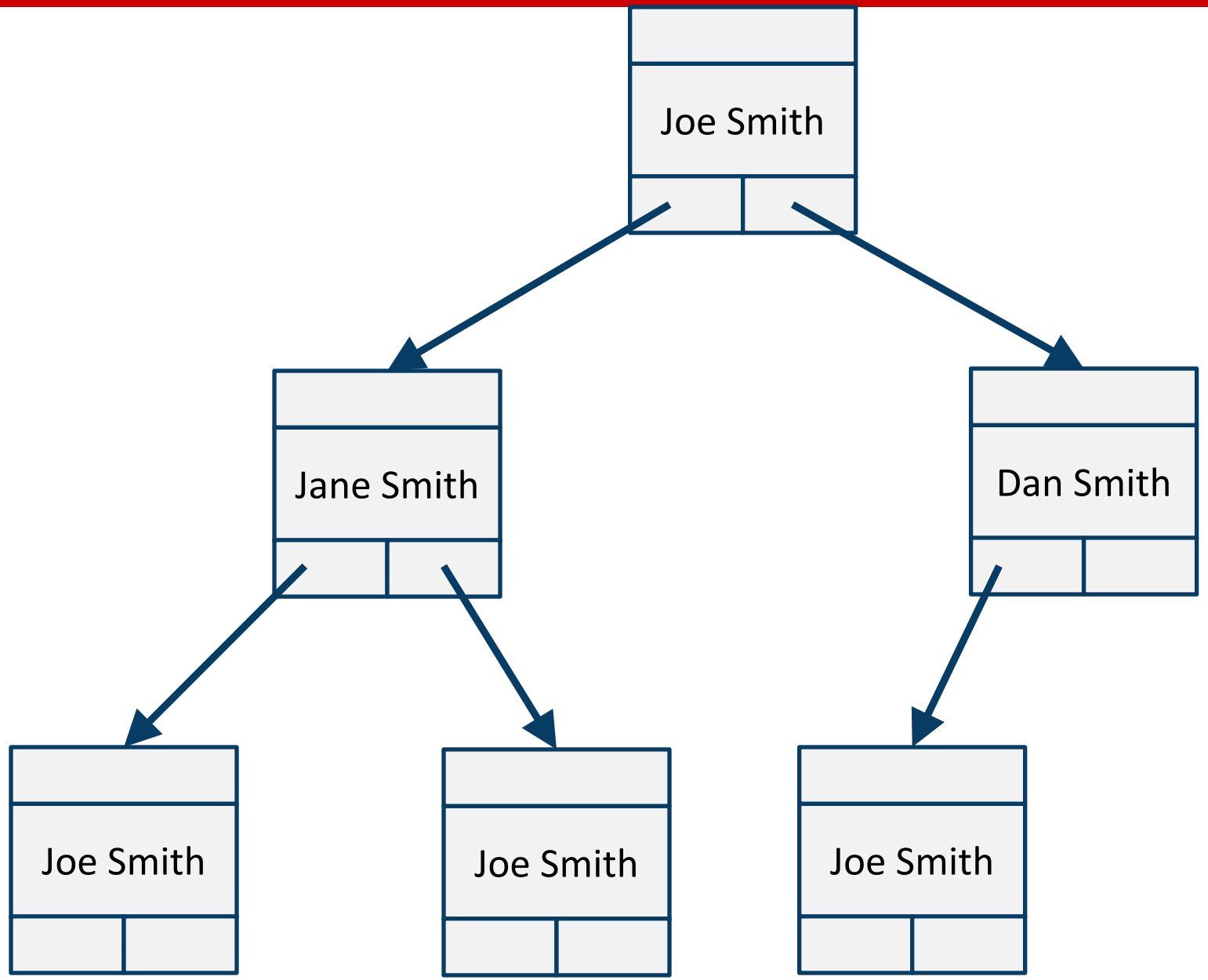


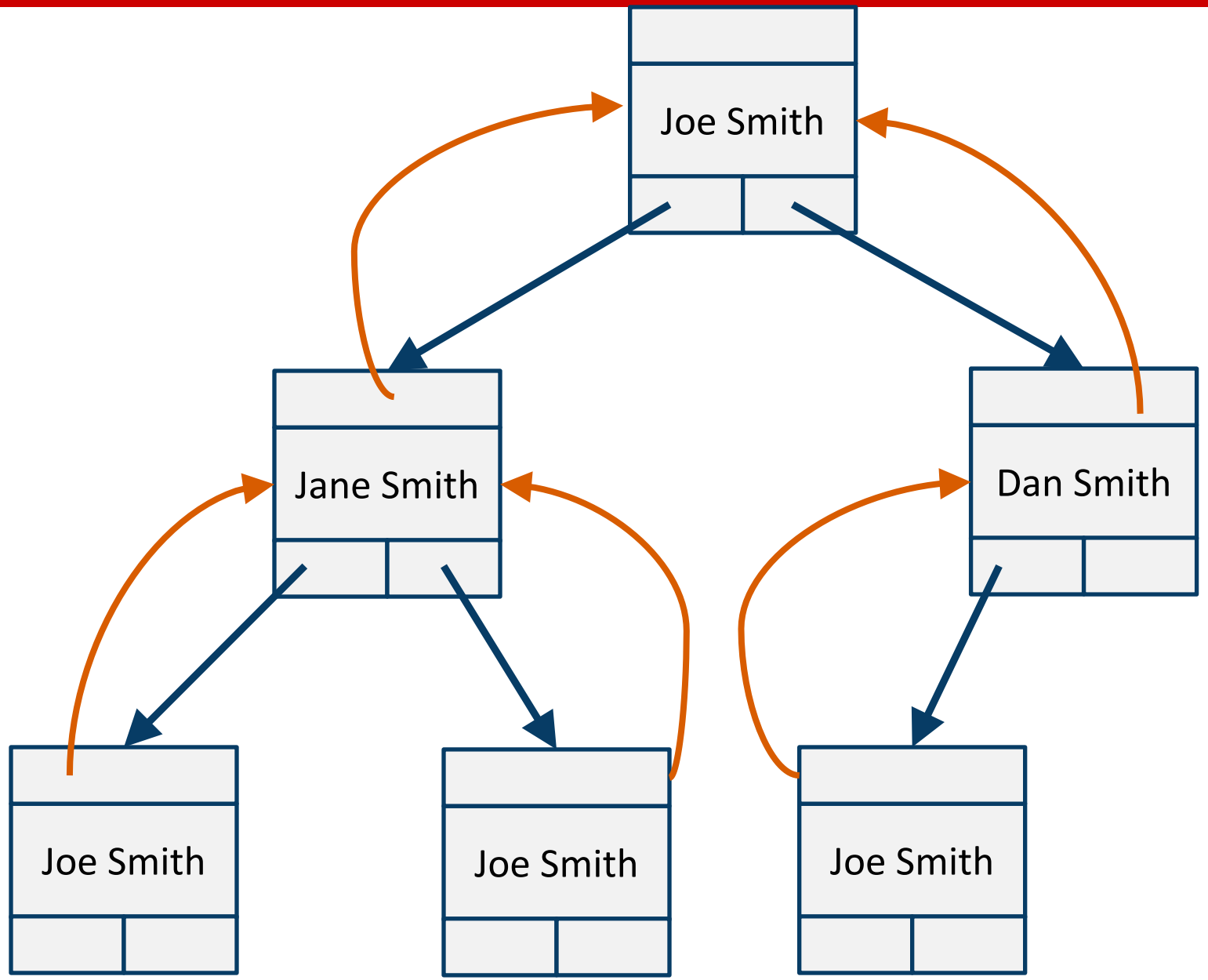
`remove()`

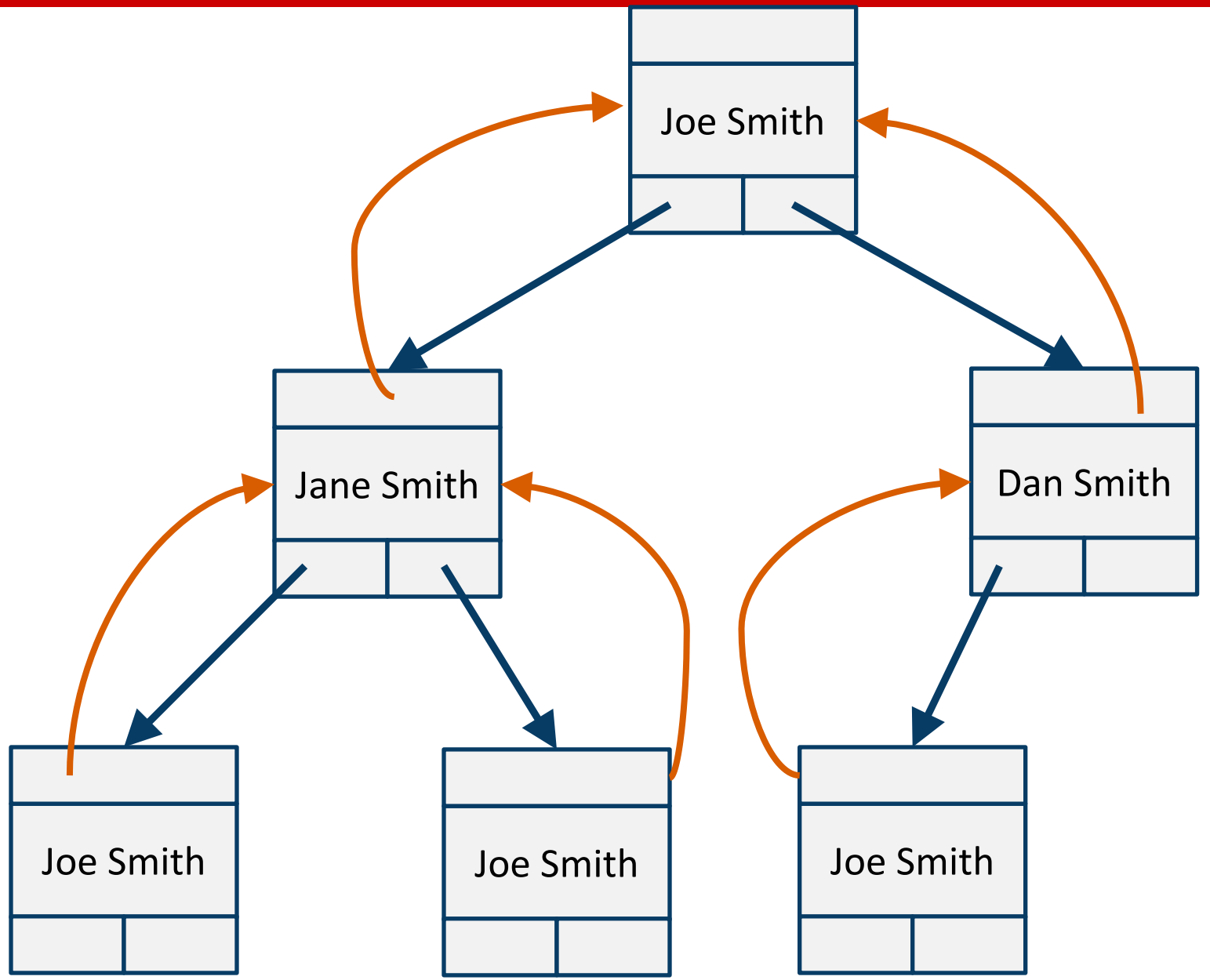




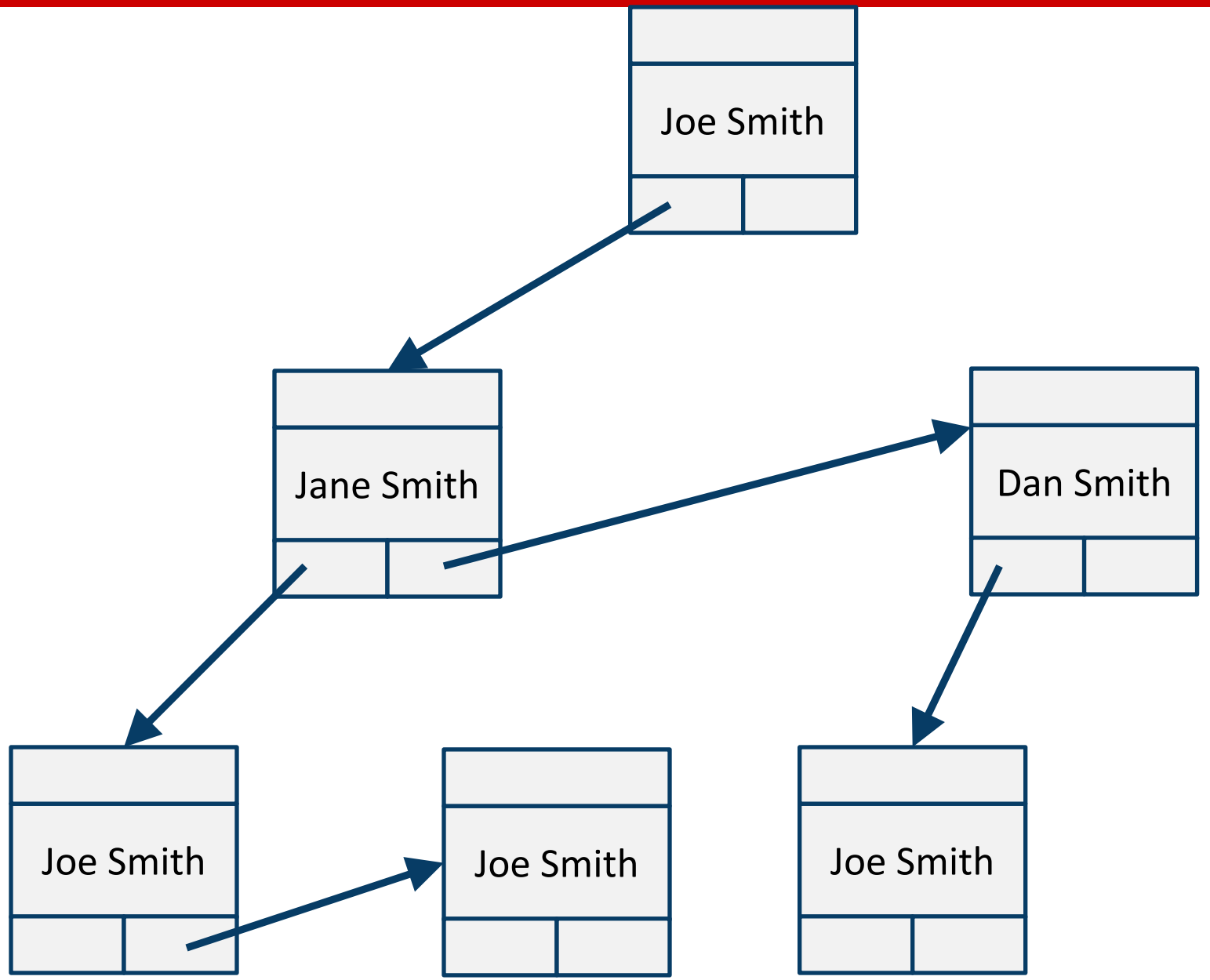






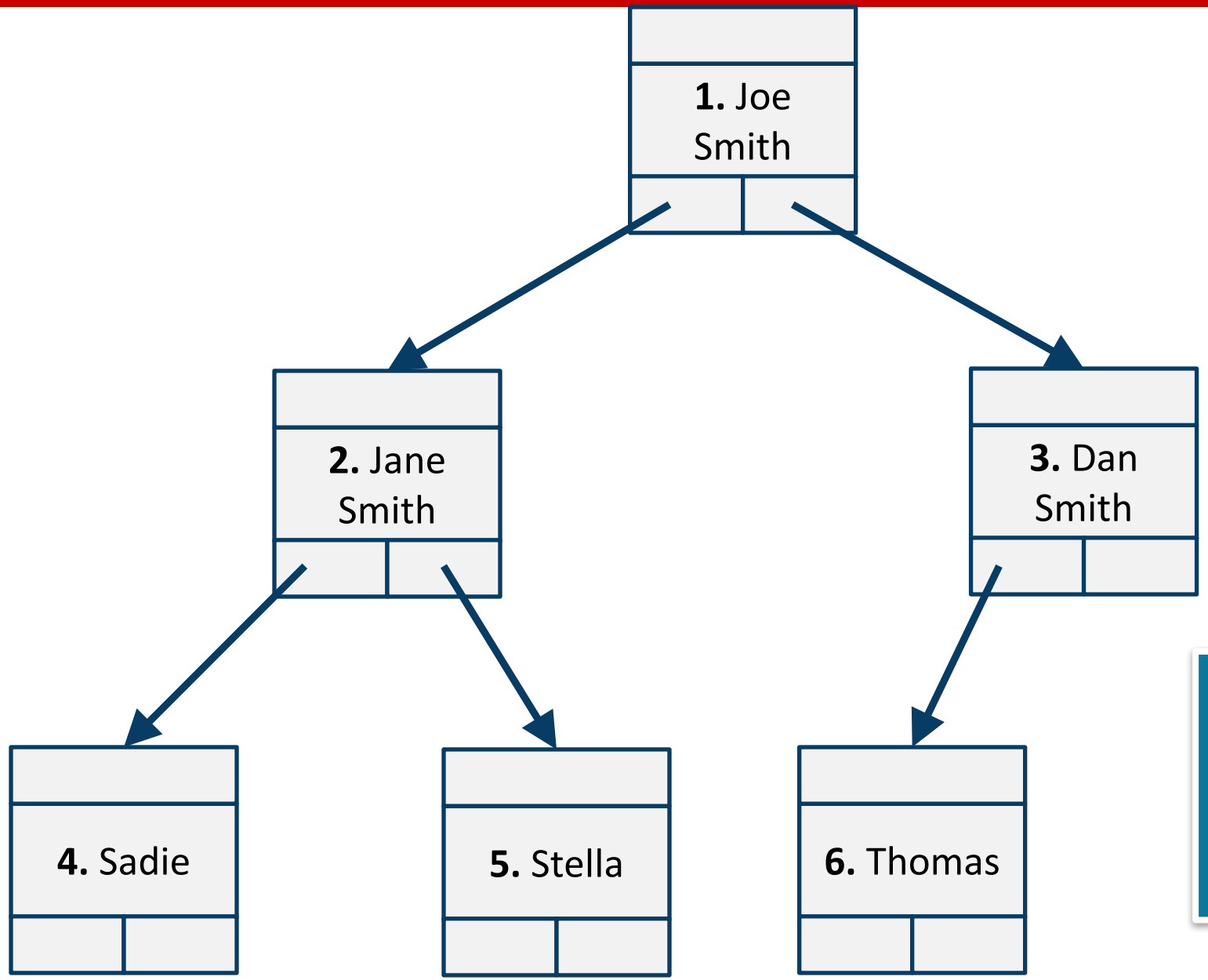


For simplicity, pointers to the parent will not be in these diagrams. Just note that your implementation can have this link.



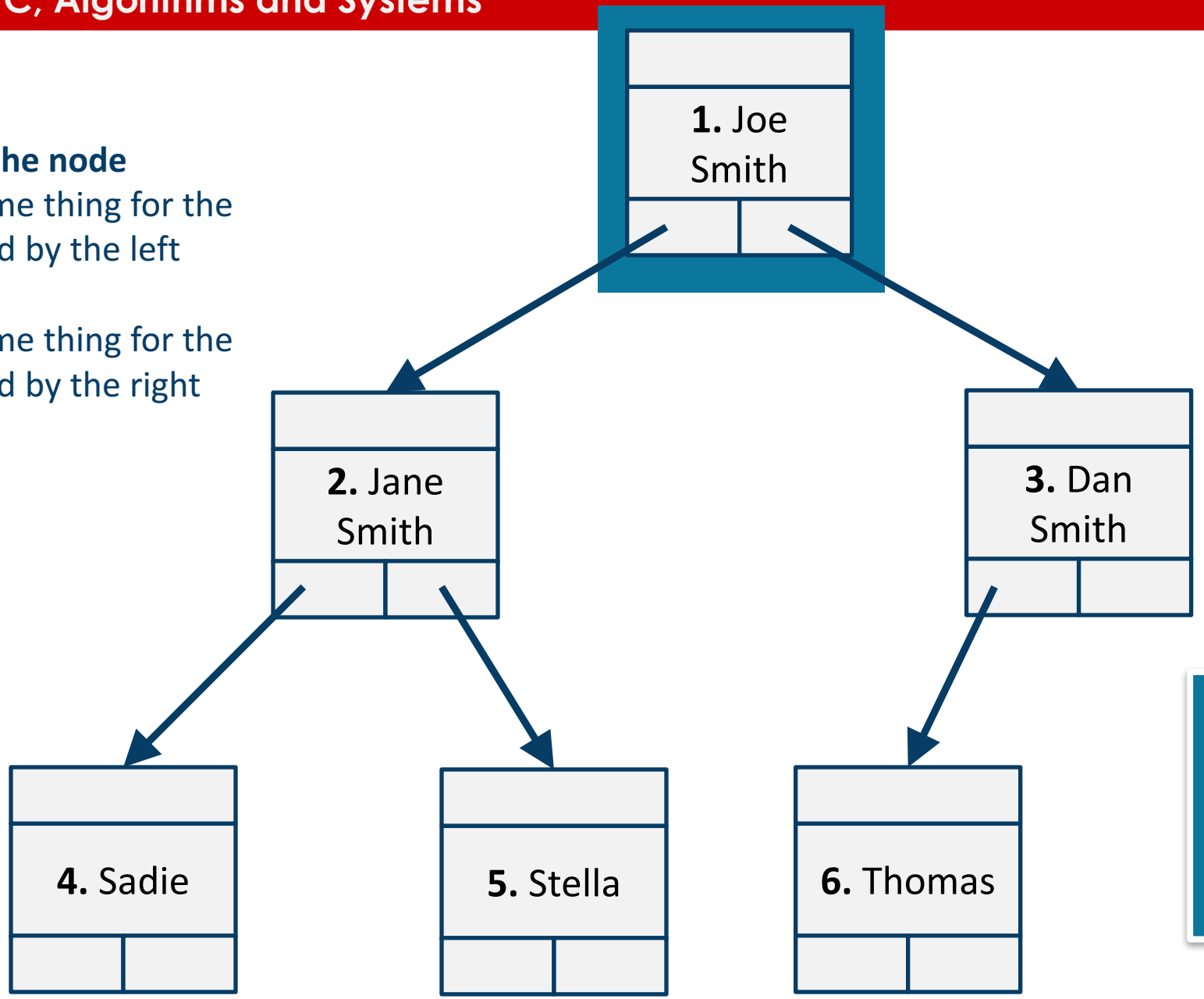
Left-child,
Right-sibling
representation

Traversing the Tree



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

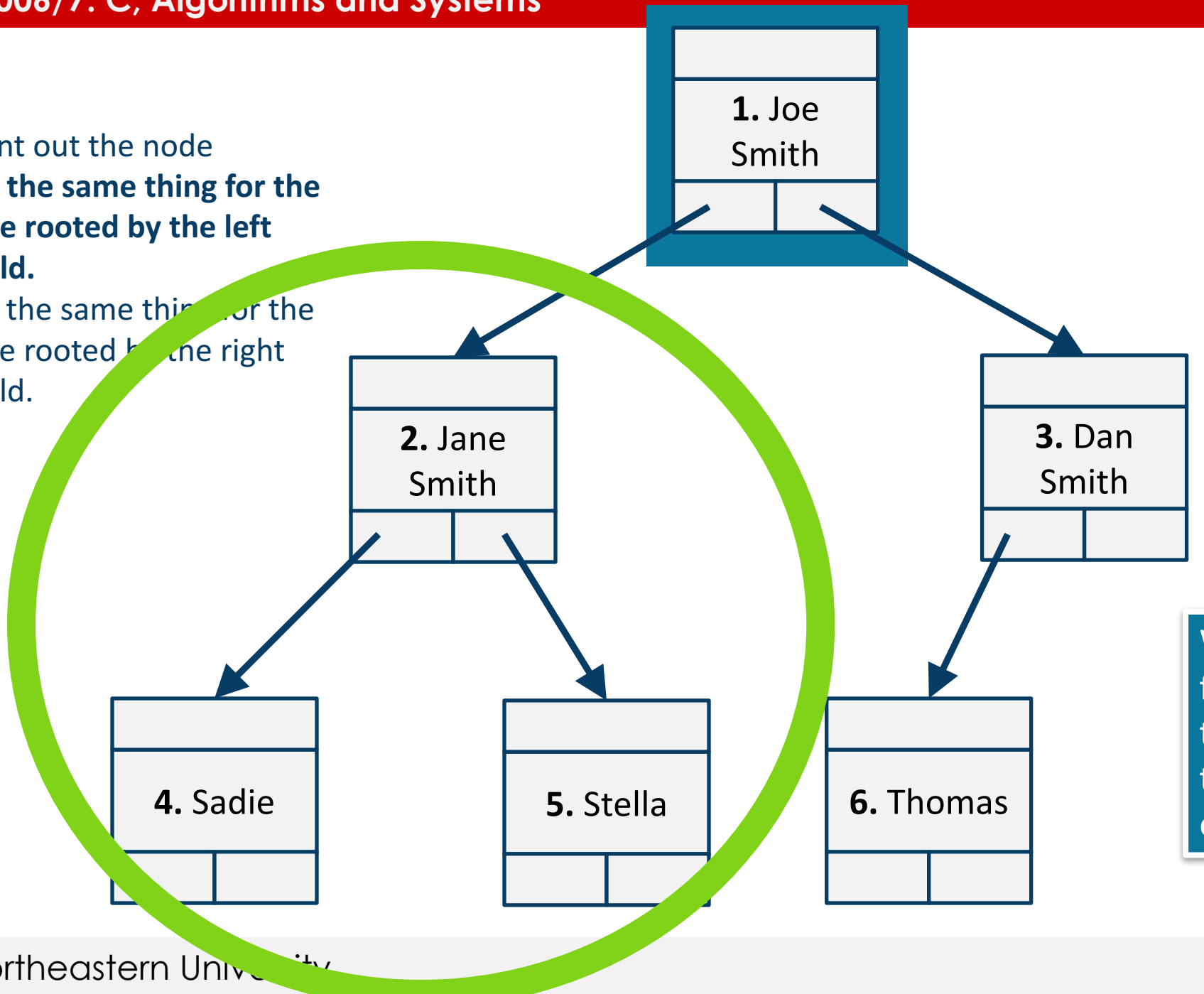
1. **Print out the node**
2. Do the same thing for the tree rooted by the left child.
3. Do the same thing for the tree rooted by the right child.




```
Joe Smith
```

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

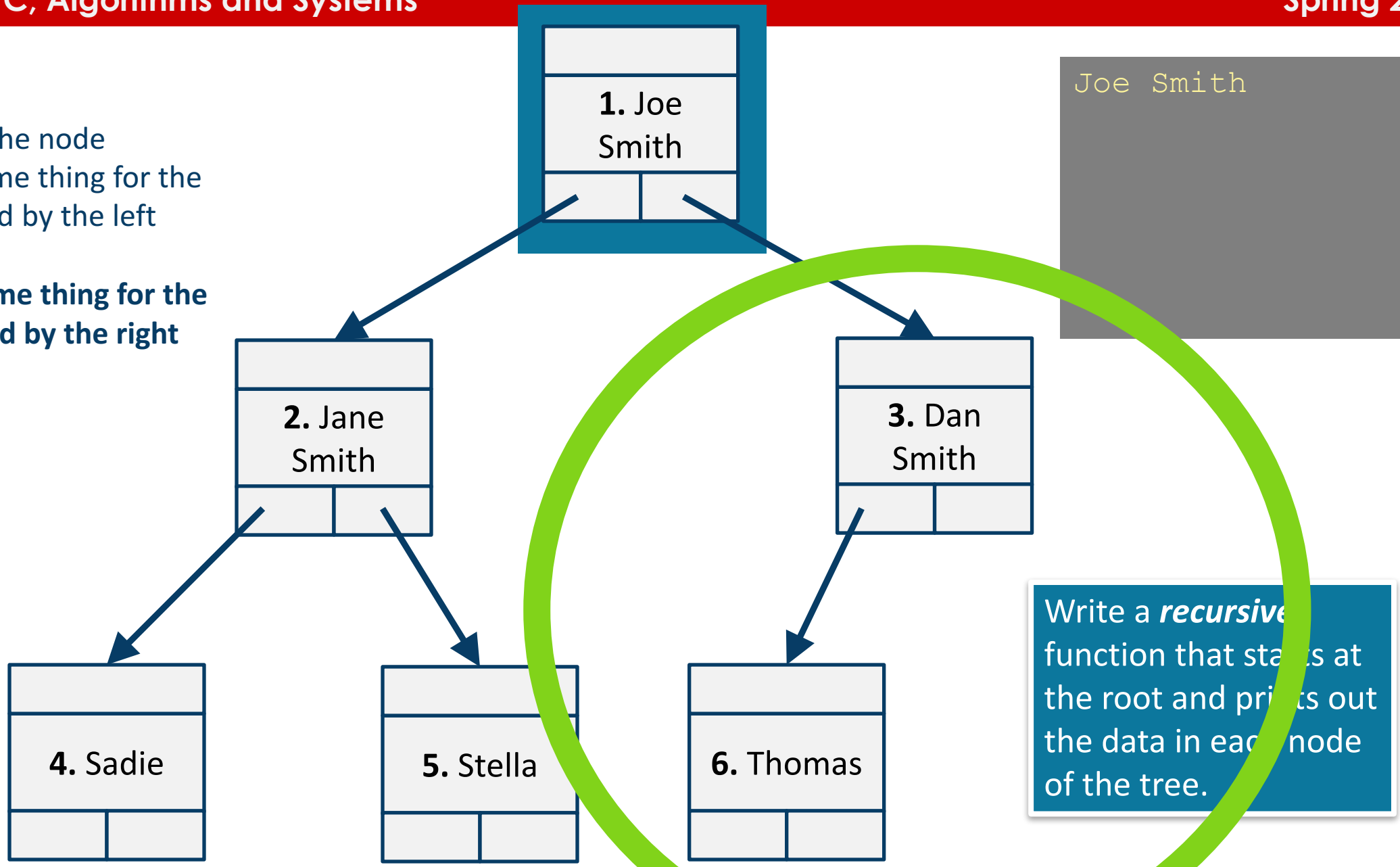
1. Print out the node
2. **Do the same thing for the tree rooted by the left child.**
3. Do the same thing for the tree rooted by the right child.




```
Joe Smith
```

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

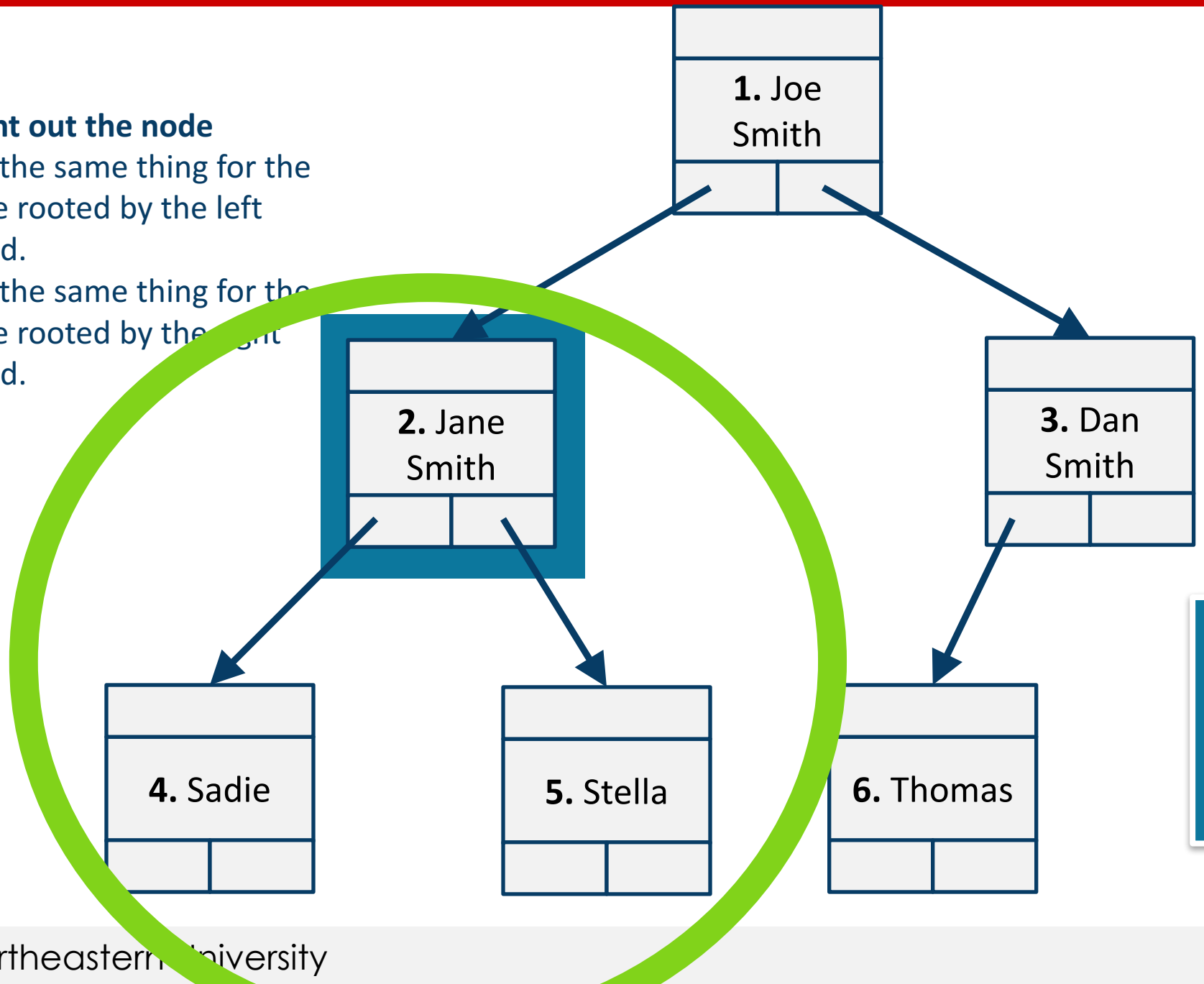
1. Print out the node
2. Do the same thing for the tree rooted by the left child.
3. **Do the same thing for the tree rooted by the right child.**




```
Joe Smith
```

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

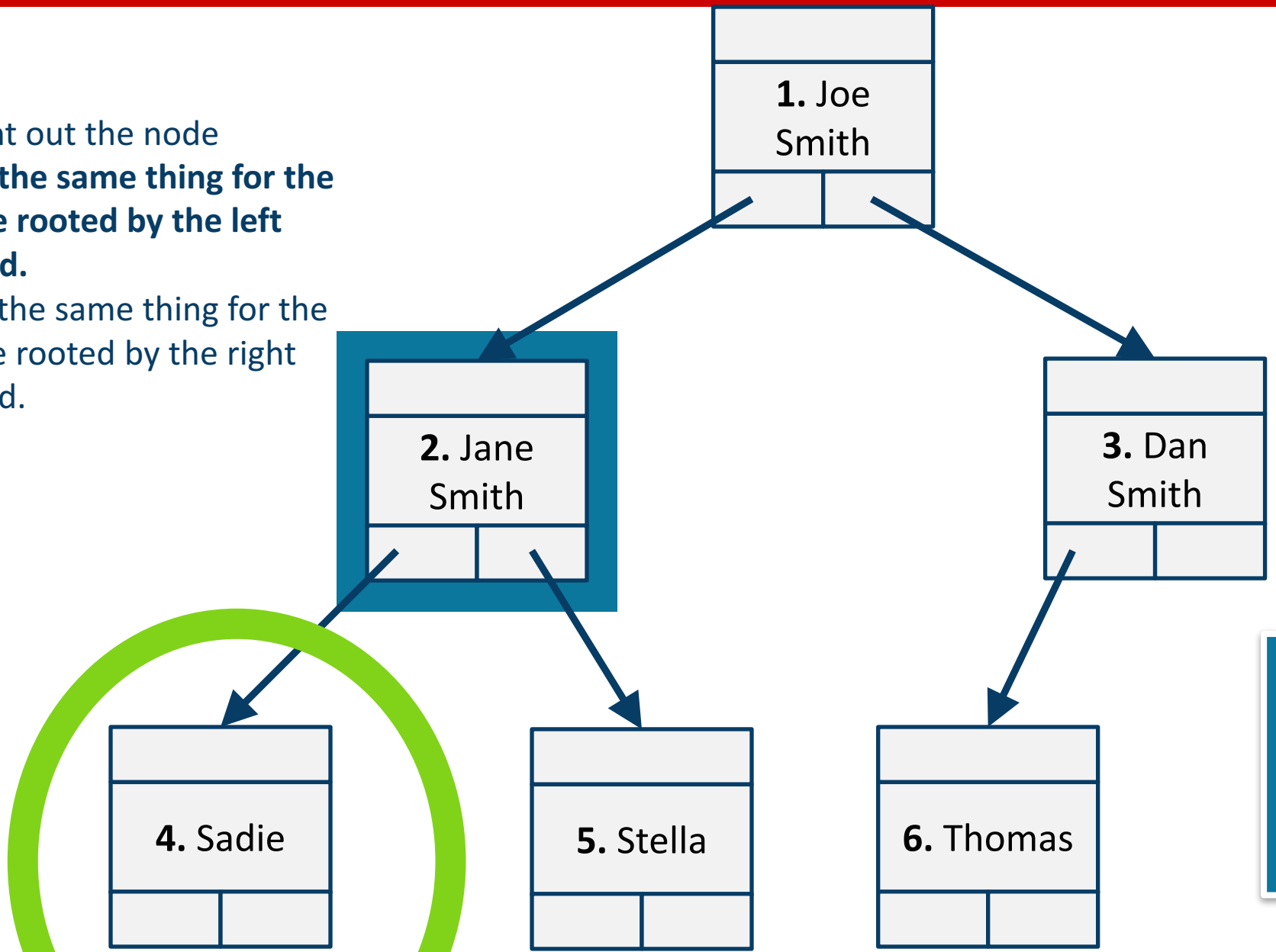
1. **Print out the node**
2. Do the same thing for the tree rooted by the left child.
3. Do the same thing for the tree rooted by the right child.




```
Joe Smith
Jane Smith
```

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

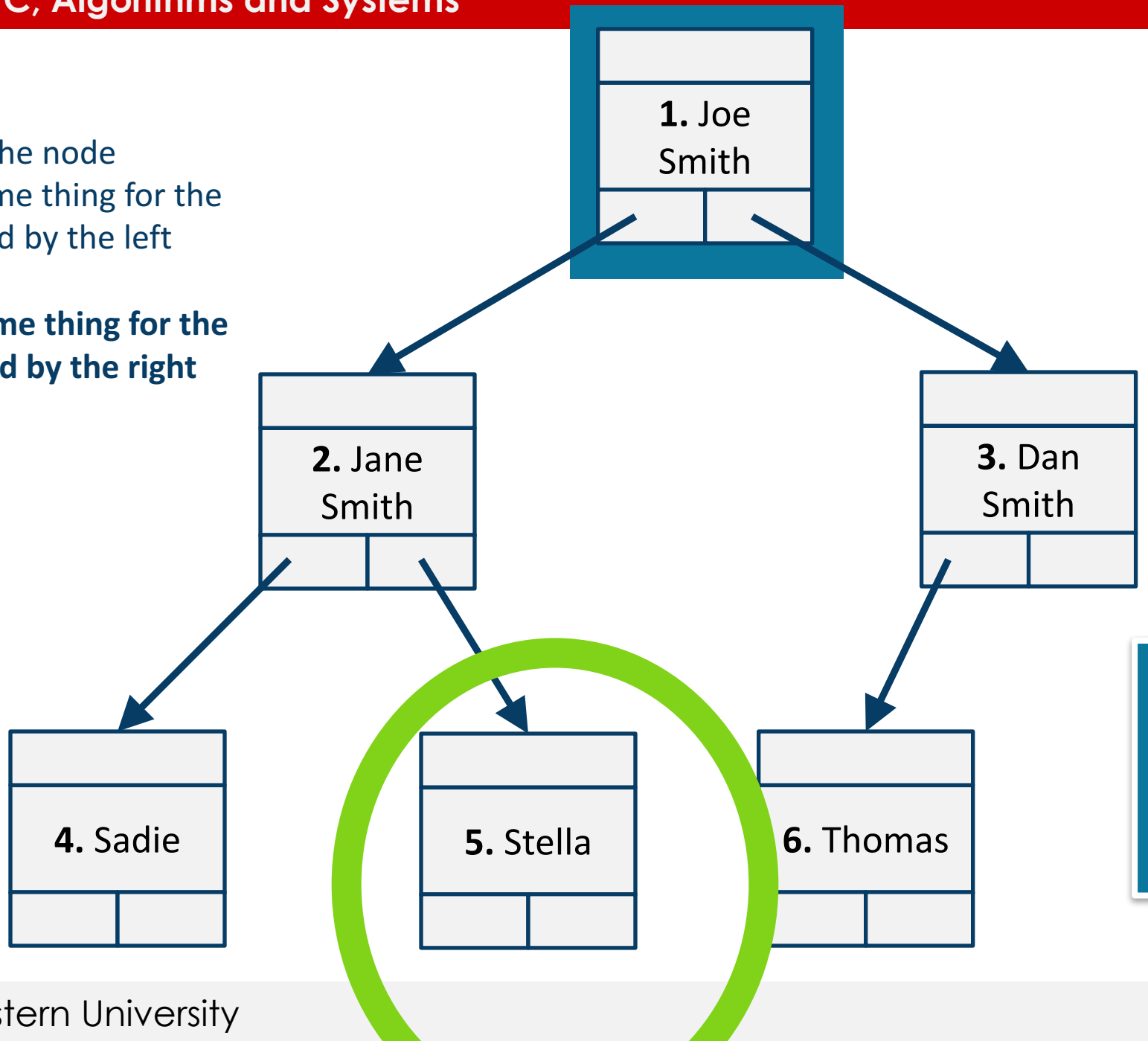
1. Print out the node
2. **Do the same thing for the tree rooted by the left child.**
3. Do the same thing for the tree rooted by the right child.




```
Joe Smith
Jane Smith
```

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

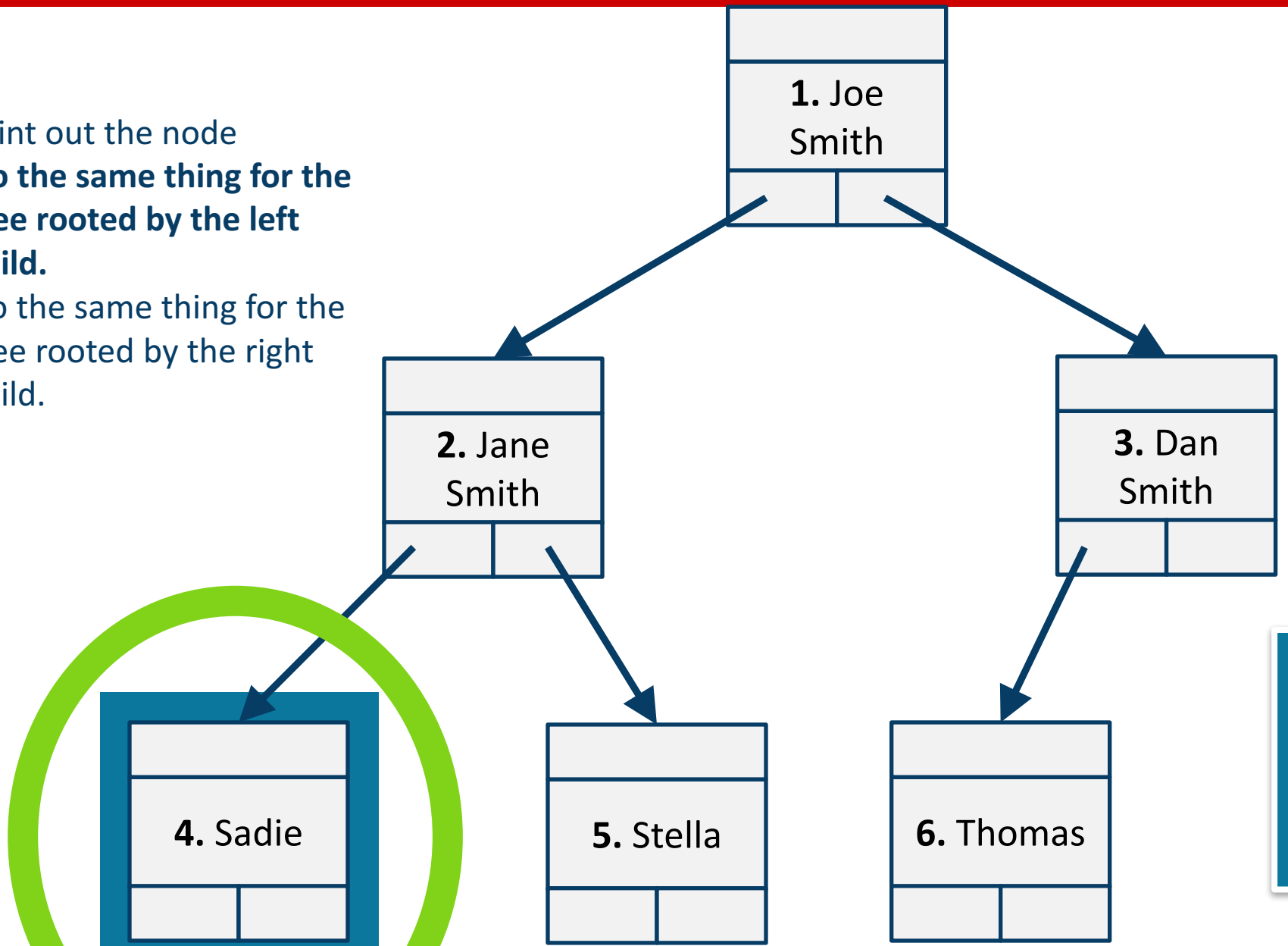
1. Print out the node
2. Do the same thing for the tree rooted by the left child.
3. **Do the same thing for the tree rooted by the right child.**




```
Joe Smith
Jane Smith
```

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

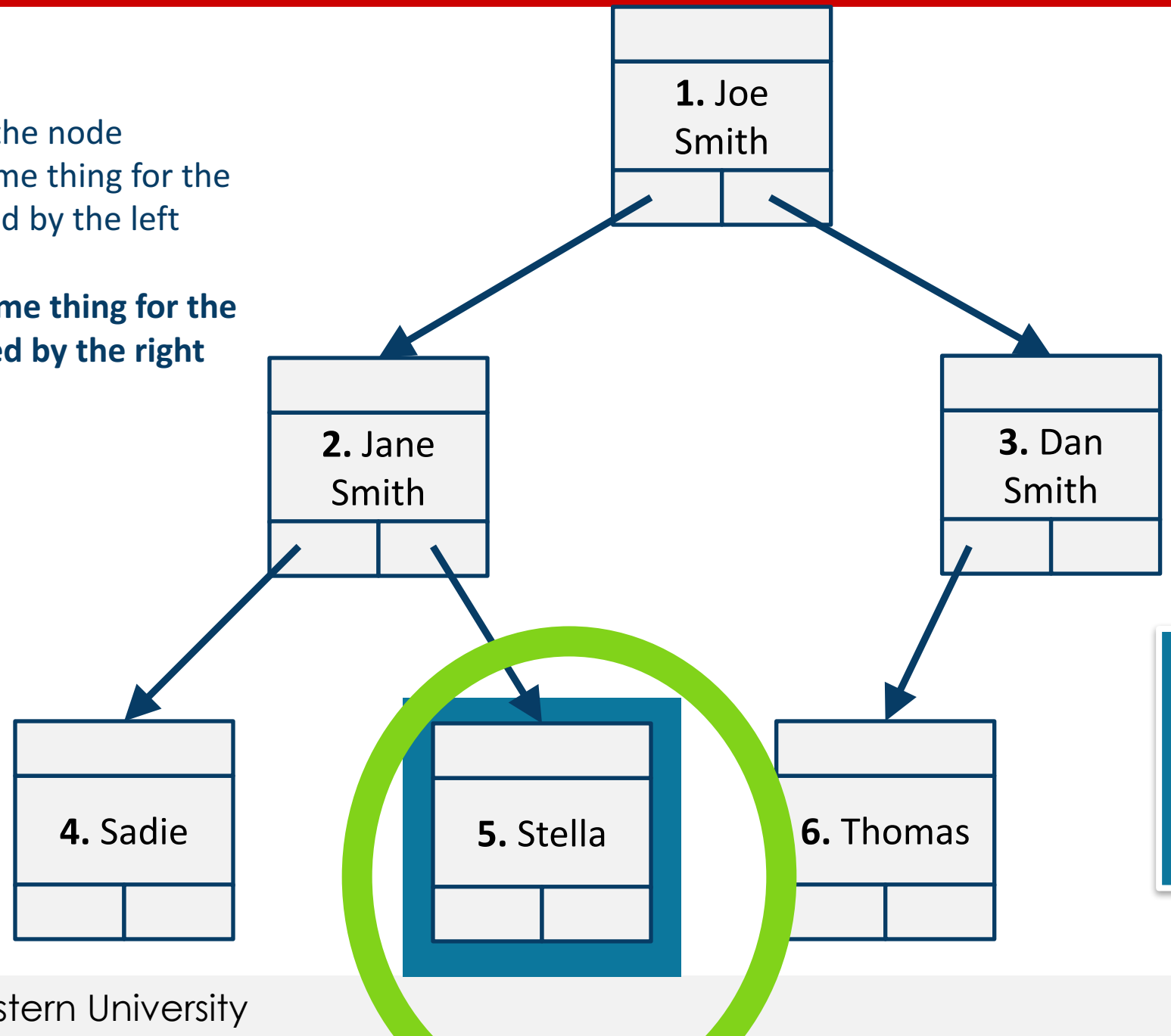
1. Print out the node
2. **Do the same thing for the tree rooted by the left child.**
3. Do the same thing for the tree rooted by the right child.




```
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.

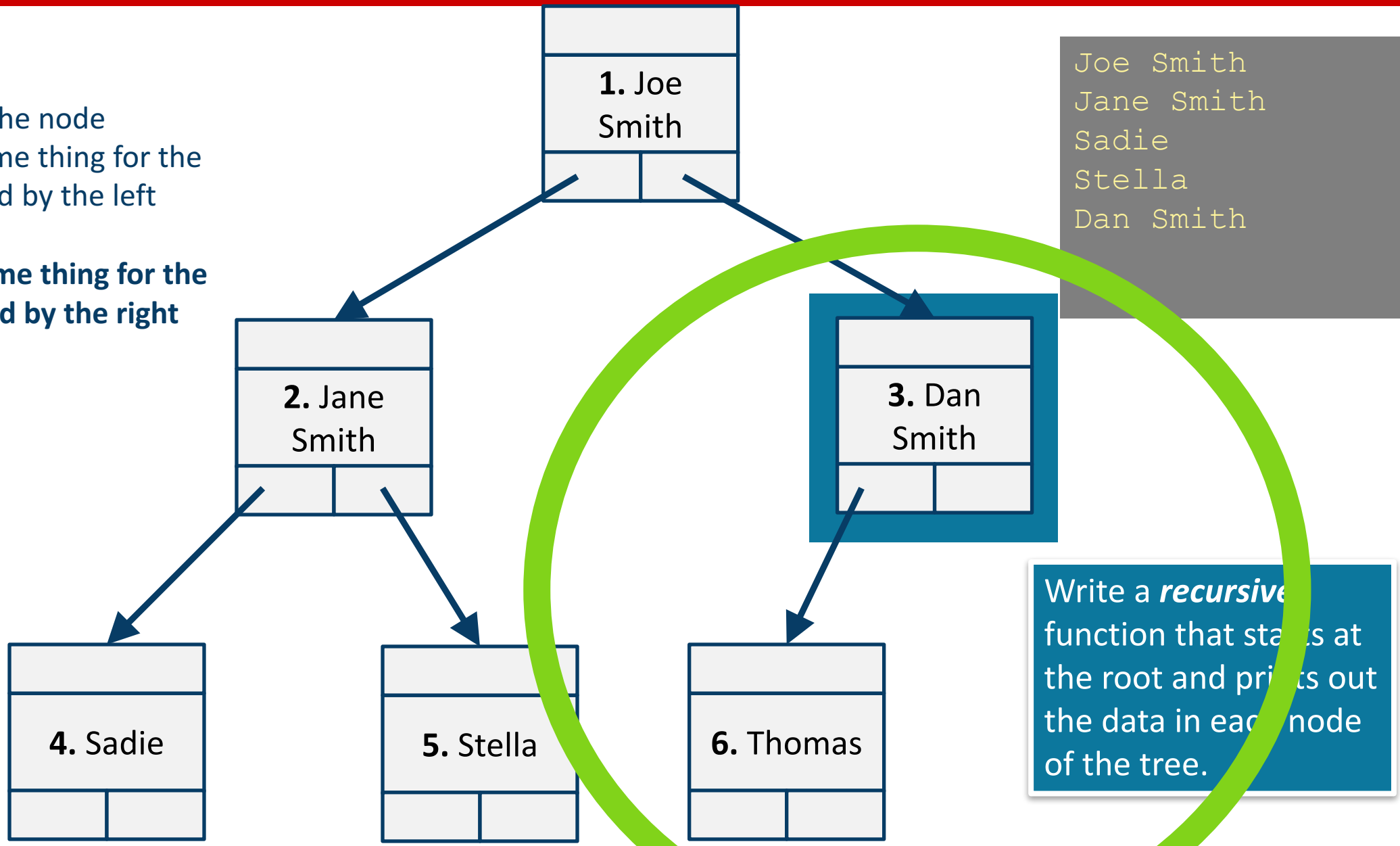
1. Print out the node
2. Do the same thing for the tree rooted by the left child.
3. **Do the same thing for the tree rooted by the right child.**




```
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.

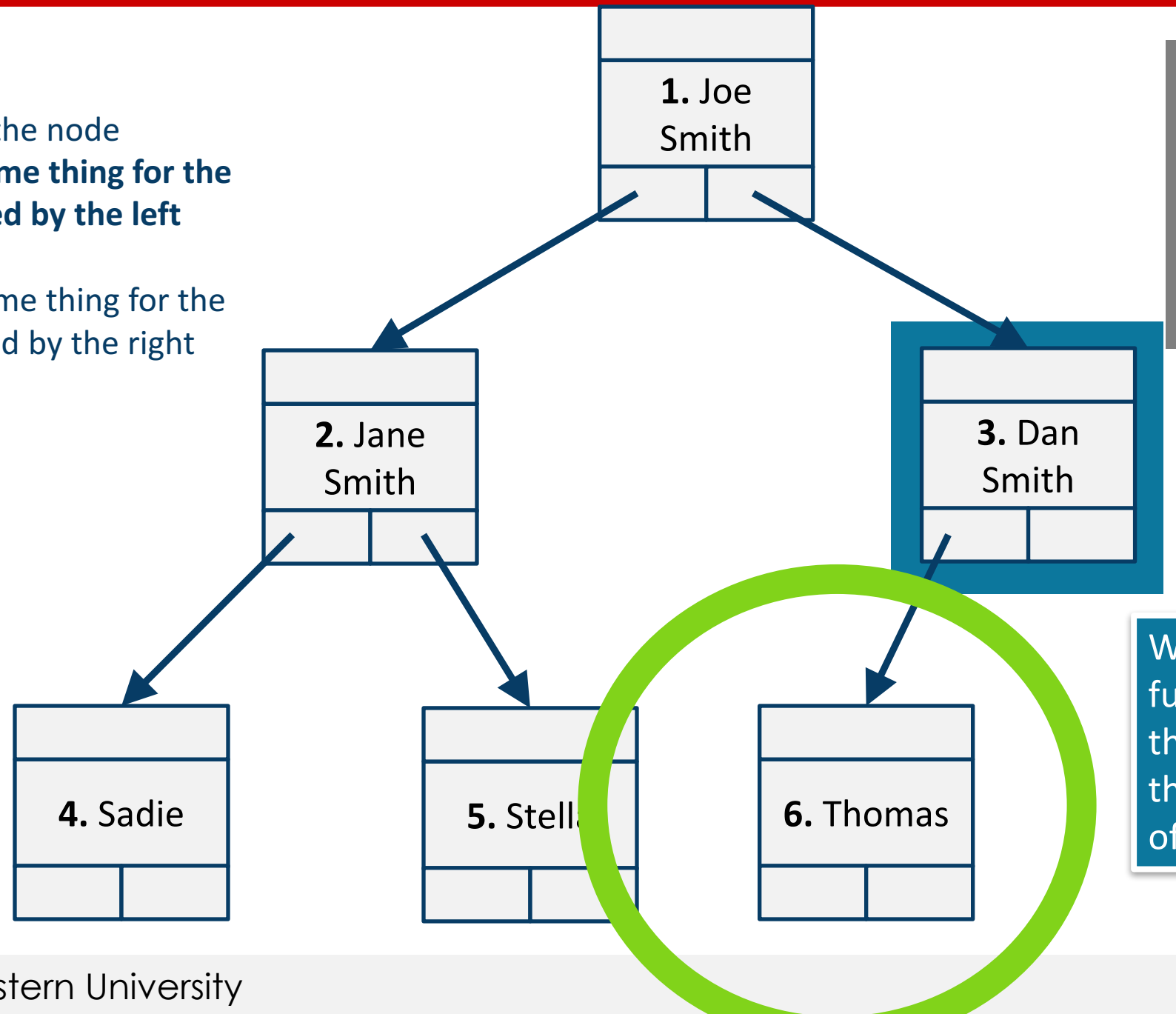
1. Print out the node
2. Do the same thing for the tree rooted by the left child.
3. **Do the same thing for the tree rooted by the right child.**




```
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.

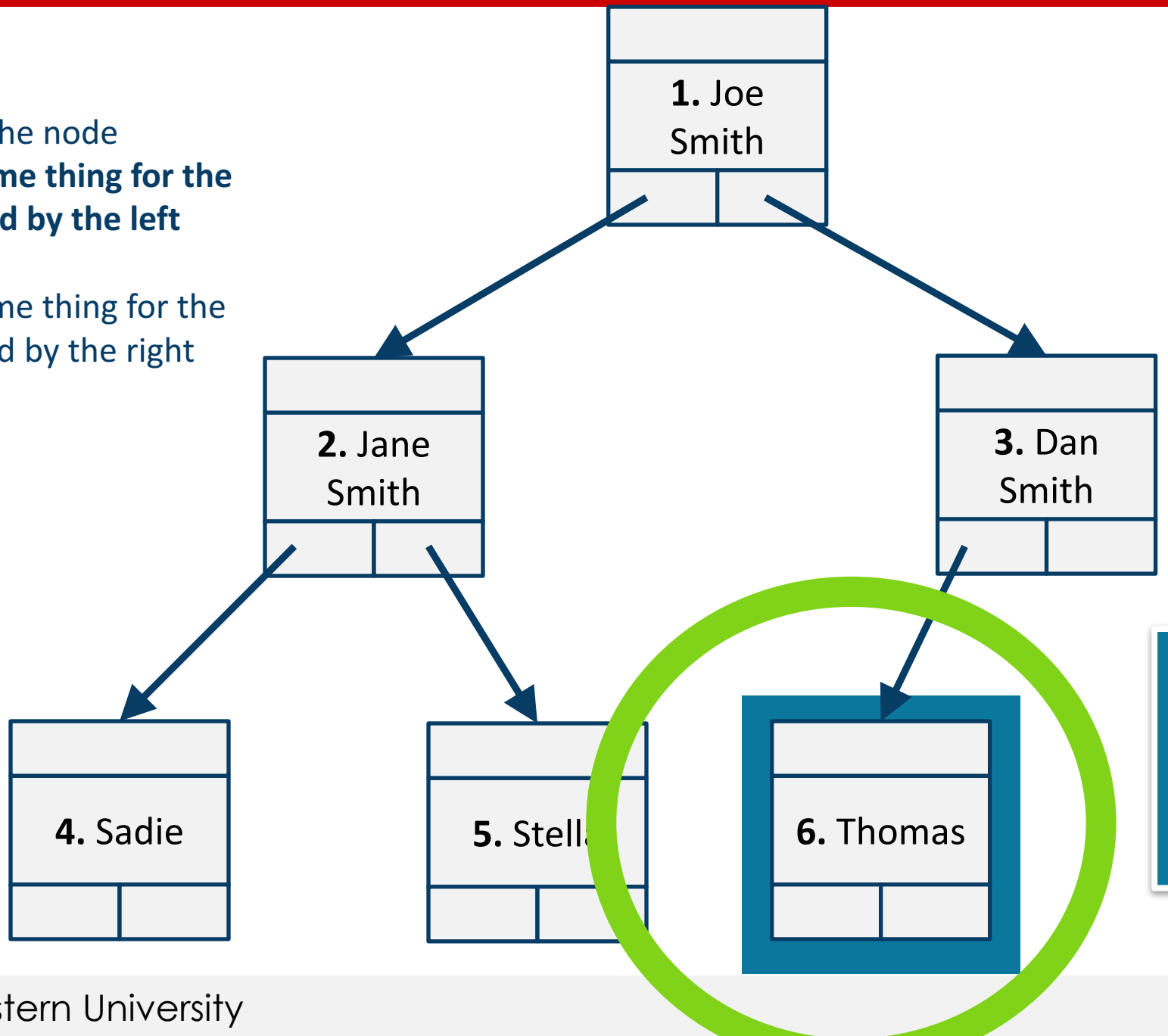
1. Print out the node
2. **Do the same thing for the tree rooted by the left child.**
3. Do the same thing for the tree rooted by the right child.




```
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.

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
```

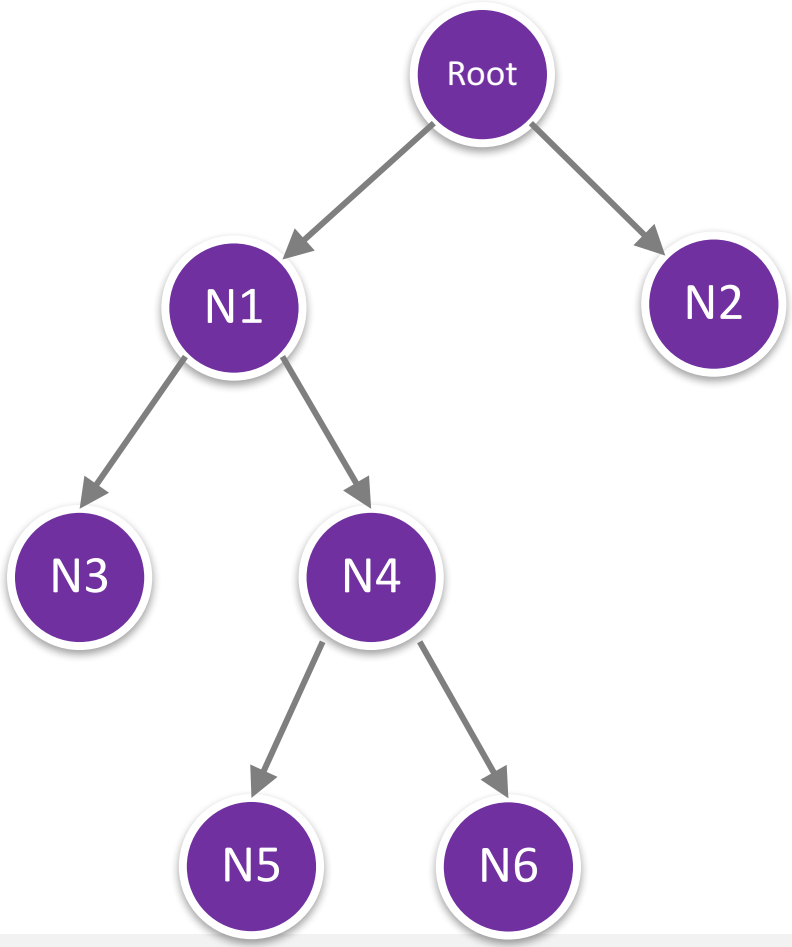
Write a ***recursive*** function that starts at the root and prints out the data in each node of the tree.

Summary

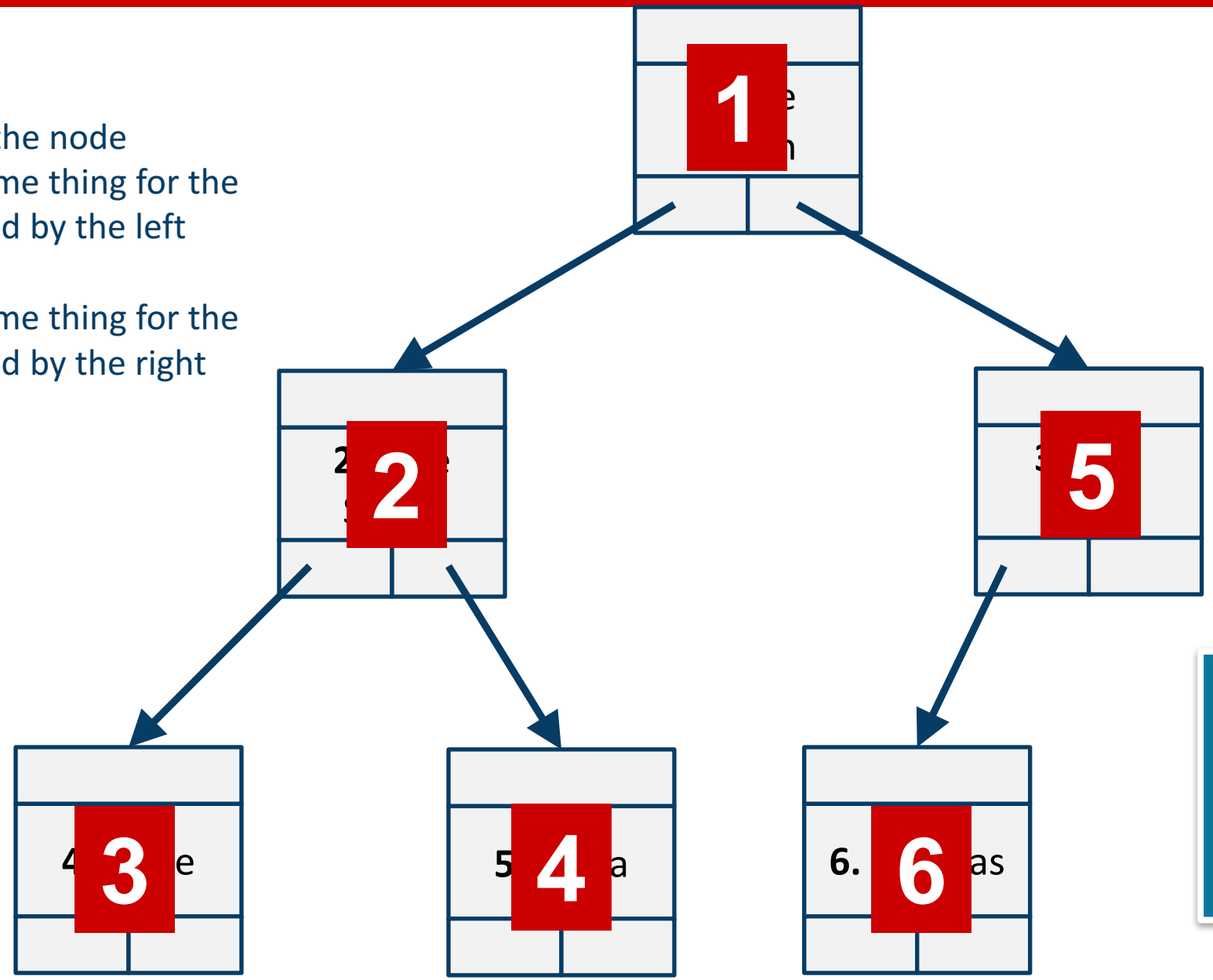
```
void printTree(Node *root){
    printf("%s\n", root->data);
    printTree(root->leftChild);
    printTree(root->rightChild);
}
```

DEMO: See the code in action

The tree in the demo:



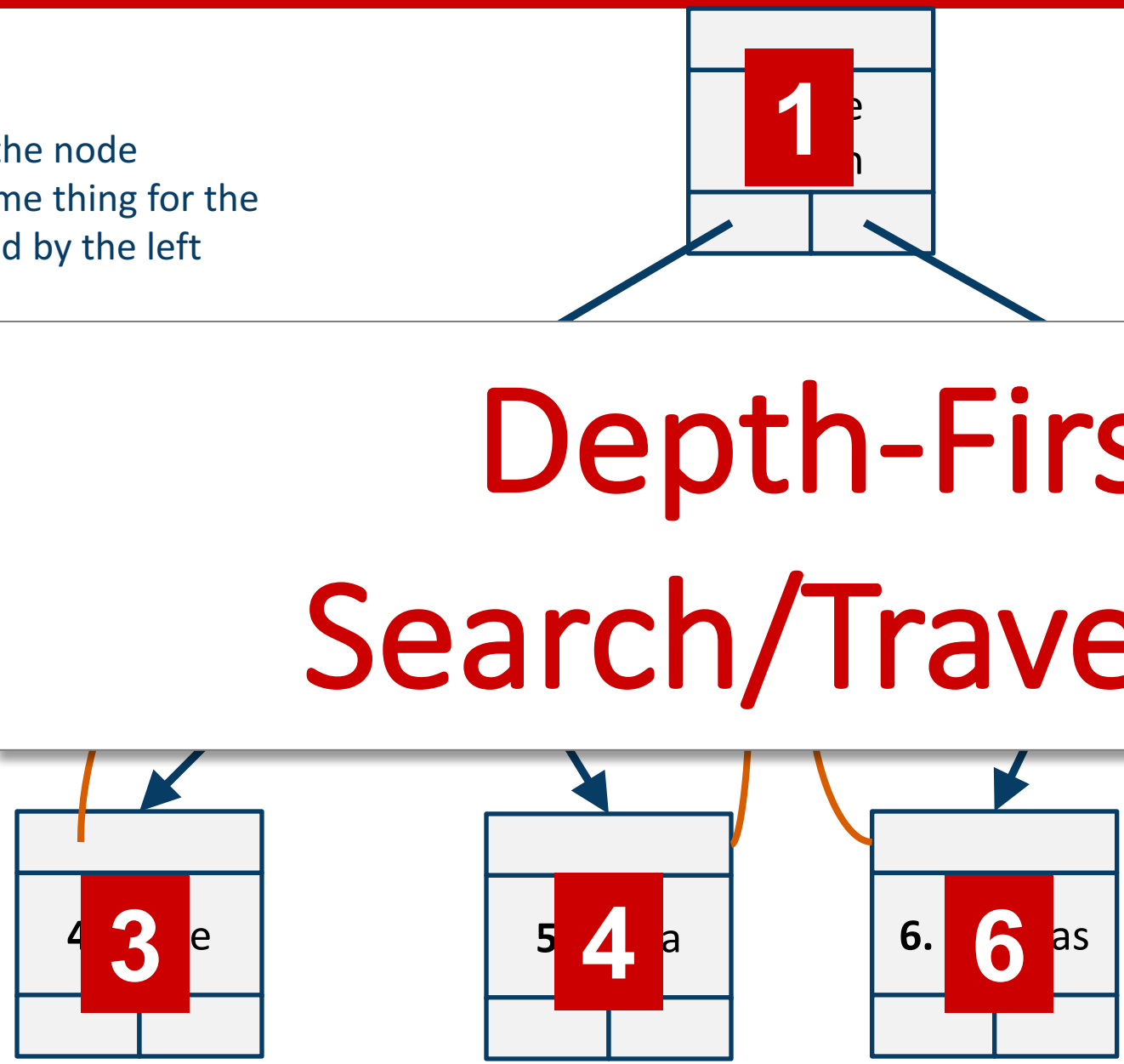
1. Print out the node
2. Do the same thing for the tree rooted by the left child.
3. Do the same thing for the tree rooted by the right child.




```
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.

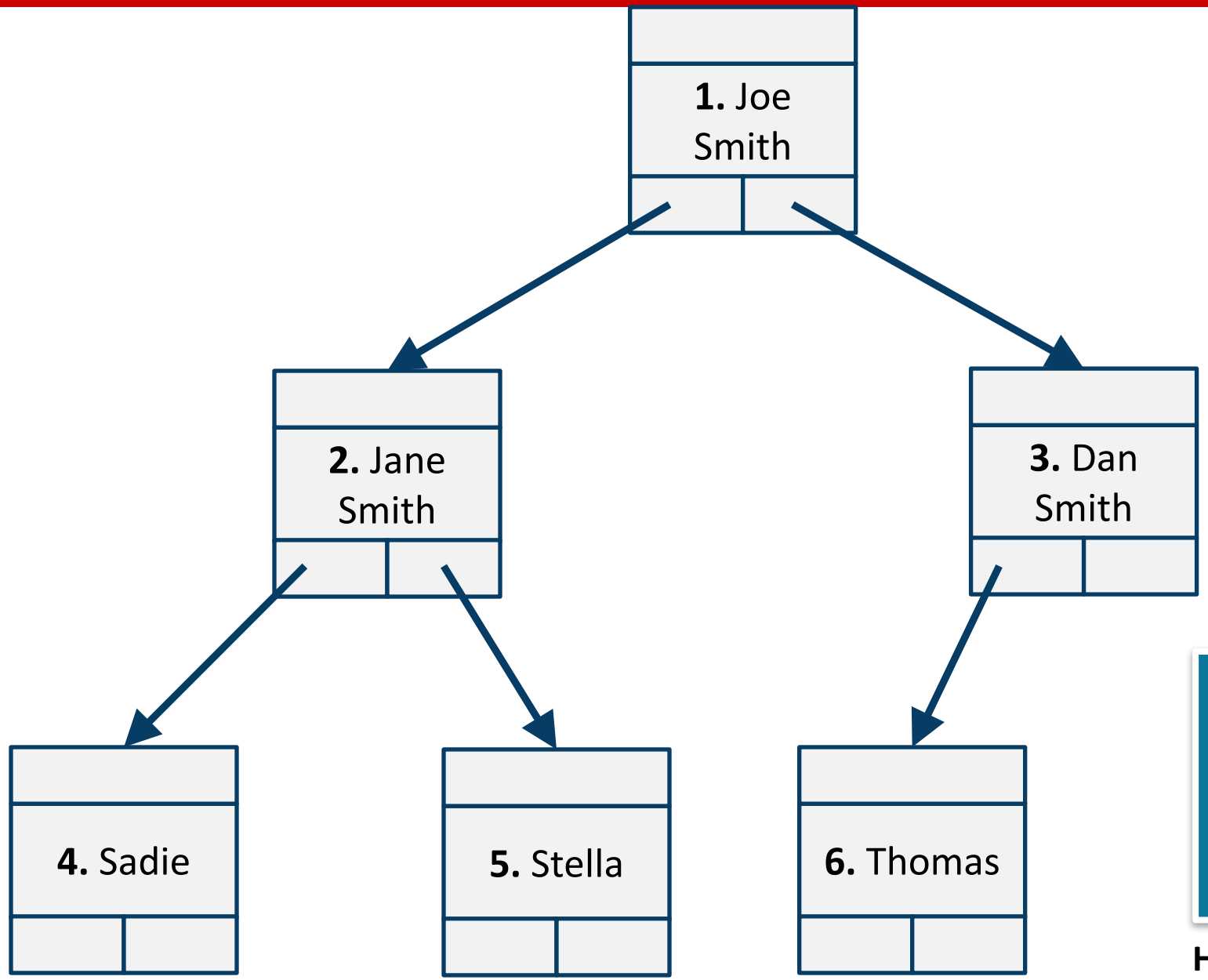
1. Print out the node
2. Do the same thing for the tree rooted by the left child.
3. Do the sa
   tree roote
   child.




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

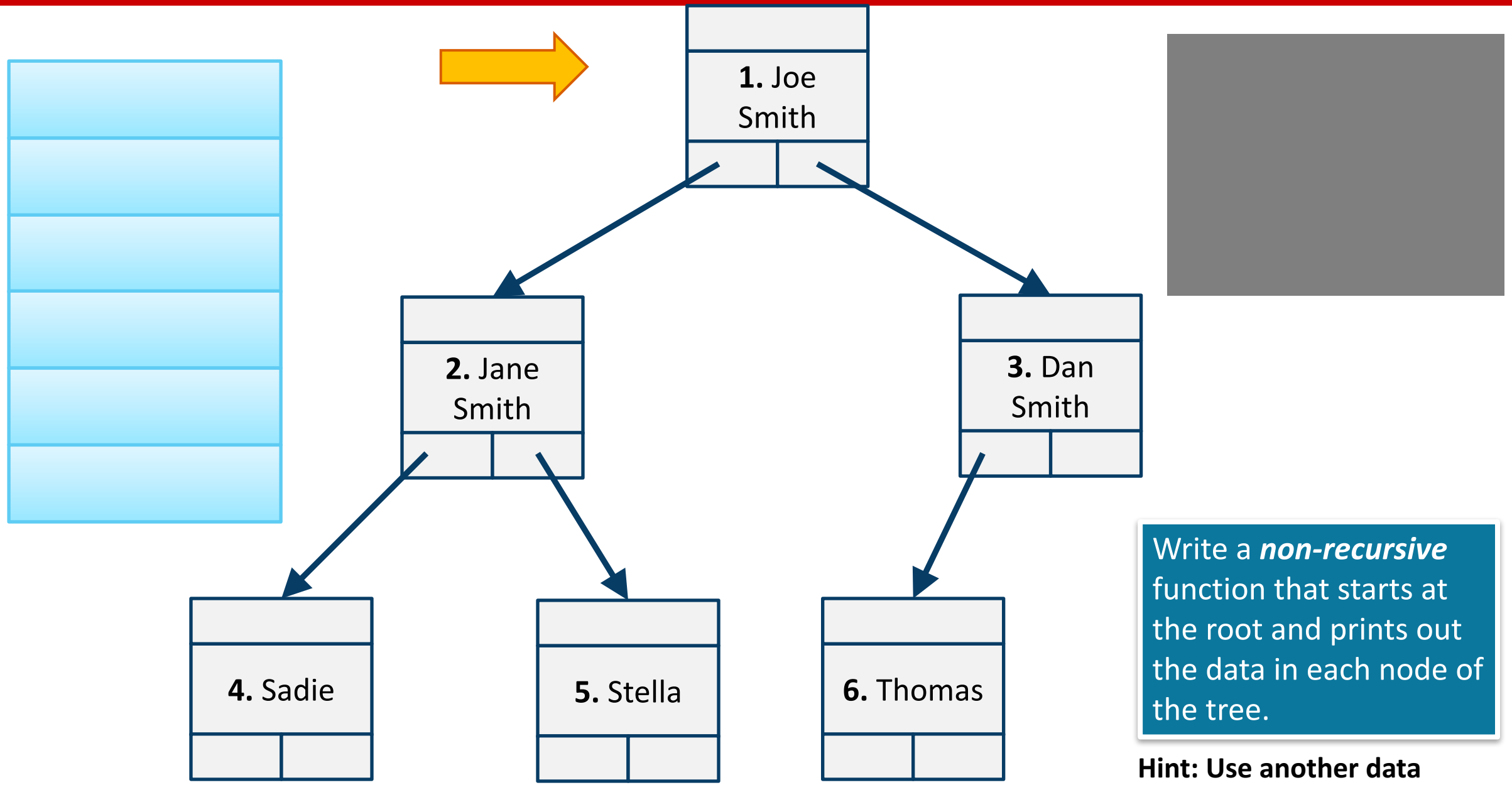
Depth-First Search/Traversal

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



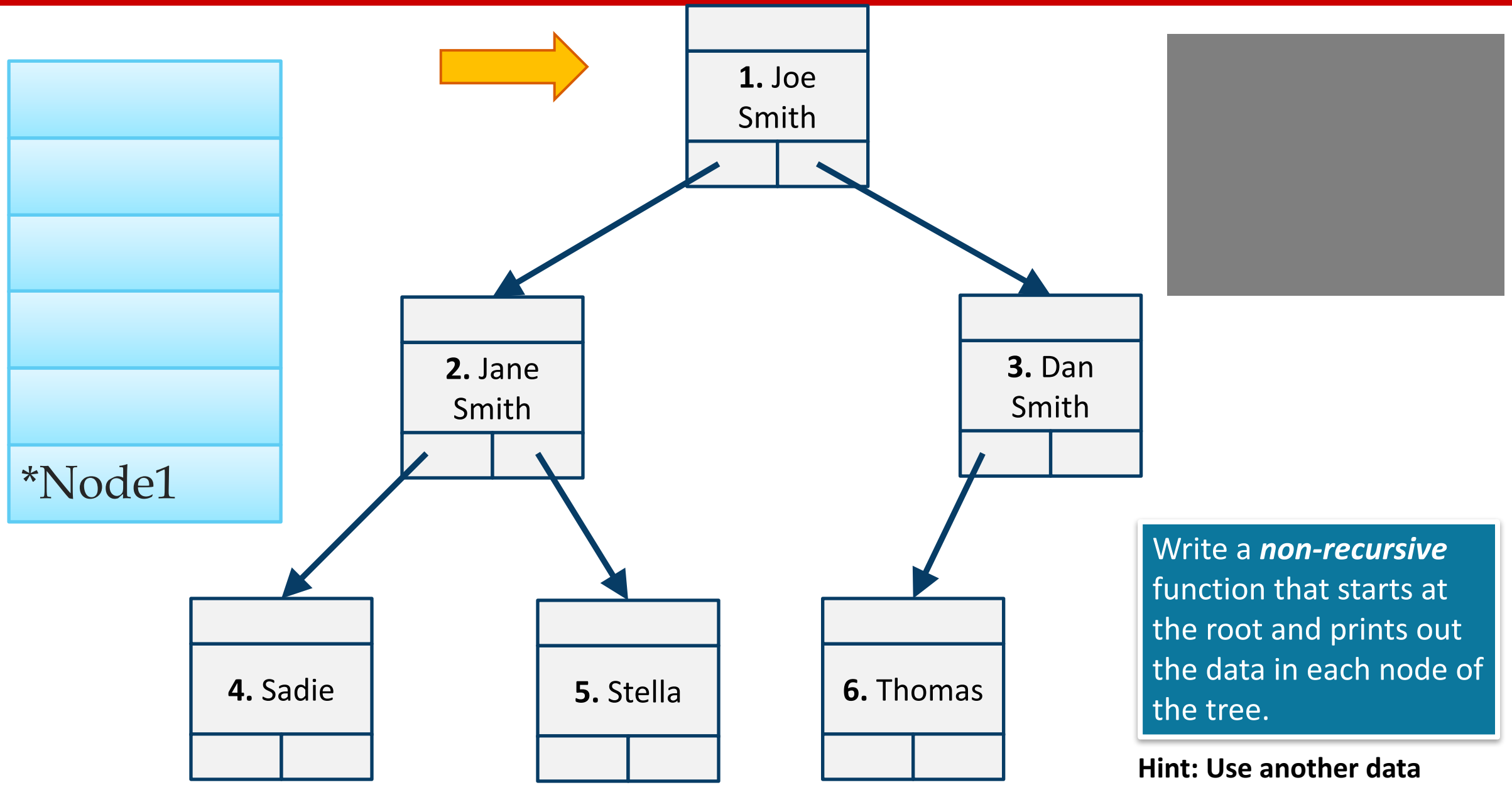
Write a ***non-recursive*** function that starts at the root and prints out the data in each node of the tree.

Hint: Use another data structure as part of your solution.



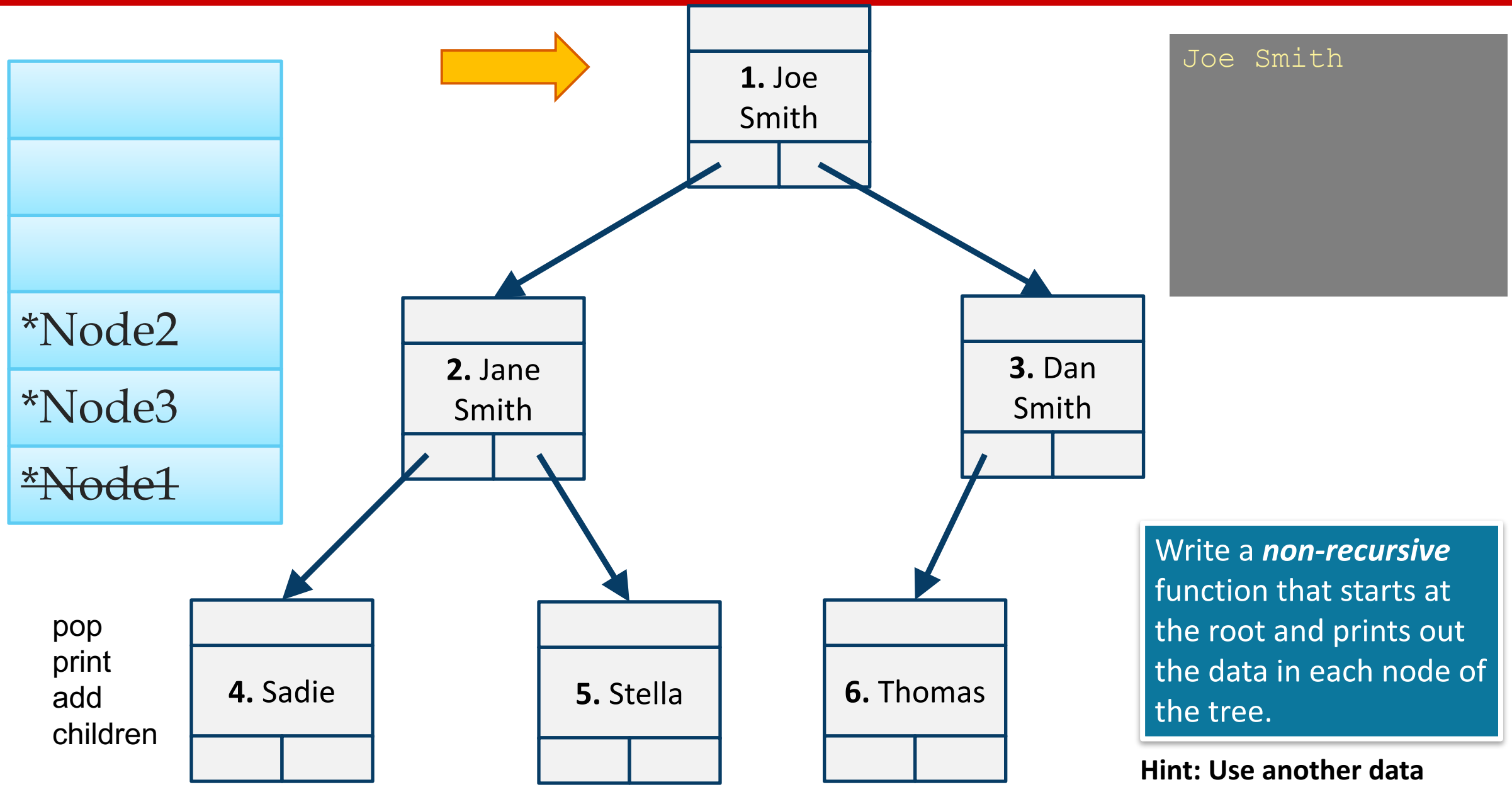
Write a ***non-recursive*** function that starts at the root and prints out the data in each node of the tree.

Hint: Use another data structure as part of your solution.



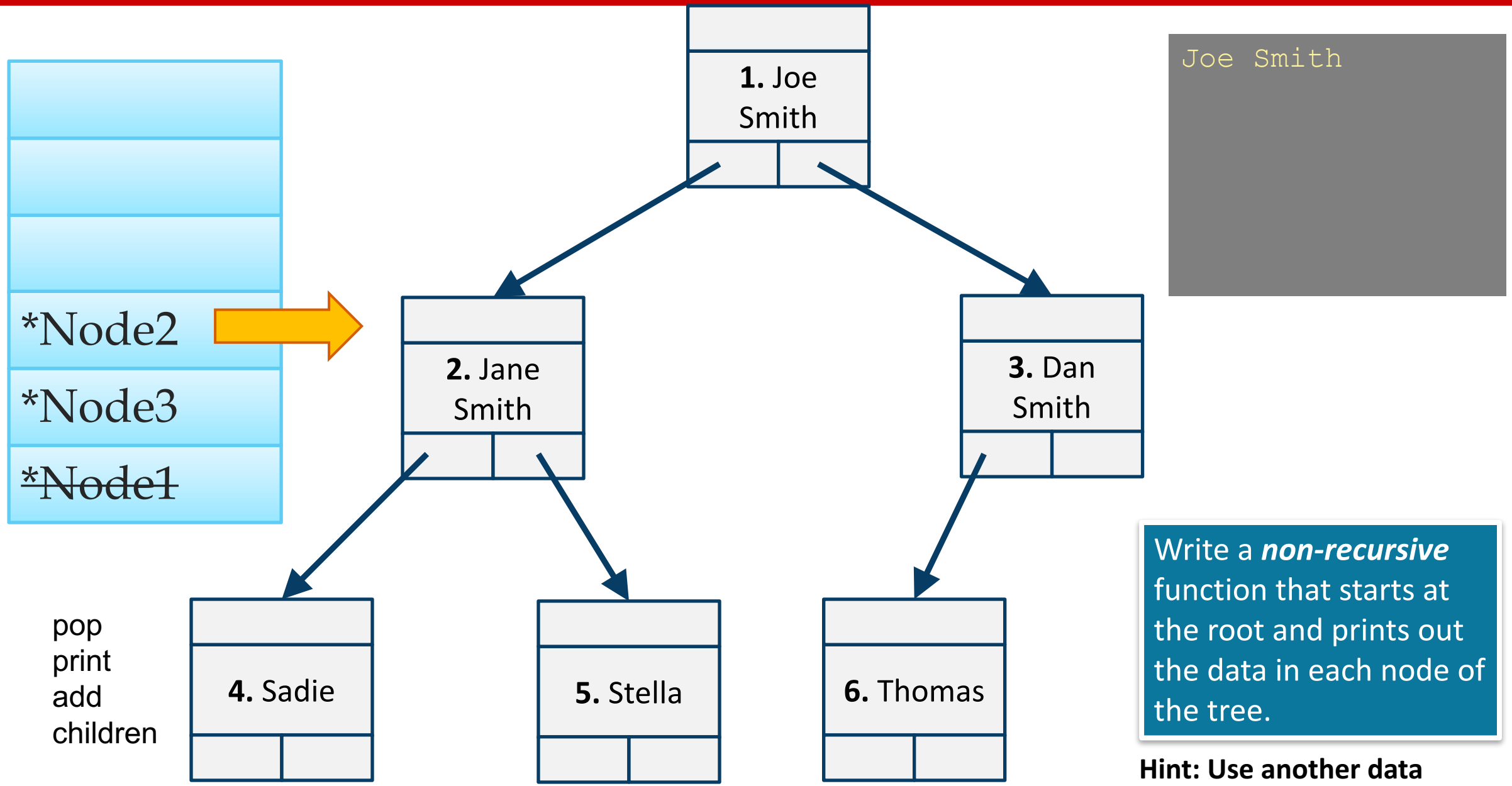
Write a ***non-recursive*** function that starts at the root and prints out the data in each node of the tree.

Hint: Use another data structure as part of your solution.



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

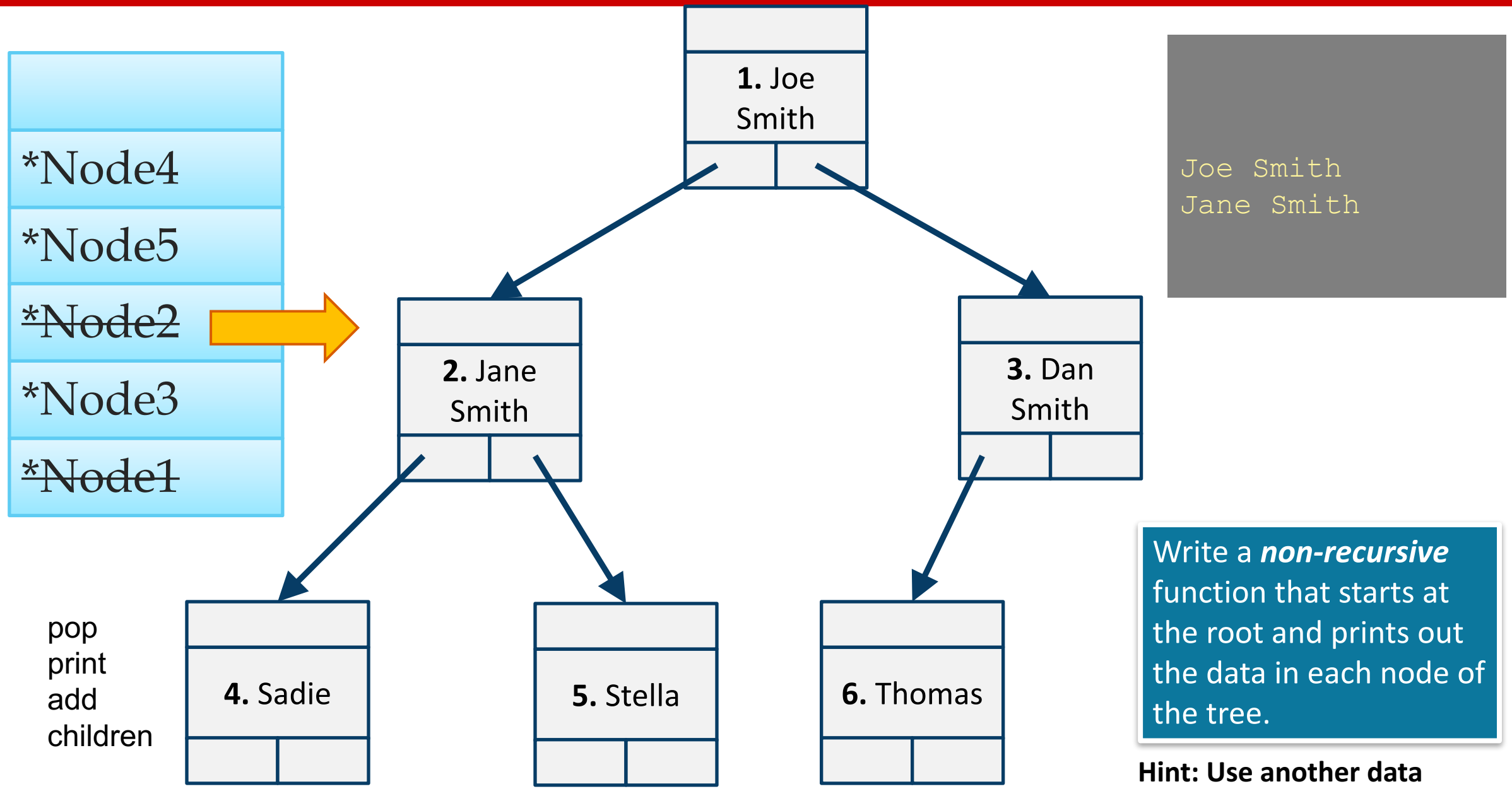
Hint: Use another data structure as part of your solution.




```
Joe Smith
```

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

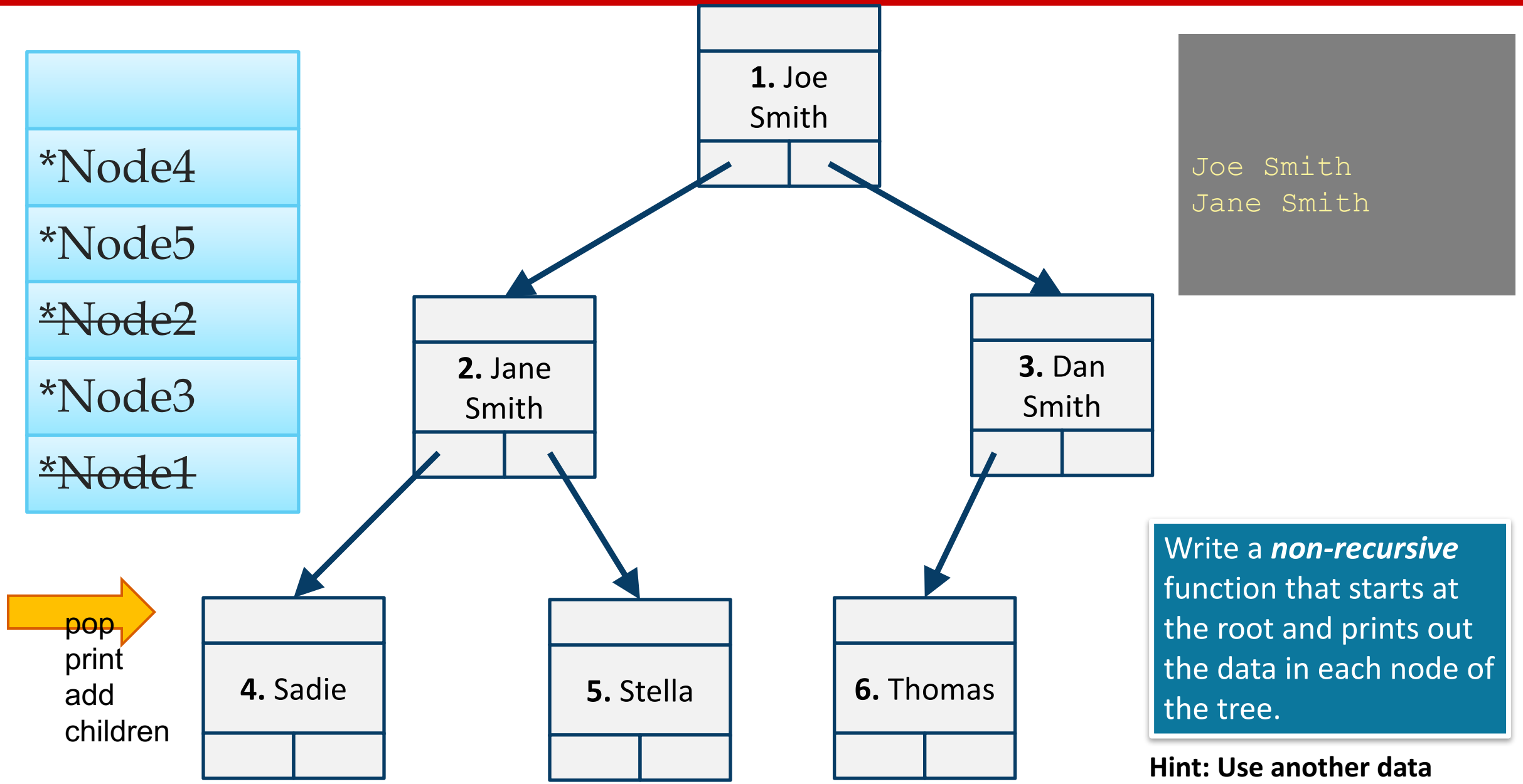
Hint: Use another data structure as part of your solution.




```
Joe Smith
Jane Smith
```

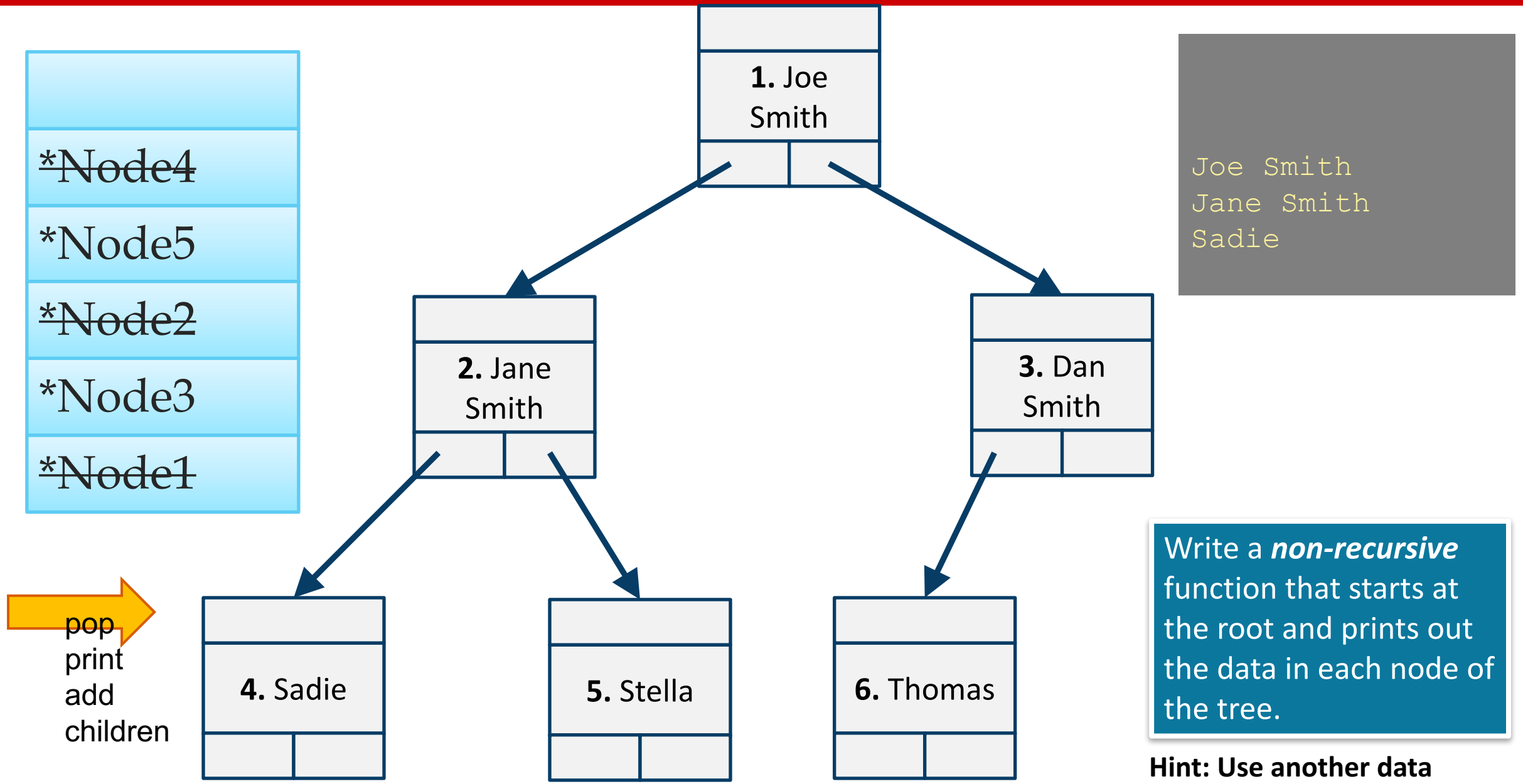
Write a ***non-recursive*** function that starts at the root and prints out the data in each node of the tree.

Hint: Use another data structure as part of your solution.



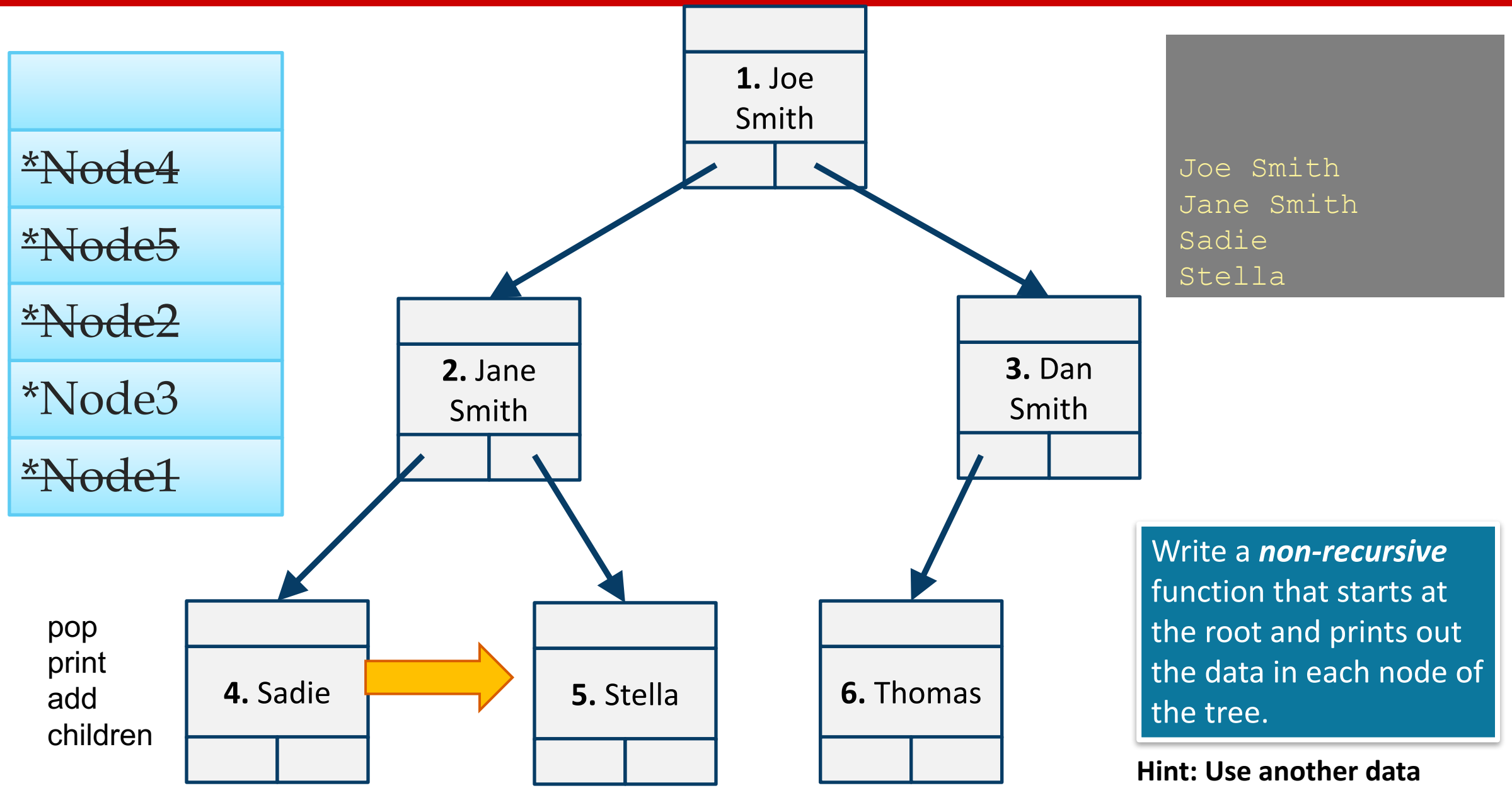
Write a ***non-recursive*** function that starts at the root and prints out the data in each node of the tree.

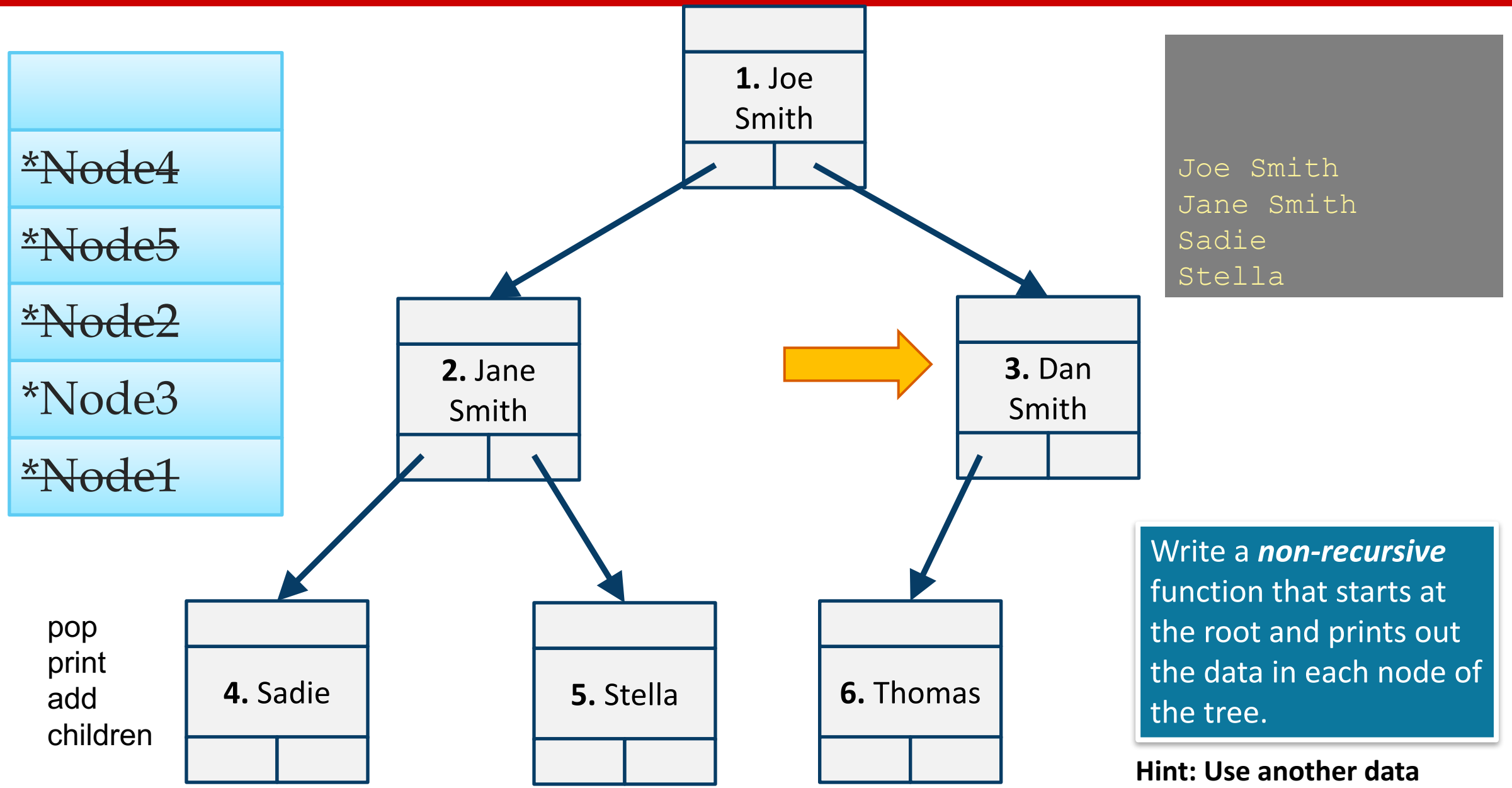
Hint: Use another data structure as part of your solution.



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

Hint: Use another data structure as part of your solution.

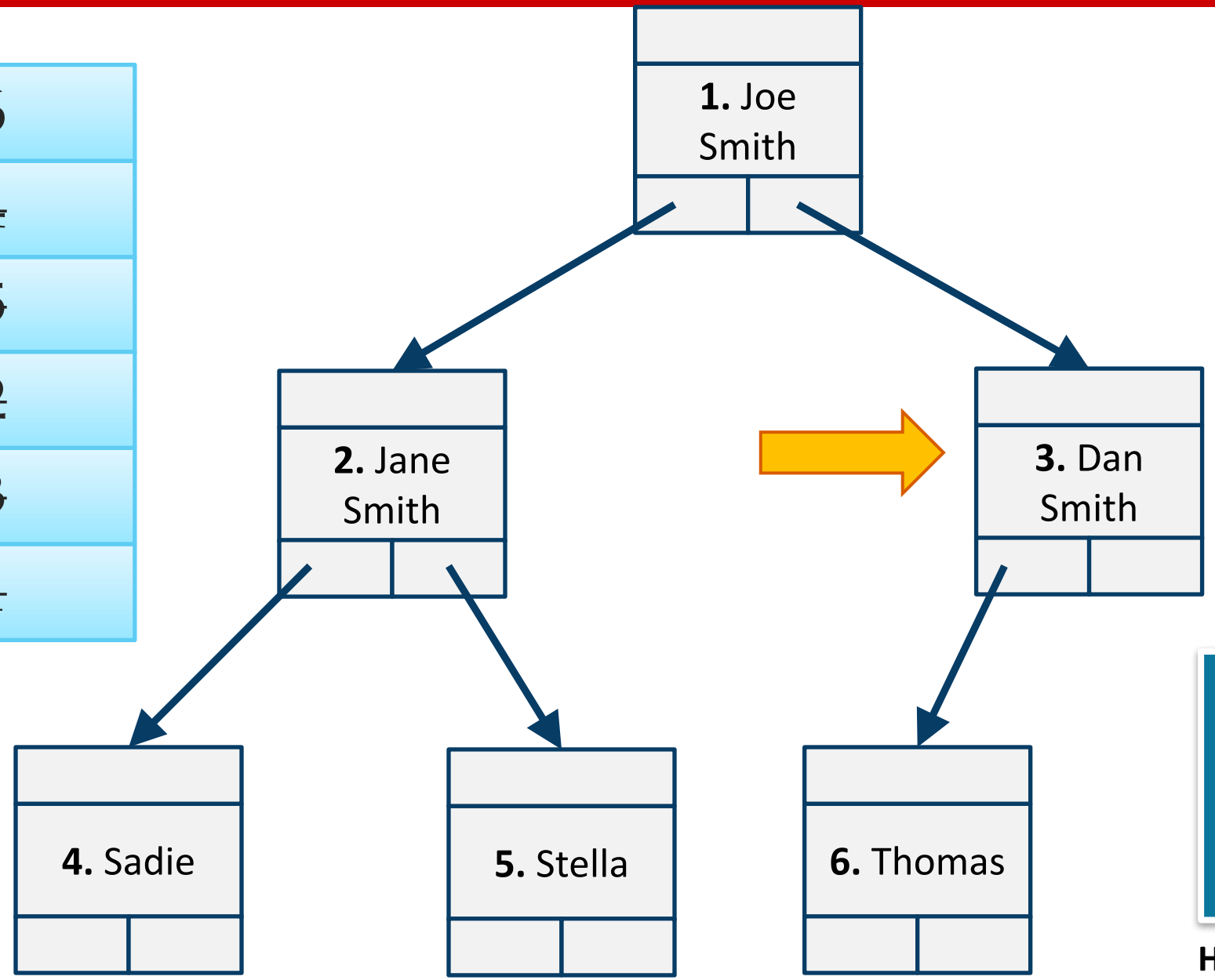




Write a ***non-recursive*** function that starts at the root and prints out the data in each node of the tree.

Hint: Use another data structure as part of your solution.

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



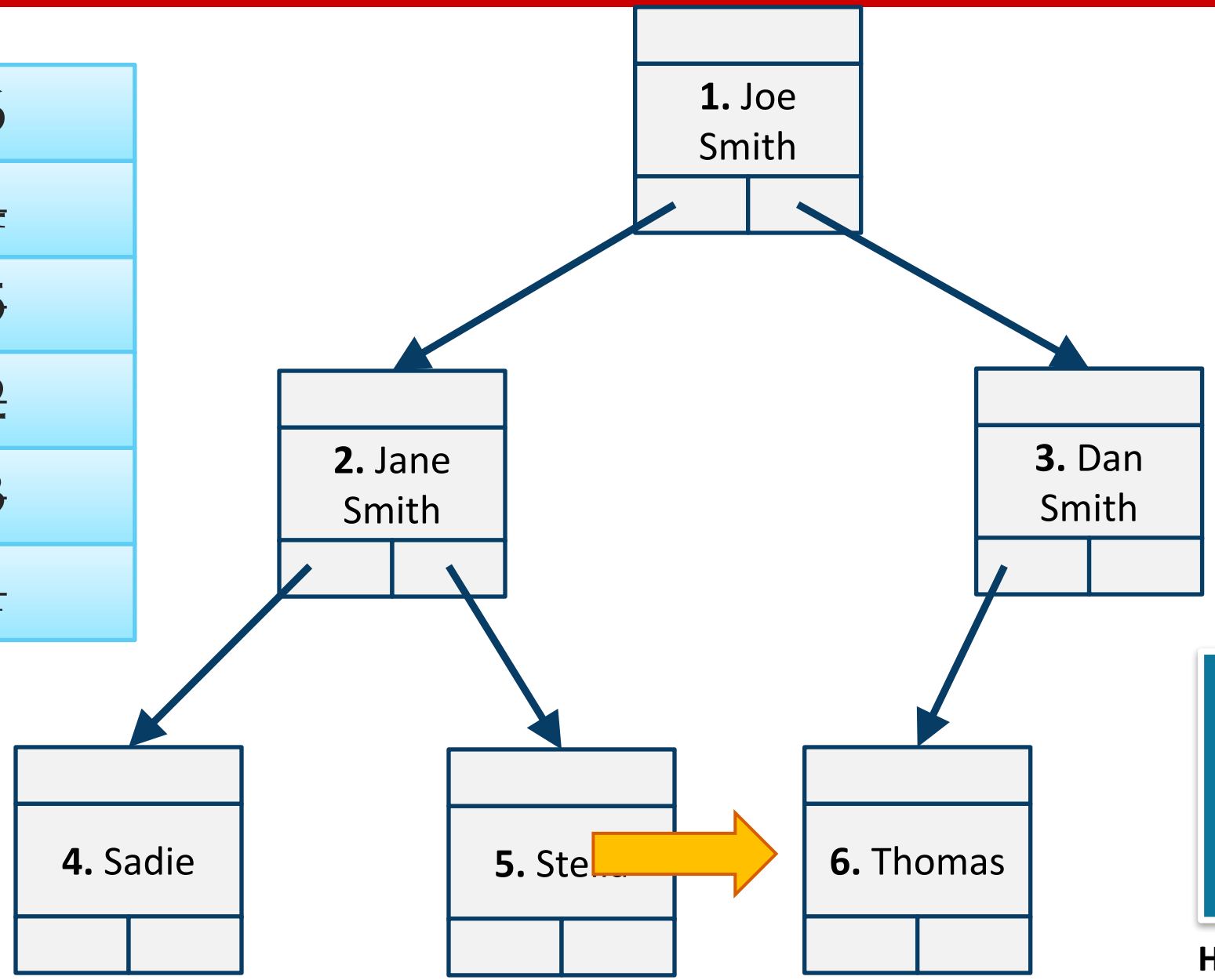

```
Joe Smith
Jane Smith
Sadie
Stella
Dan Smith
```

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.

Hint: Use another data structure as part of your solution.

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




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

pop
print
add
children

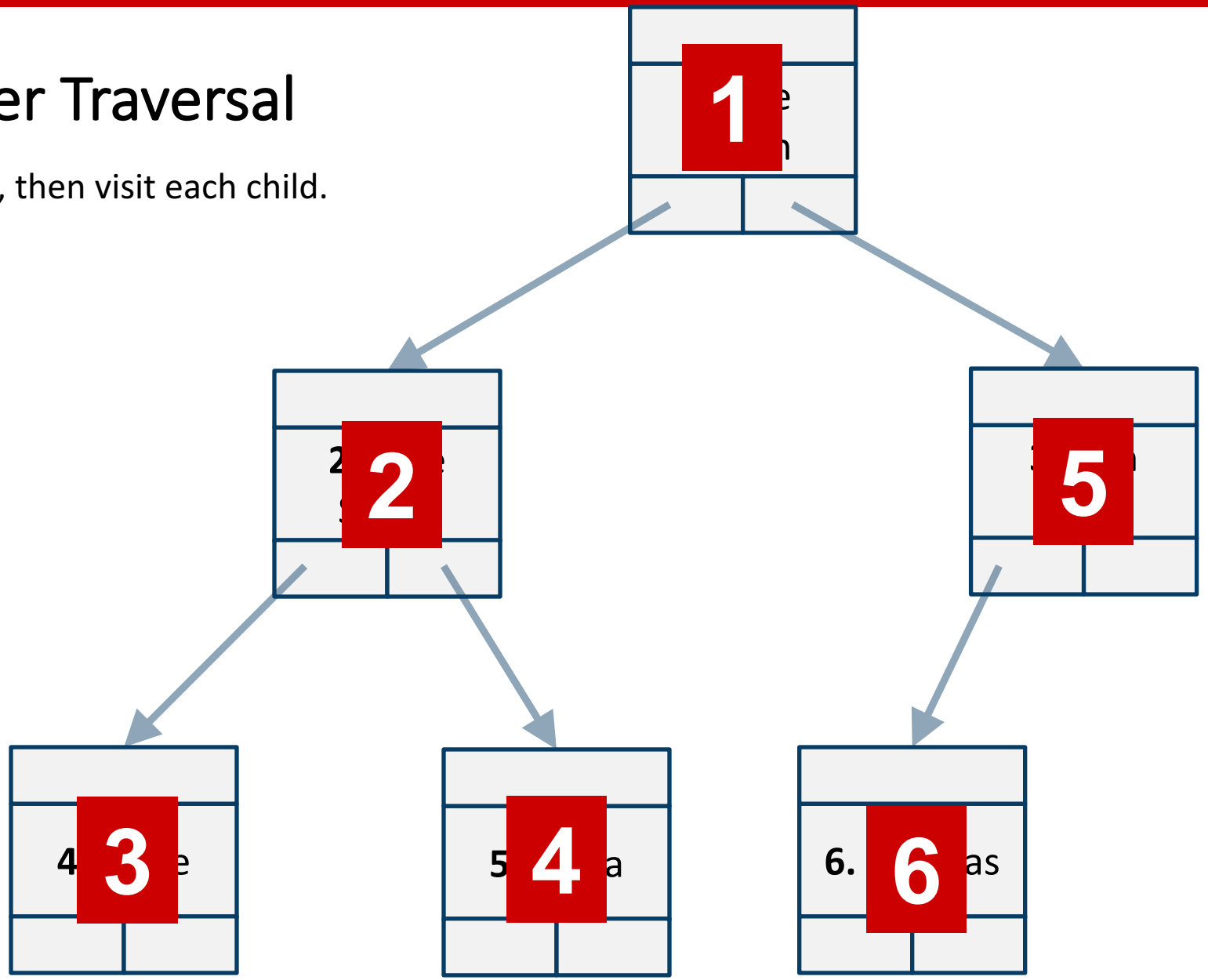
Write a ***non-recursive*** function that starts at the root and prints out the data in each node of the tree.

Hint: Use another data structure as part of your solution.

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

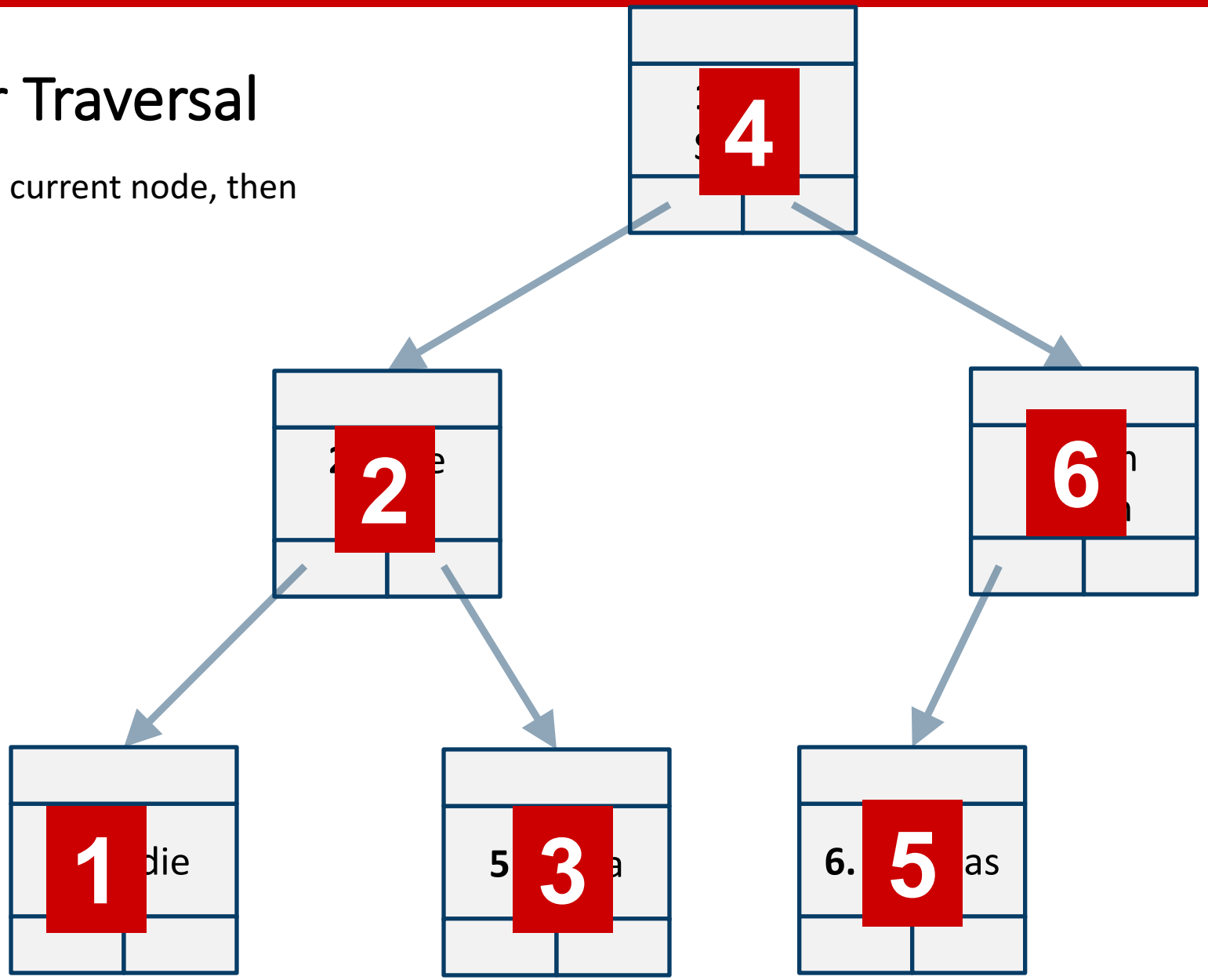
Pre-order Traversal

Visit the node, then visit each child.



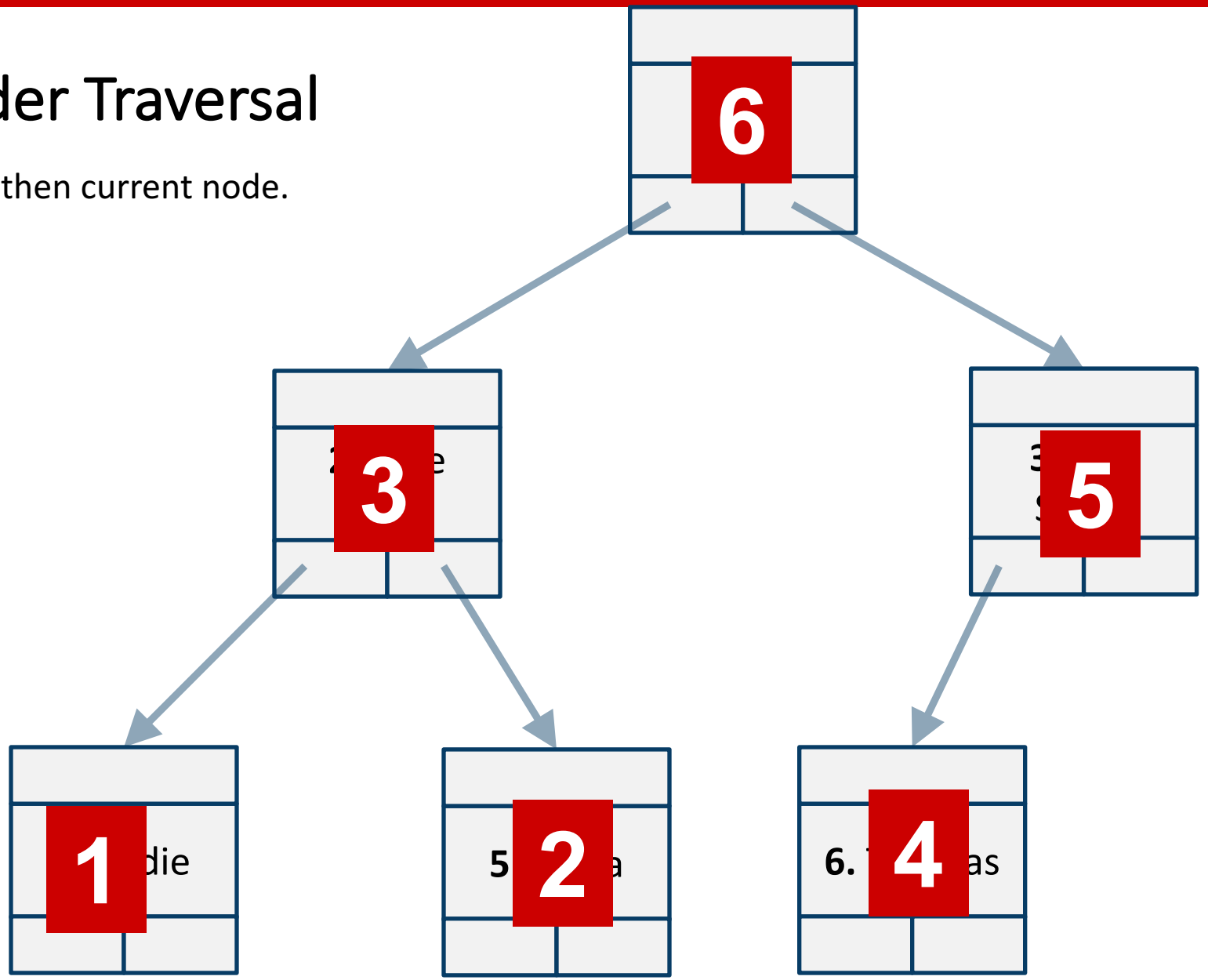
In-order Traversal

Visit left child, current node, then right child.



Post-order Traversal

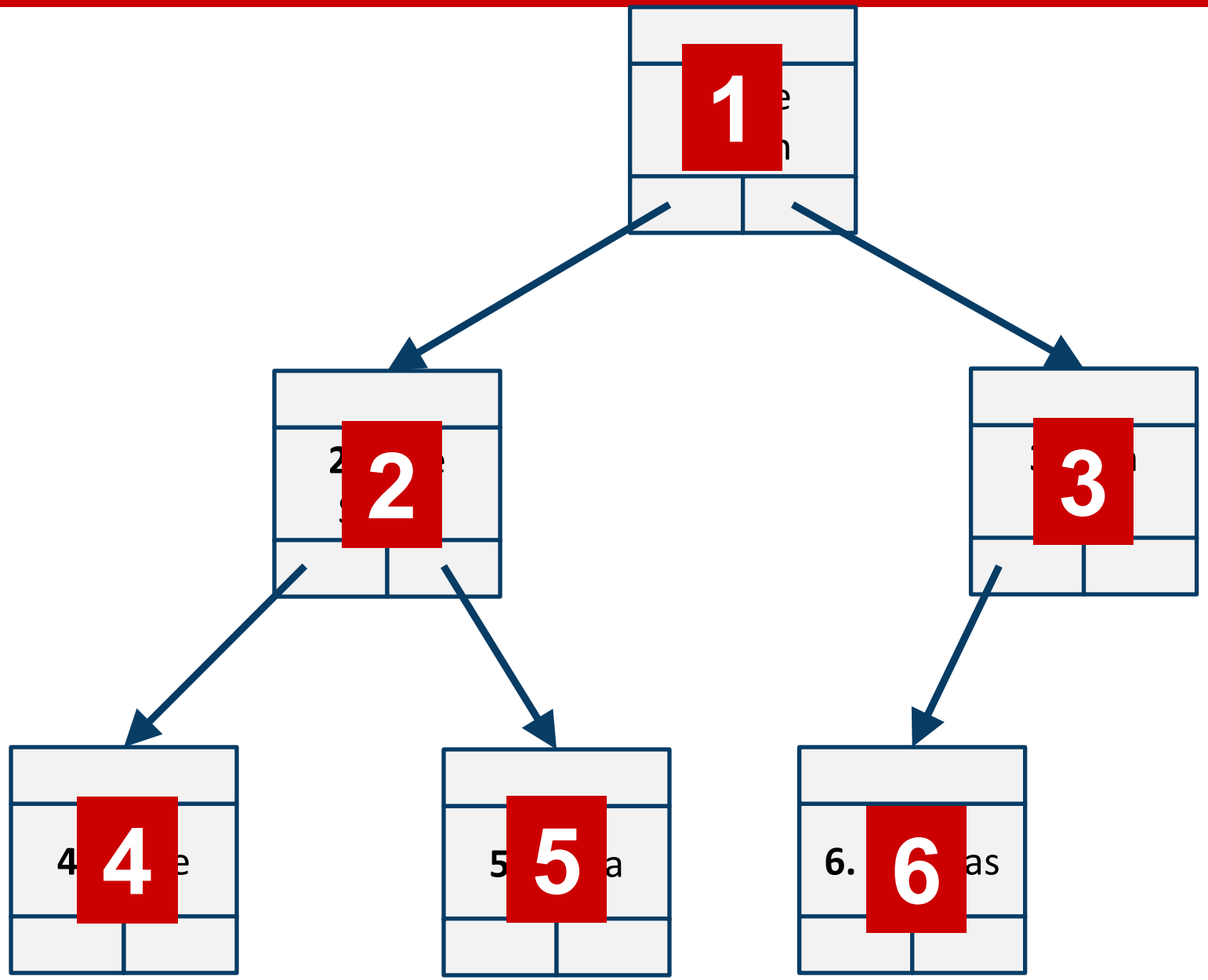
Visit children, then current node.

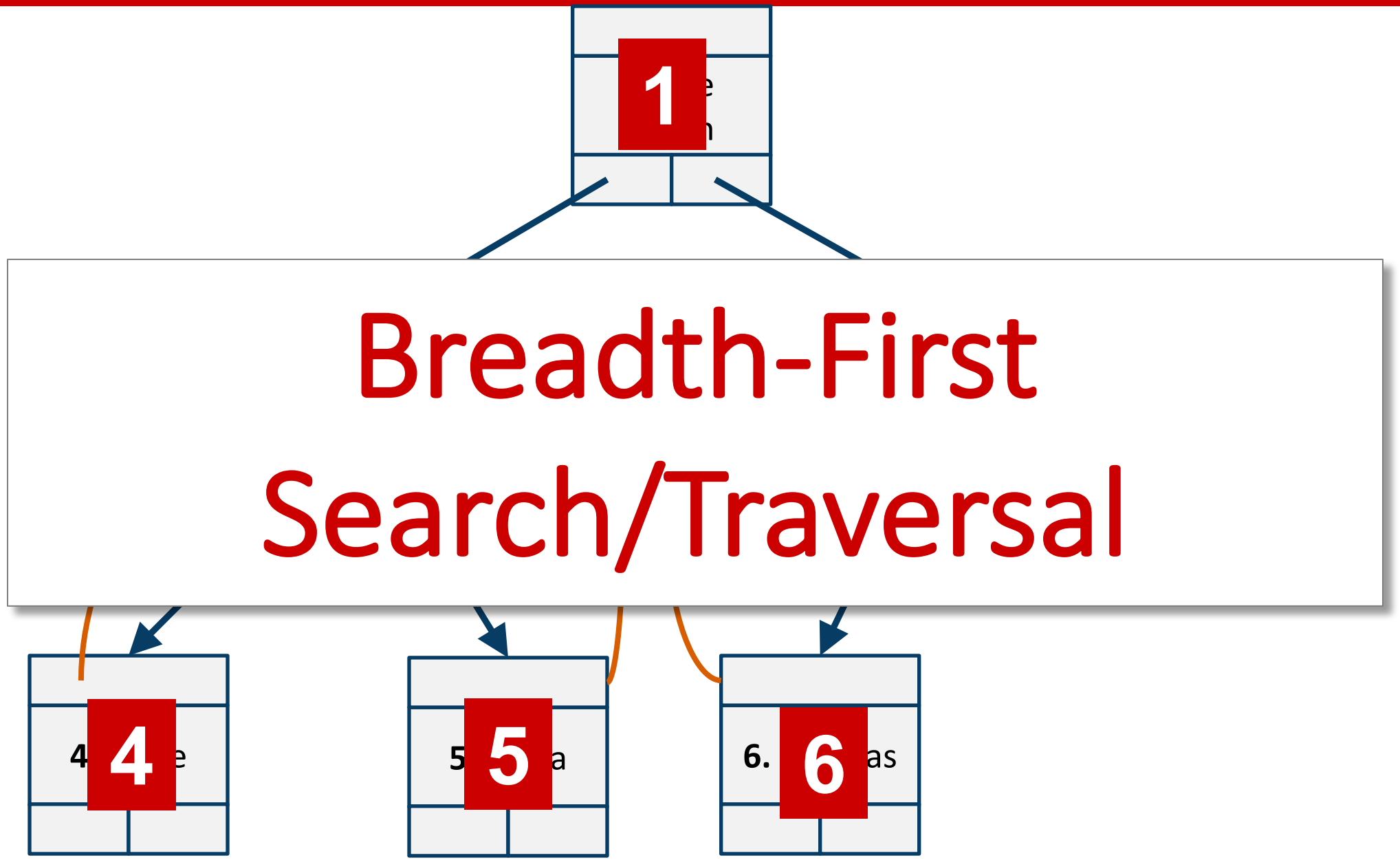


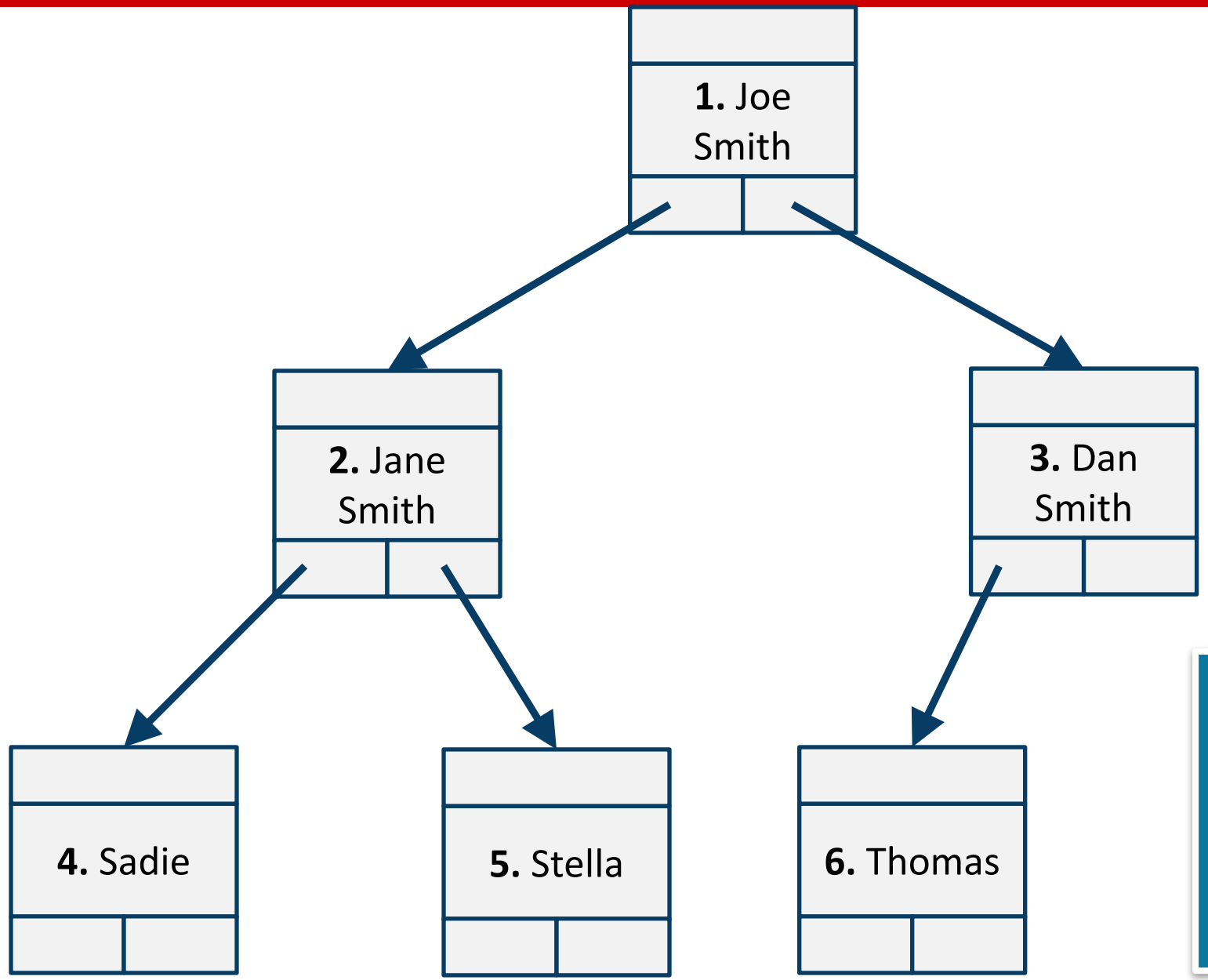
Challenge:

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

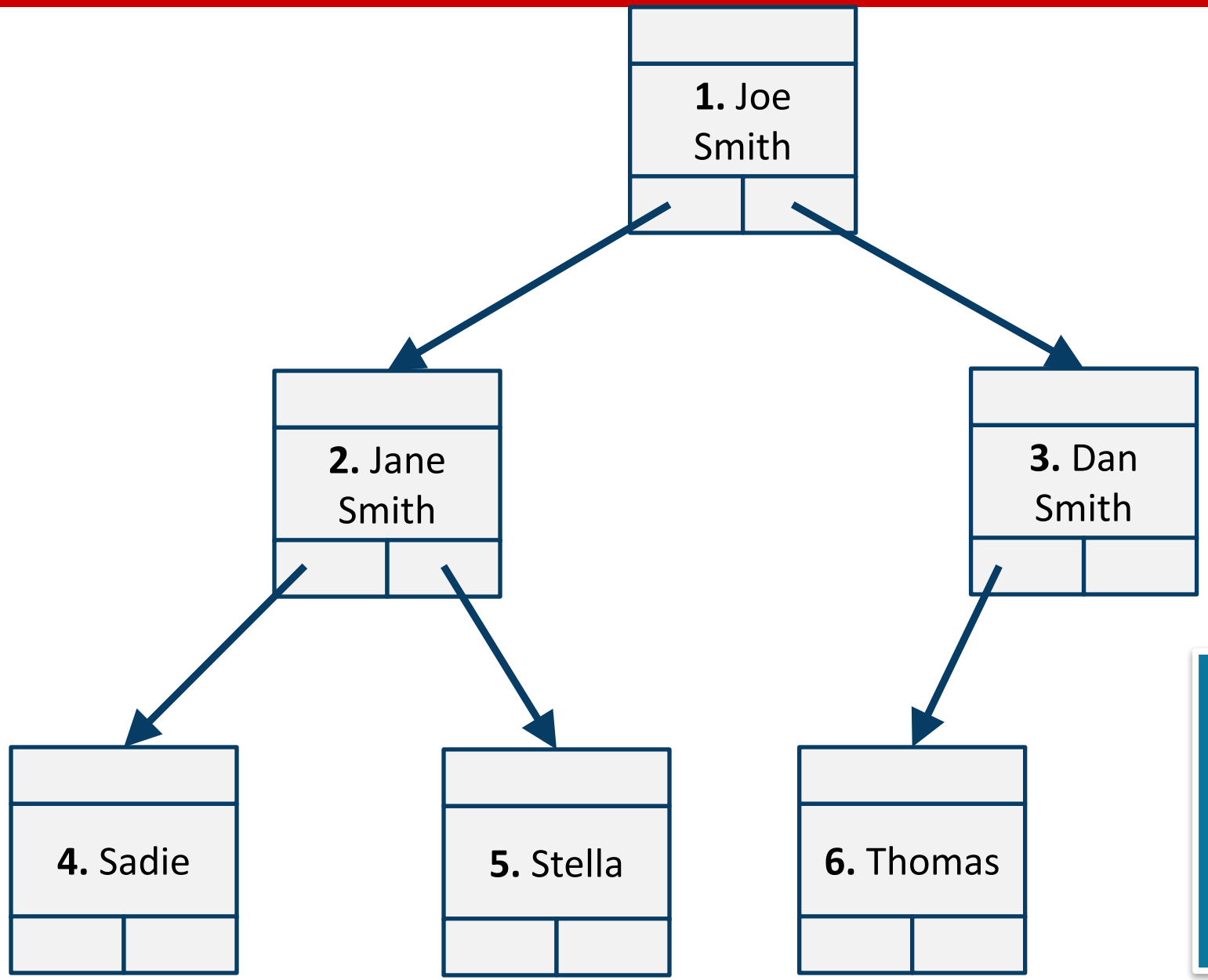
HINT: You might need to add a helper variable somewhere.





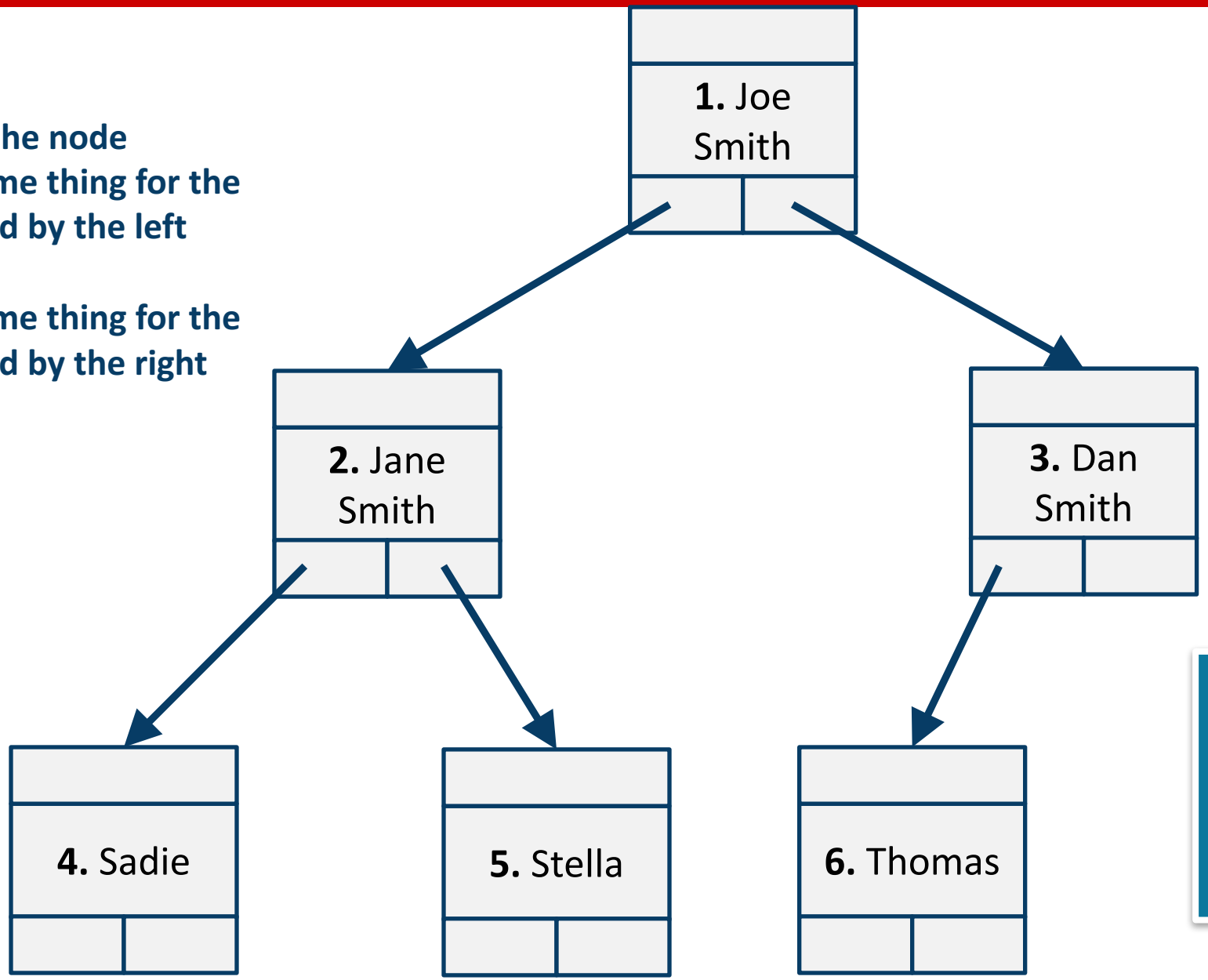


Think about how you would write a non-recursive function to visit each node in a breadth-first manner.



Think about how you would write a non-recursive function to visit each node in a breadth-first manner.

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.



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

- <to put here: More diagrams of BFS>

Implementing Binary Trees in C

```
struct treeNode{
    char* data;
    treeNode* leftChild;
    treeNode* rightChild;
};
```

Operations

- create(), destroy()
- insertNode(), removeNode()
- traverse()

}

Pretty much the same as for a linked list.
If you know the node to insert/remove from, the operation is trivial.
Otherwise, you need to search.
All of that comes down to traversals.

Operations

- create(), destroy()
- insertNode(), removeNode()
- **traverse()**

}

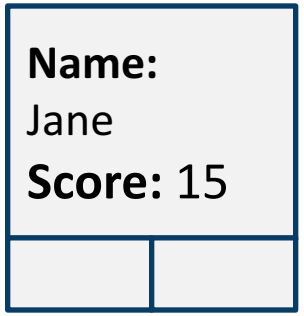
Pretty much the same as for a linked list.
If you know the node to insert/remove from, the operation is trivial.
Otherwise, you need to search.
All of that comes down to traversals.

BINARY SEARCH TREE

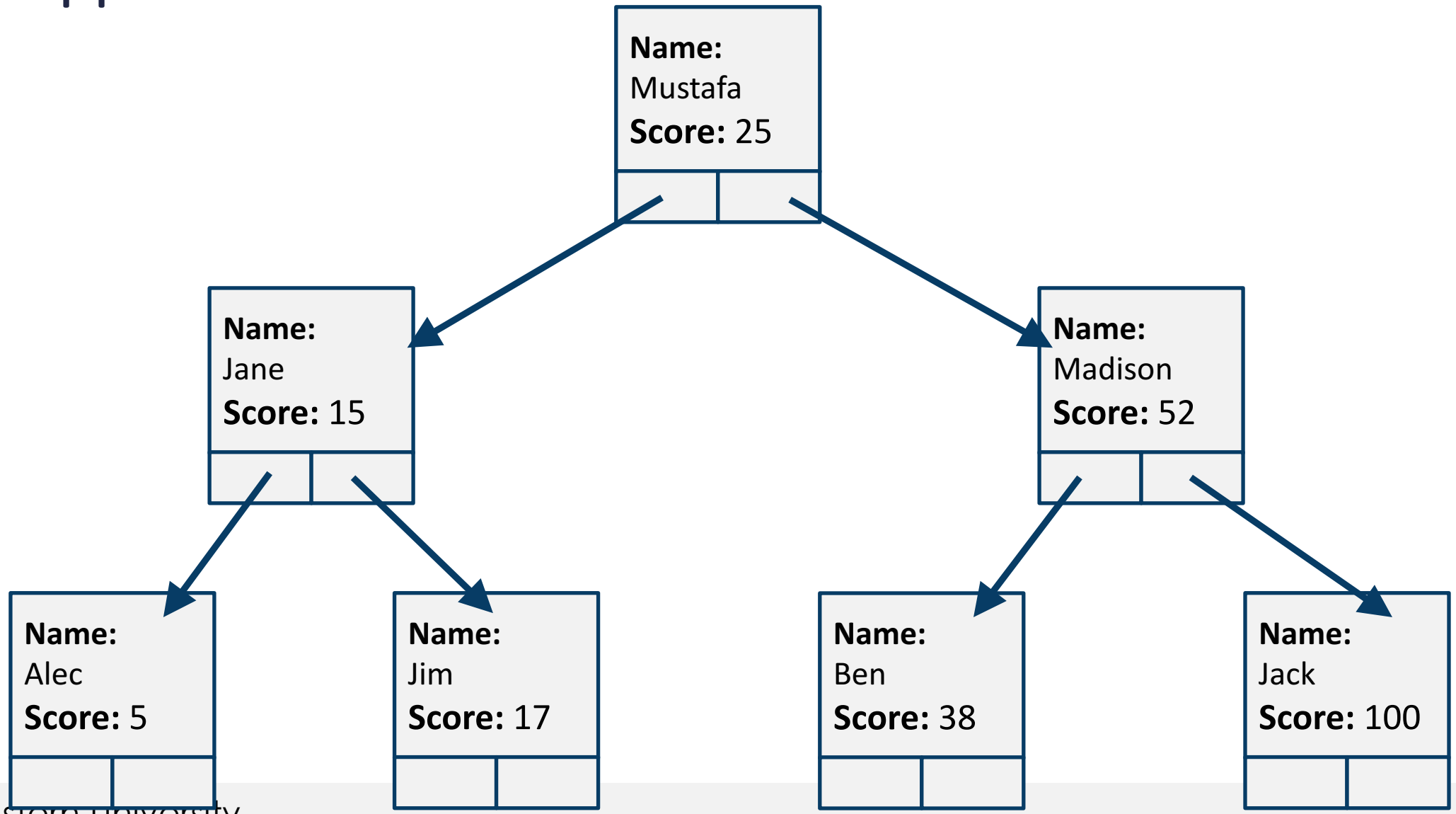
What is special about a BST?

- It's in sorted order.
- 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.
- This makes it easy to search for a value.

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

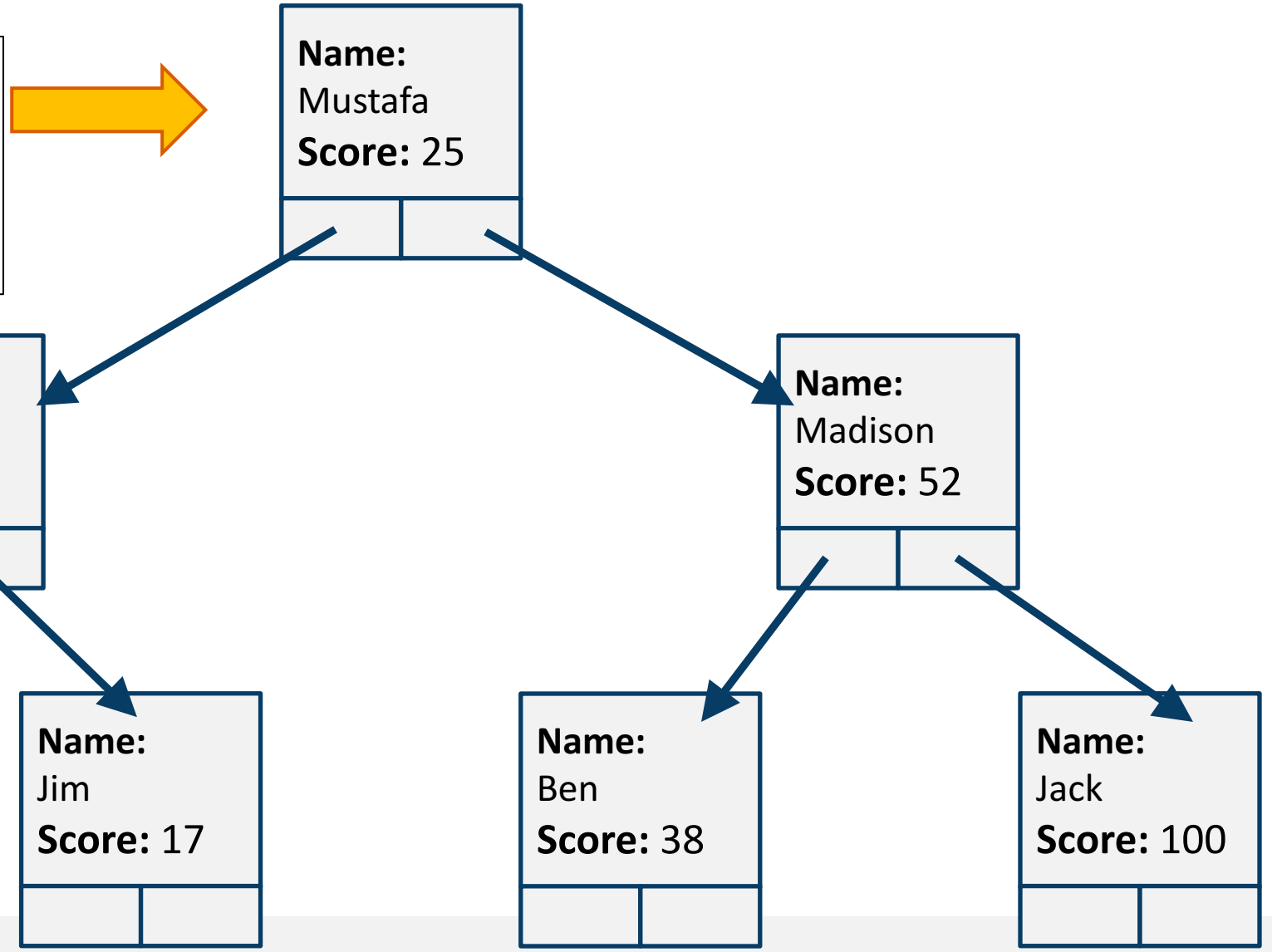


Suppose we want to find who has the score of 15…



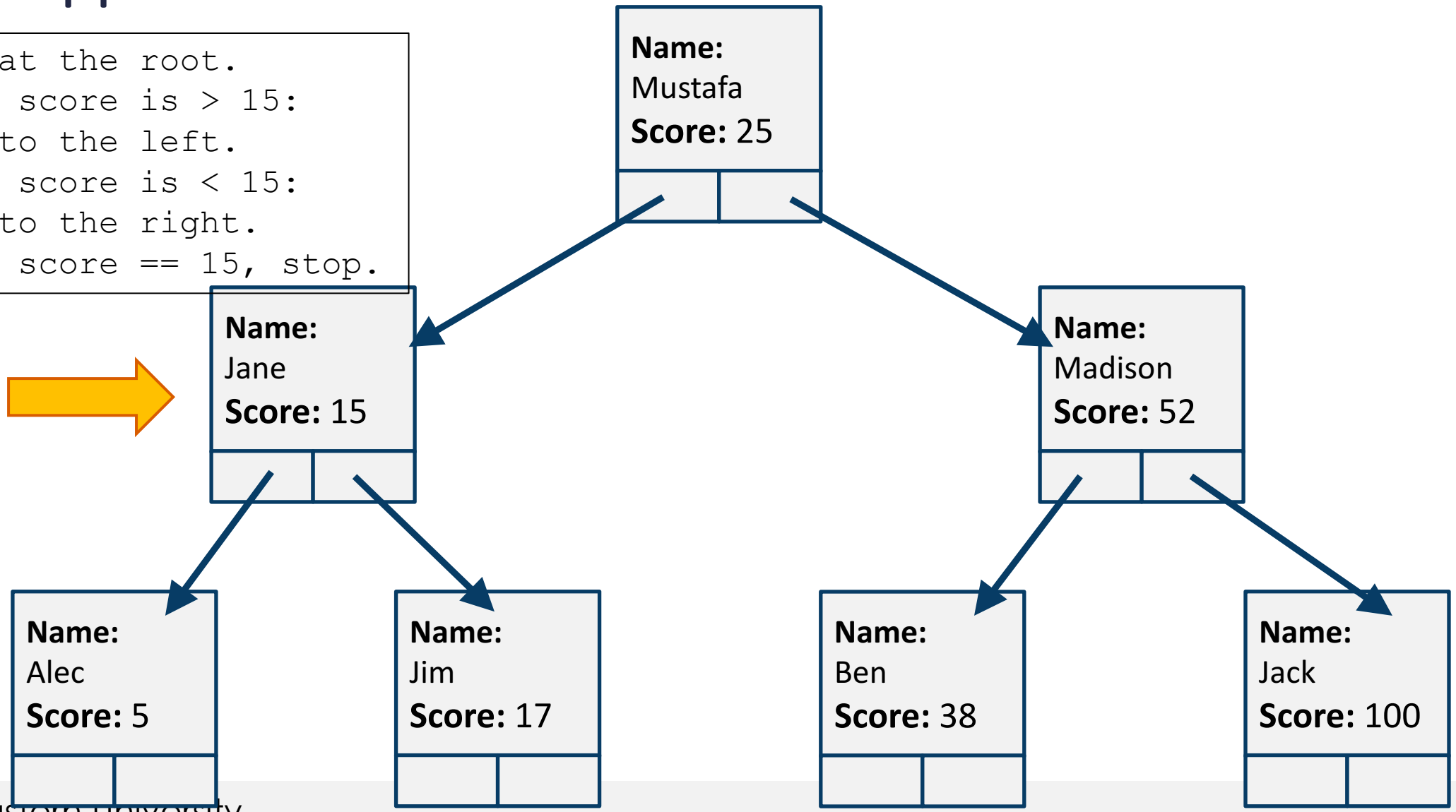
Suppose we want to find who has the score of 15…

```
Start at the root.
If the score is > 15:
   go to the left.
If the score is < 15:
   go to the right.
```



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.
```



Operations for BST:

- **`create(), destroy()`**
- **`insert()`** – this one is not trivial!
- **`remove()`** – again, not trivial!
- **`search()`**: simple

Insert()

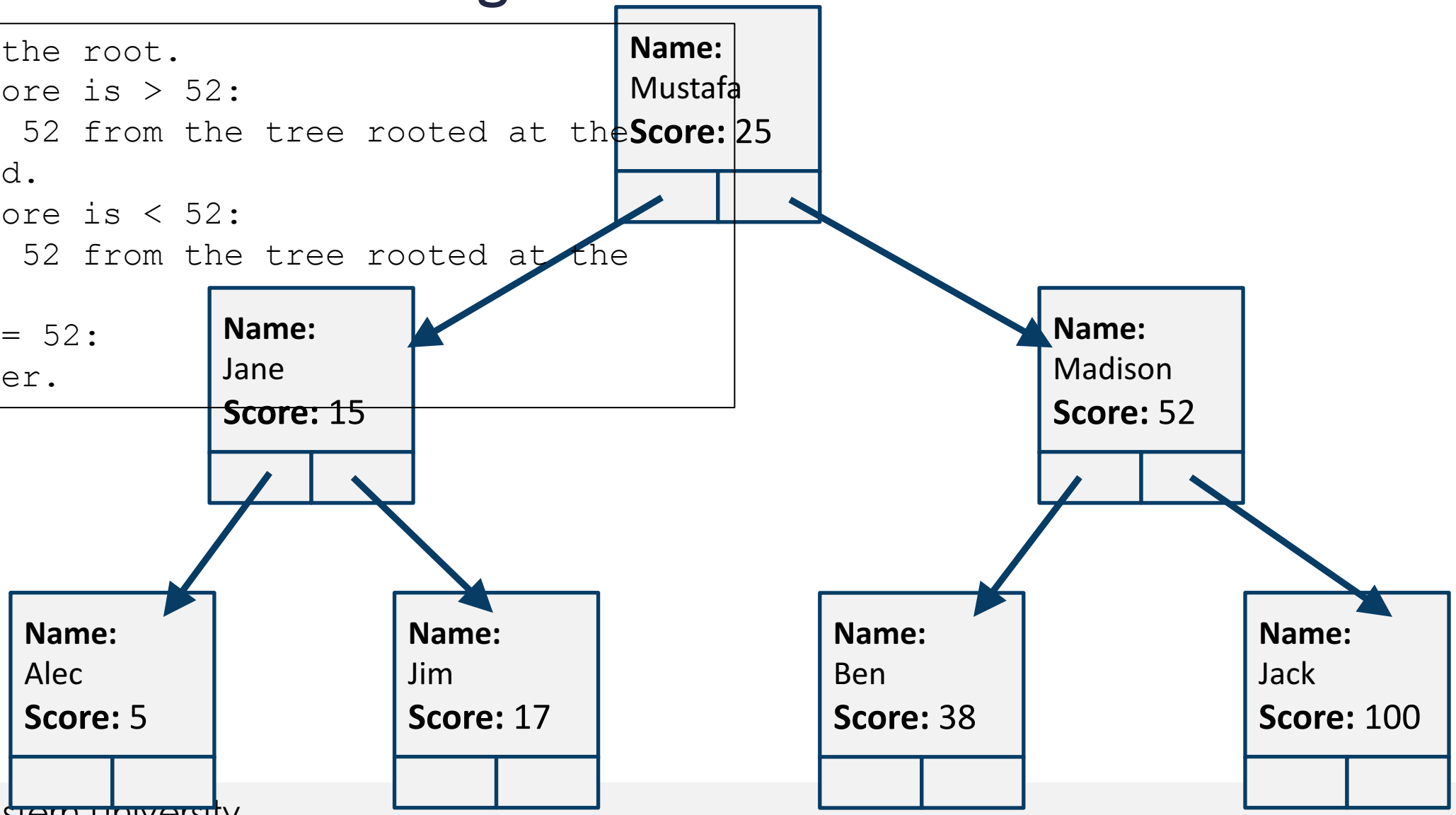
```
Node* insert(Node* root, int key, int value) {
      if (!root)
            root = new Node(key, value);
      else if (key < root->key)
            root->left = insert(root->left, key, value);
      else
            // key >= root->key
            root->right = insert(root->right, key, value);
      return root;
}
```

Remove()

- Deleting a node with no children:
  - simply remove the node from the tree.
- Deleting a node with one child:
  - remove the node and replace it with its child.
- Deleting a node with two children:
  - Call the node to be deleted *D*. Do not delete *D*. Instead, choose either its in-order predecessor node or its in-order successor node as replacement node *E.*
  - Copy the user values of *E* to *D*.
  - If *E* does not have a child simply remove *E* from its previous parent *G*. If *E* has a child, say *F*, it is a right child. Replace *E* with *F* at *E*'s parent.

Deleting node with score 52…

```
Start at the root.
If the score is > 52:
   Delete 52 from the tree rooted at the
left child.
If the score is < 52:
   Delete 52 from the tree rooted at the
right.
If score = 52:
  be clever.
```



Remove()

```
38void BinaryTreeDelete(Node* node, int key) {
39  if (key < node->key) {
40    BinaryTreeDelete(node->left_child, key);
41    return;
42  } else  {
43    if (key > node->key) {
44      BinaryTreeDelete(node->right_child, key);
45      return;
46    }
47  }
48  // We have found the node with the key value
49  if (node->left_child && node->right_child) {
50    // There are two children
51    Node* successor = FindMin(node->right_child);
52    node->key = successor->key;
53    BinaryTreeDelete(successor, successor->key);
54  } else {
55    if (node->left_child) {
56      // The node has only a left child
57      // Replace this node with the value of the left child; delete the left child
58      ReplaceNodeInParent(node, node->left_child);
59    } else {
60      if (node->right_child) {
61        // The node has only a right child
62        // Replace this node with the value of the right child; delete the right child
63        ReplaceNodeInParent(node, node->right_child);
64      } else {
65        // node has no children
66        // Just delete this node
67        DeleteNode(node);
68      }
69    }
70
71  }
72}
```

BINARY HEAP/ PRIORITY QUEUE

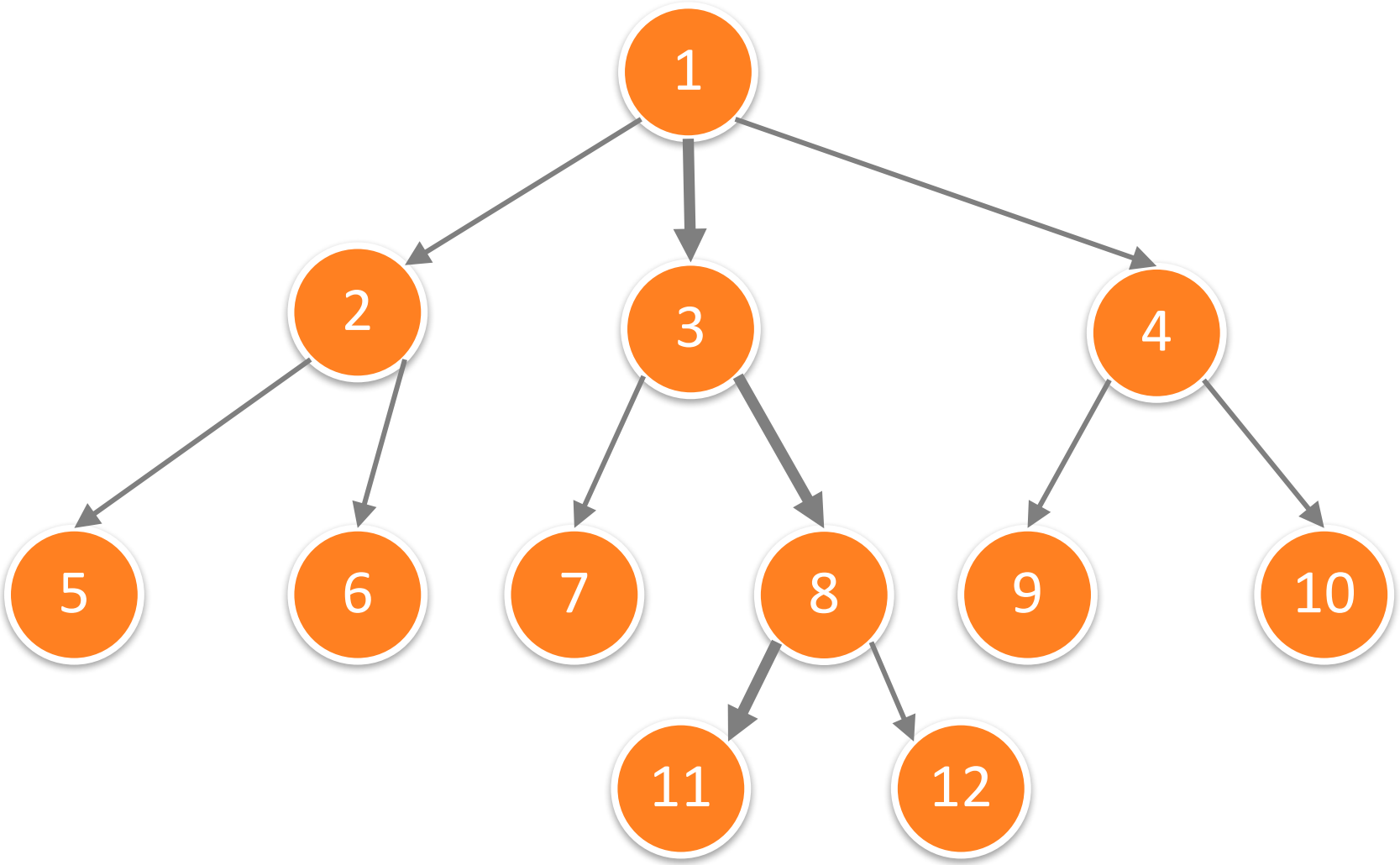
What's special about a priority queue?

- **Priority queues** is an Abstract Data Type (ADT) where elements are dequeued based on priority.
- Each element has a **priority**, an element of a totally ordered set (usually a number)
- More important things come out *first*, even if they were added later.
- Useful for event-based simulators (with priority = simulated time), real-time games, searching, routing, compression via Huffman coding

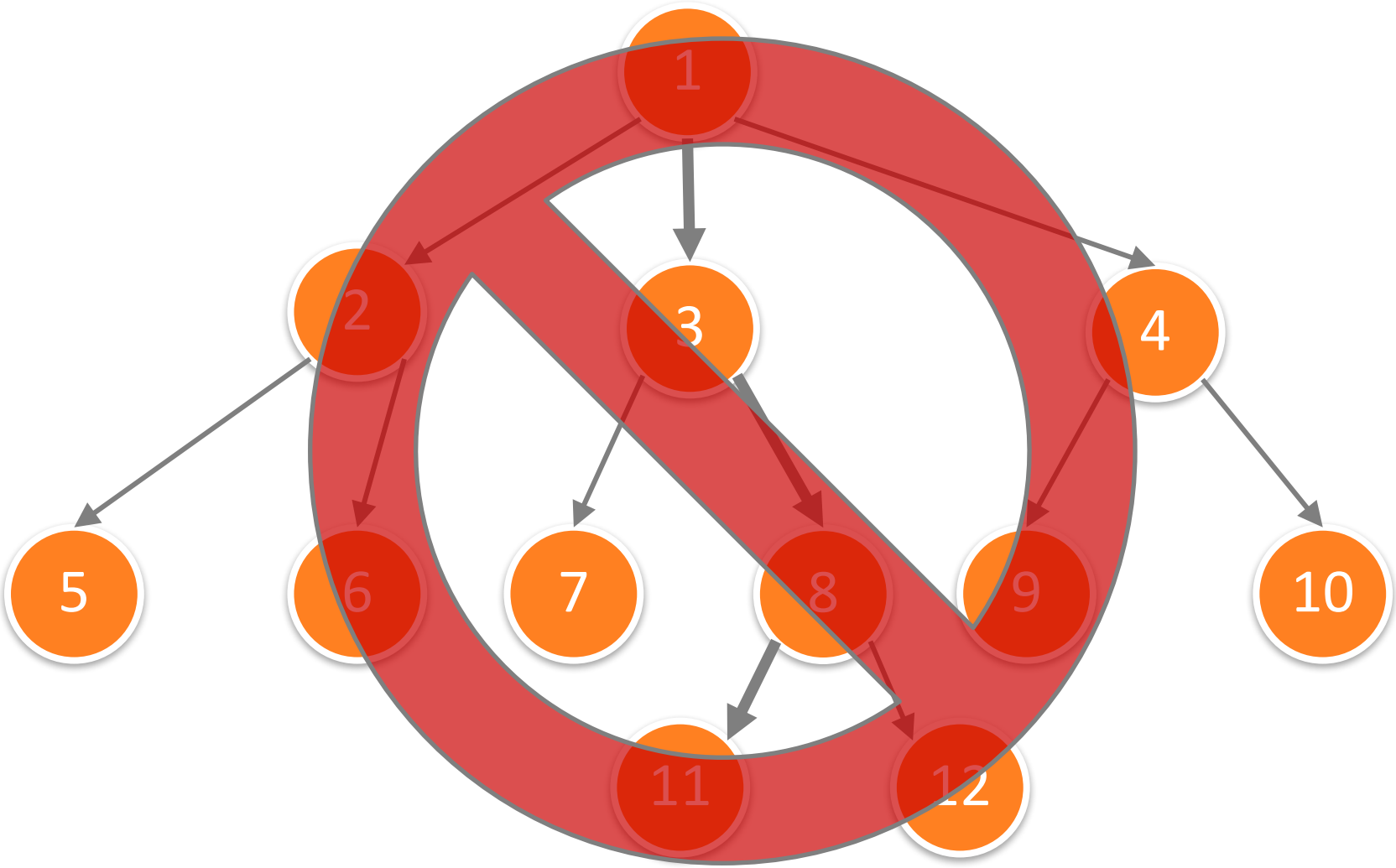
What's special about a binary heap?

- A **binary heap** is often referred to as a **heap**
- It's a special kind of **balanced** binary tree:
  - The priorities of the children of a node are at least as large as the priority of the parent.
    - This implies that the node at the top (root) of the tree has highest priority.
  - The different paths from root to leaf differ in height by at most one.
    - At the bottom of the tree there may be some missing leaves
    - These are to the right to all of the leaves that are present.

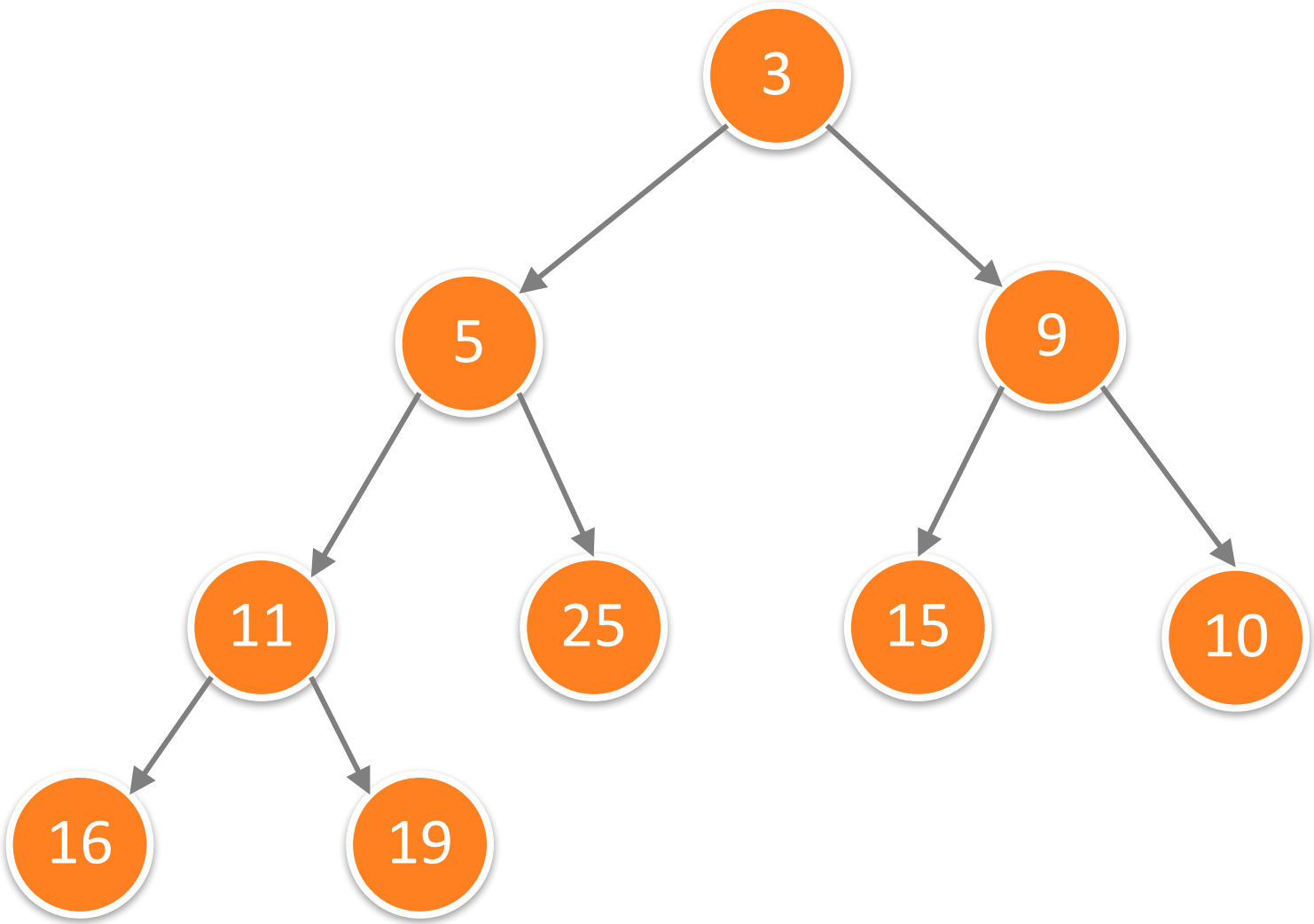
Is this a heap?



Is this a heap?

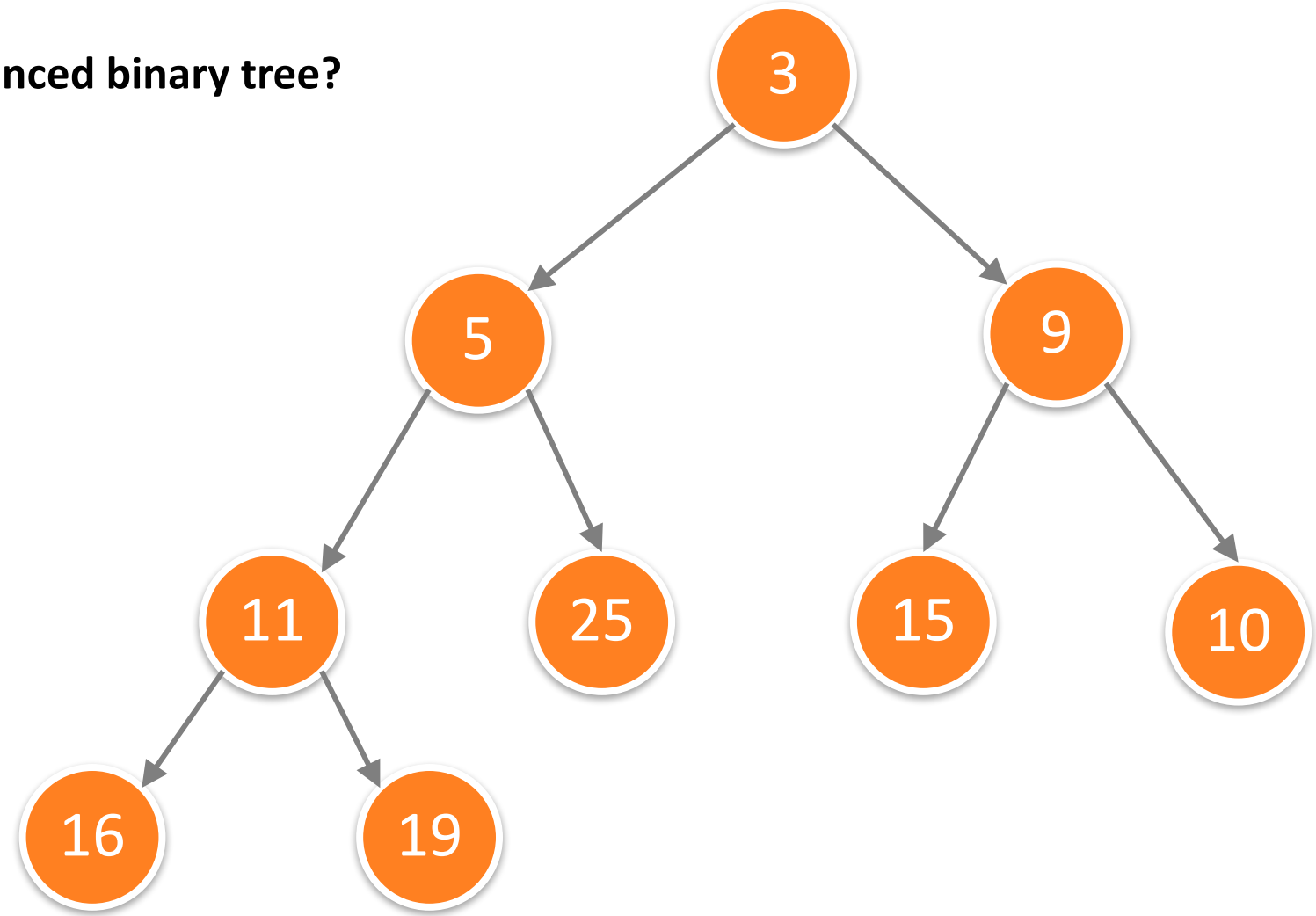


Is this a heap?



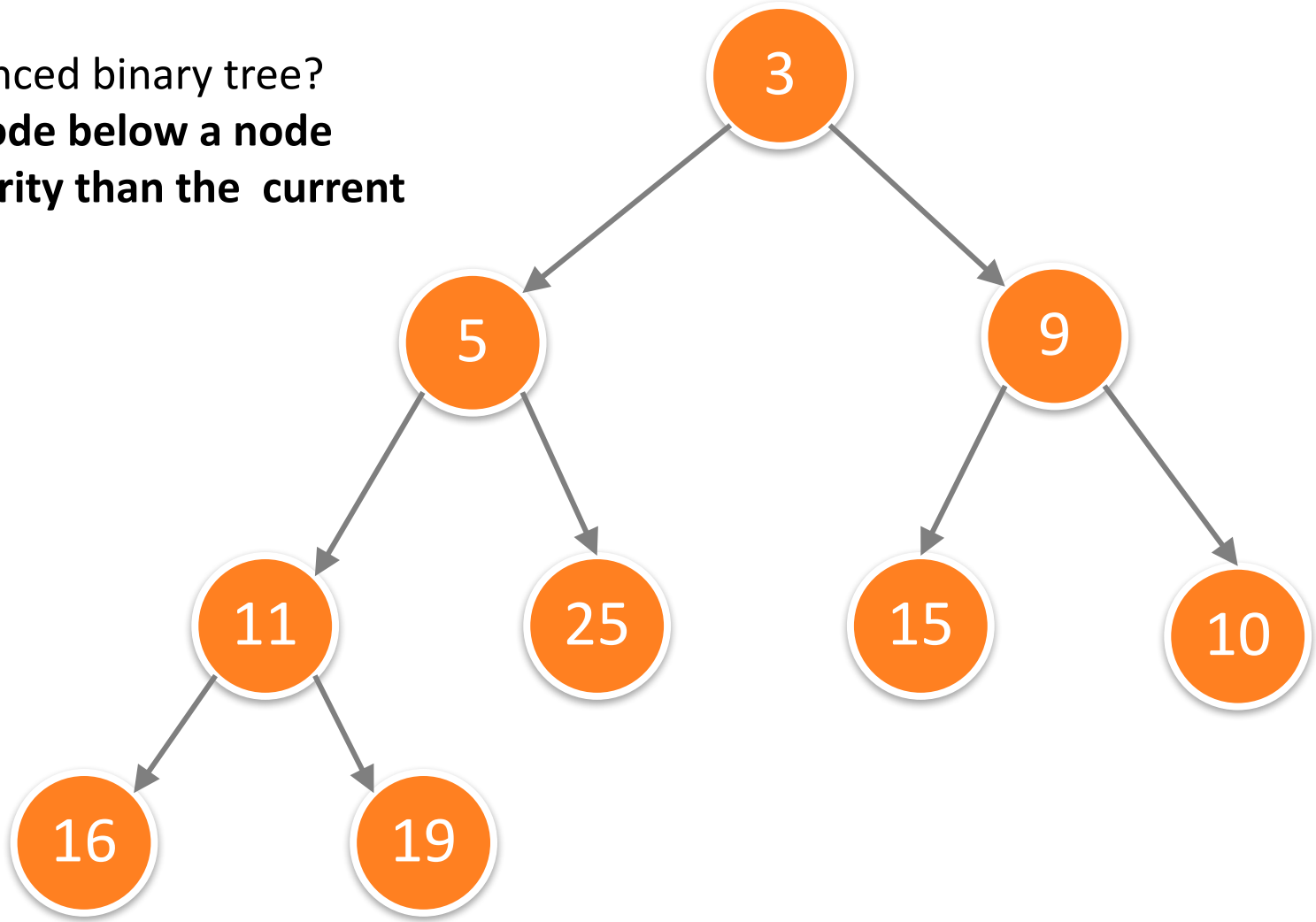
Is this a heap?

Is it a balanced binary tree?



Is this a heap?

Is it a balanced binary tree?
Is every node below a node lower priority than the current node?

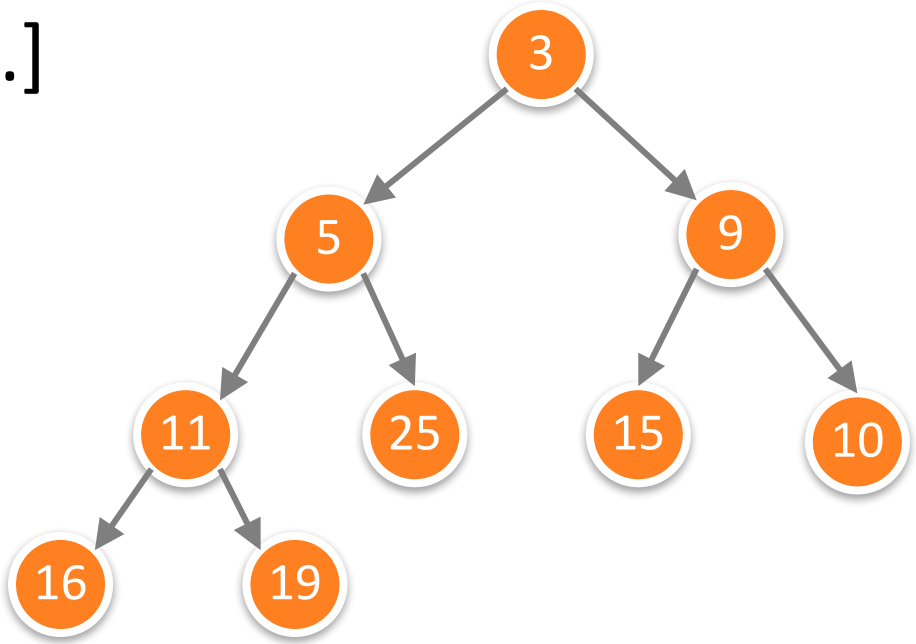


Is this a heap?

Is it a balanced binary tree?
Is every node below a node lower priority than the current node?

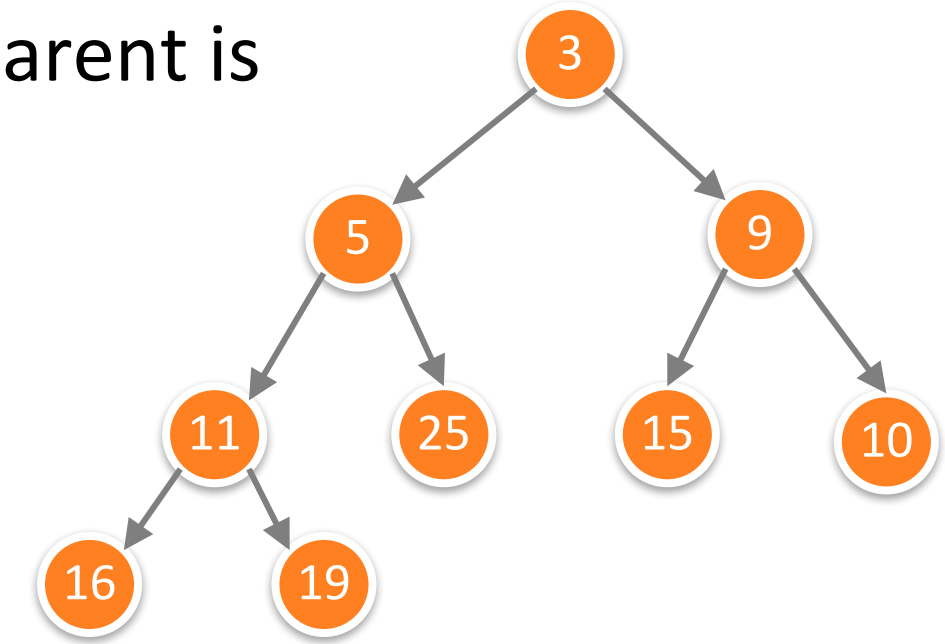
What's the C implementation?

- Usually, an array (rather than nodes as we've been using):
- [3, 5, 9, 11, 25, 15, 10, 16, 19, …]



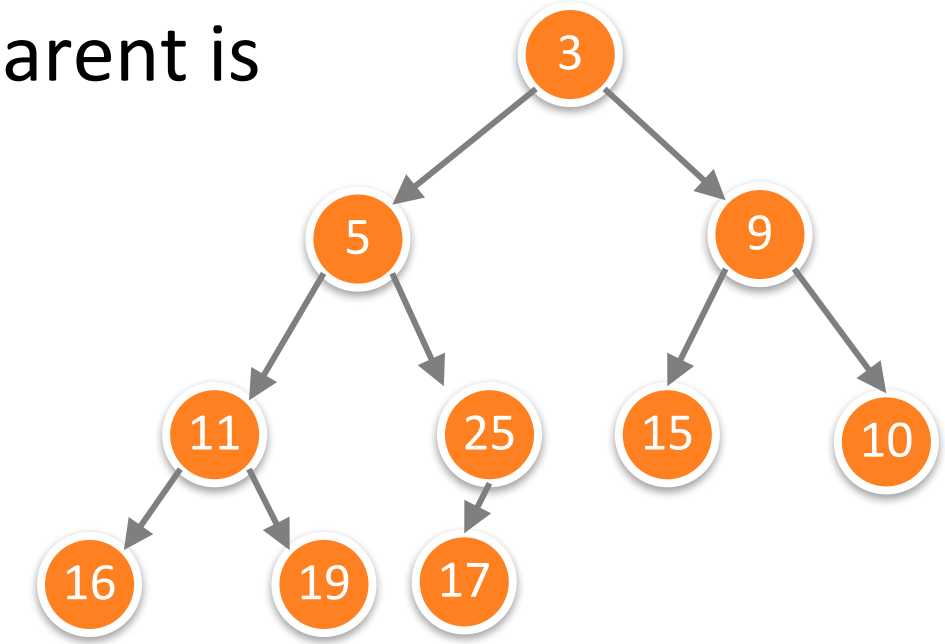
insert()

- Put the element at first missing leaf:
  [3, 5, 9, 11, 25, 15, 10, 16, 19, **17**,…]
- Switch it with its parent if its parent is larger: "**bubble up**"
- Repeat #2 as necessary



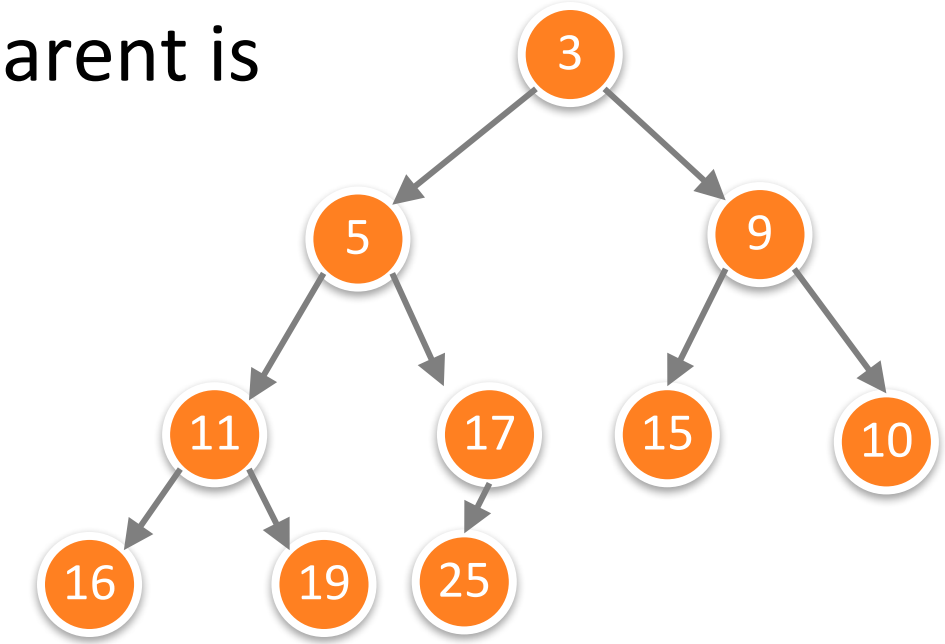
insert(17)

- Put the element at first missing leaf: [3, 5, 9, 11, 25, 15, 10, 16, 19, **17**,…]
- Switch it with its parent if its parent is larger: "**bubble up**"
- Repeat #2 as necessary



insert(17)

- Put the element at first missing leaf: [3, 5, 9, 11, 25, 15, 10, 16, 19, **17**,…]
- Switch it with its parent if its parent is larger: "**bubble up**"
- Repeat #2 as necessary



insert()

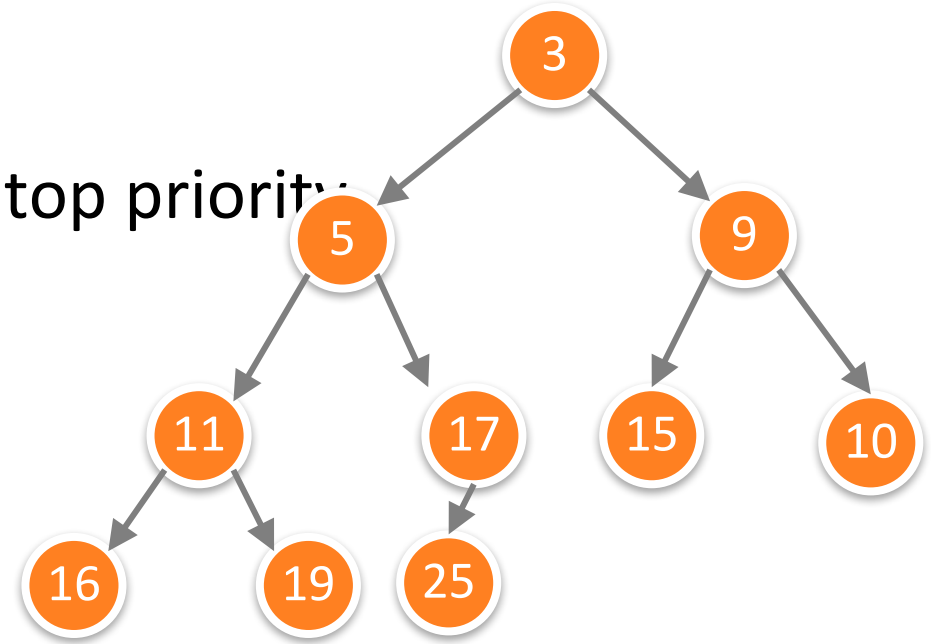
```
void insert(heap *heap, int val){
     heap->vals[heap->nextVal++] = val;
     int new_ind = heap->nextVal;
     int par_ind = (new_ind - 1)/2;

     while (heap->vals[par_ind] > heap->vals[new_ind]){
          // Swap them
          // logic goes here
     }
}
```

get_min()

- Get the next element– the one with the top priority
- Return element at root
  - Removing the element from the tree
  - Guaranteed to be the most important
- Fix the two subtrees:
  - Copy a leaf (last element) to the root (first element)
  - If it's larger than one of the children, bubble it down.
  - Swap with the higher priority child, to make sure the parent is always more important than both children

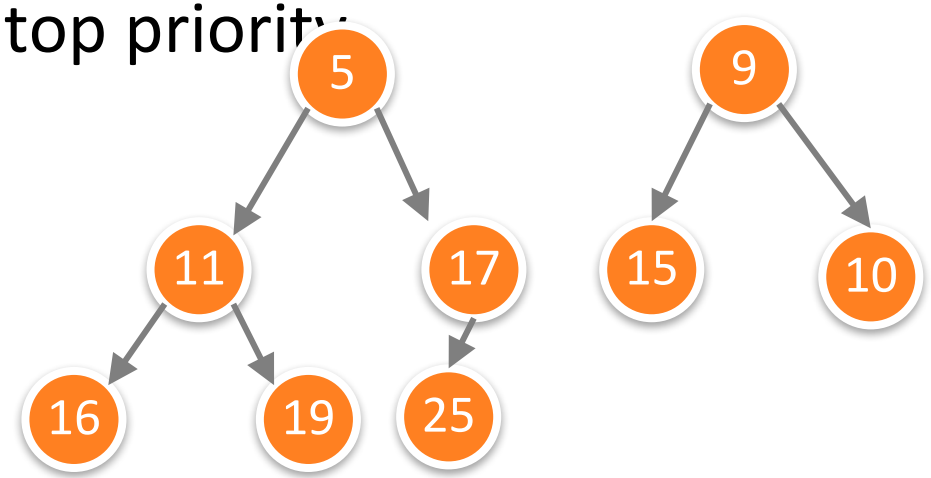
get_min()



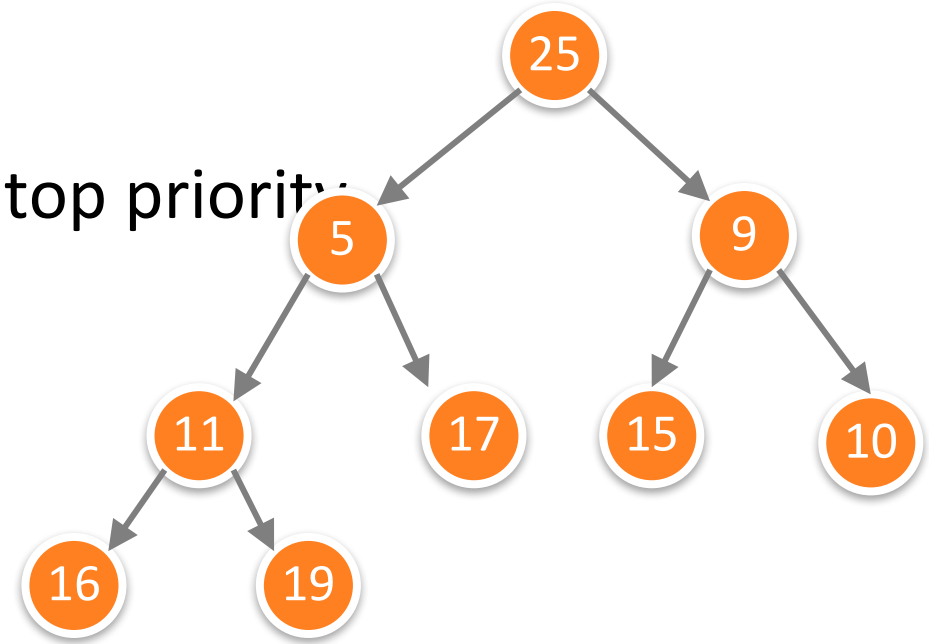
- Get the next element– the one with the top priority
- Return element at root
  - Removing the element from the tree
  - Guaranteed to be the most important
- Fix the two subtrees:
  - Copy a leaf (last element) to the root (first element)
  - If it's larger than one of the children, bubble it down.
  - Swap with the higher priority child, to make sure the parent is always more important than both children

get_min()

- Get the next element– the one with the top priority
- Return element at root
  - Removing the element from the tree
  - Guaranteed to be the most important
- Fix the two subtrees:
  - Copy a leaf (last element) to the root (first element)
  - If it's larger than one of the children, bubble it down.
  - Swap with the higher priority child, to make sure the parent is always more important than both children



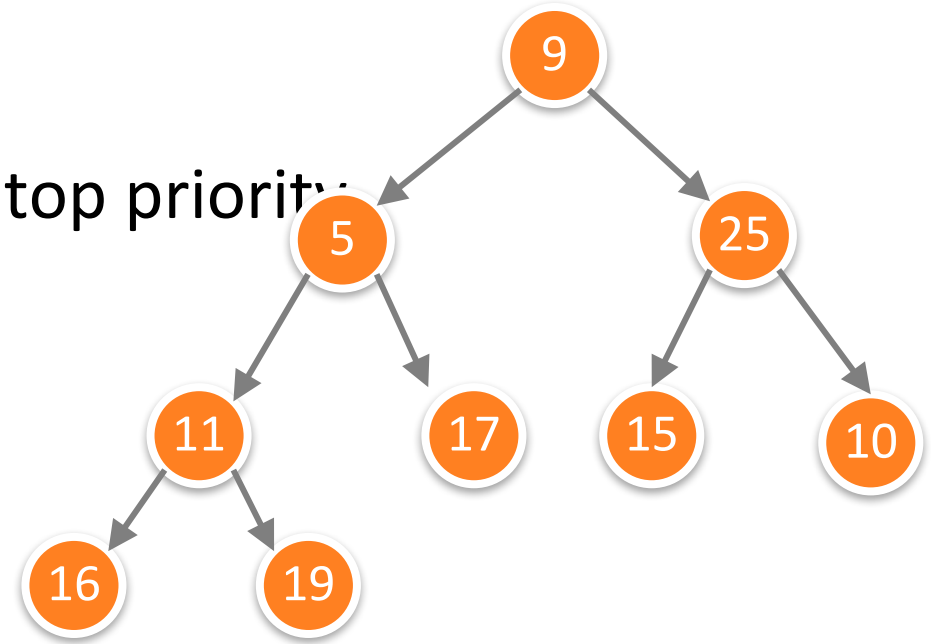
get_min()



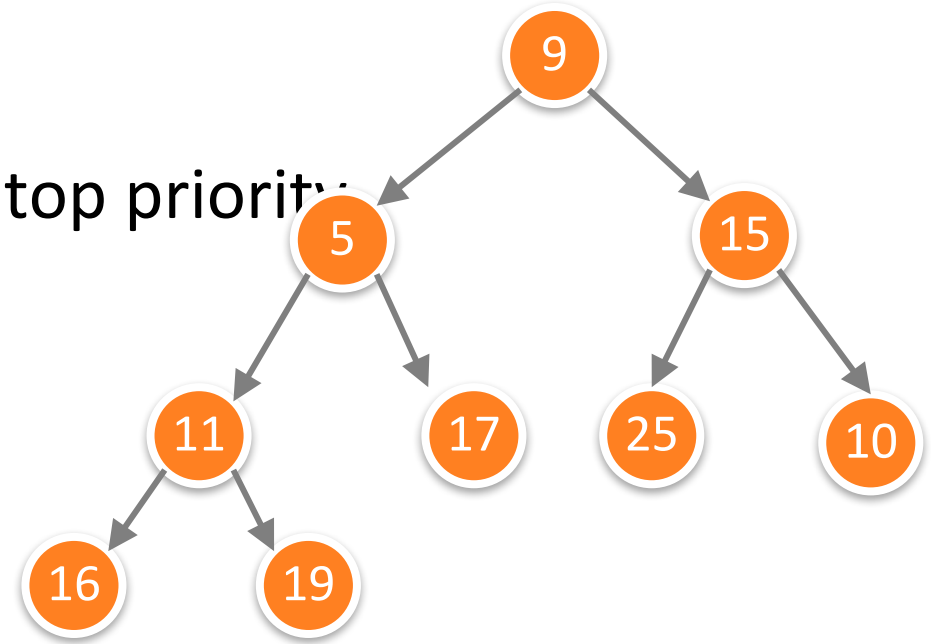
- Get the next element– the one with the top priority
- Return element at root
  - Removing the element from the tree
  - Guaranteed to be the most important
- Fix the two subtrees:
  - **Copy a leaf (last element) to the root (first element)**
  - If it's larger than one of the children, bubble it down.
  - Swap with the higher priority child, to make sure the parent is always more important than both children

get_min()

- Get the next element– the one with the top priority
- Return element at root
  - Removing the element from the tree
  - Guaranteed to be the most important
- Fix the two subtrees:
  - Copy a leaf (last element) to the root (first element)
  - **If it's larger than one of the children, bubble it down.**
    - Swap with the higher priority child, to make sure the parent is always more important than both children



get_min()



- Get the next element– the one with the top priority
- Return element at root
  - Removing the element from the tree
  - Guaranteed to be the most important
- Fix the two subtrees:
  - Copy a leaf (last element) to the root (first element)
  - **If it's larger than one of the children, bubble it down.**
    - Swap with the higher priority child, to make sure the parent is always more important than both children

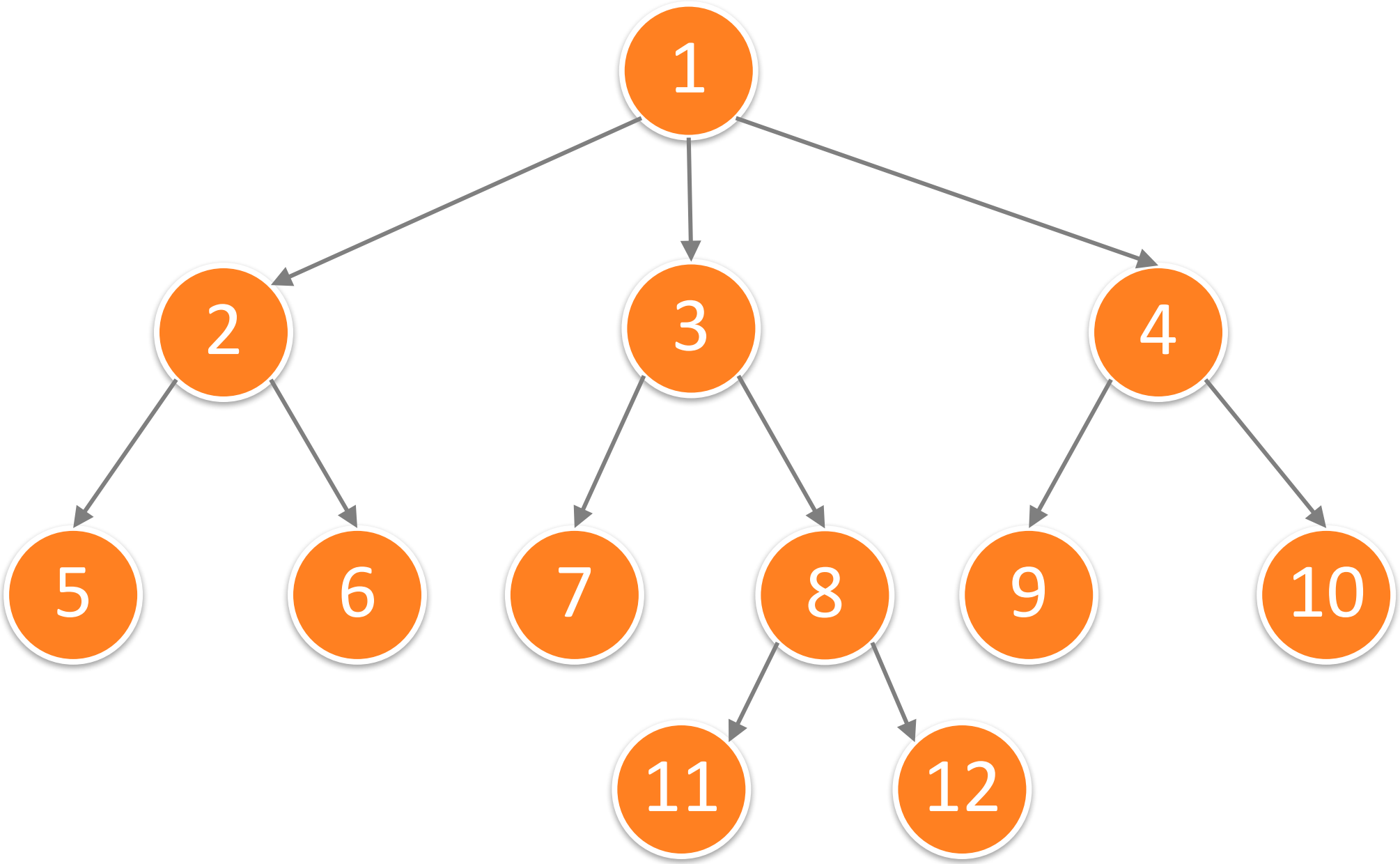
get_min()

```
int get_min(){

}
```

If there's time...

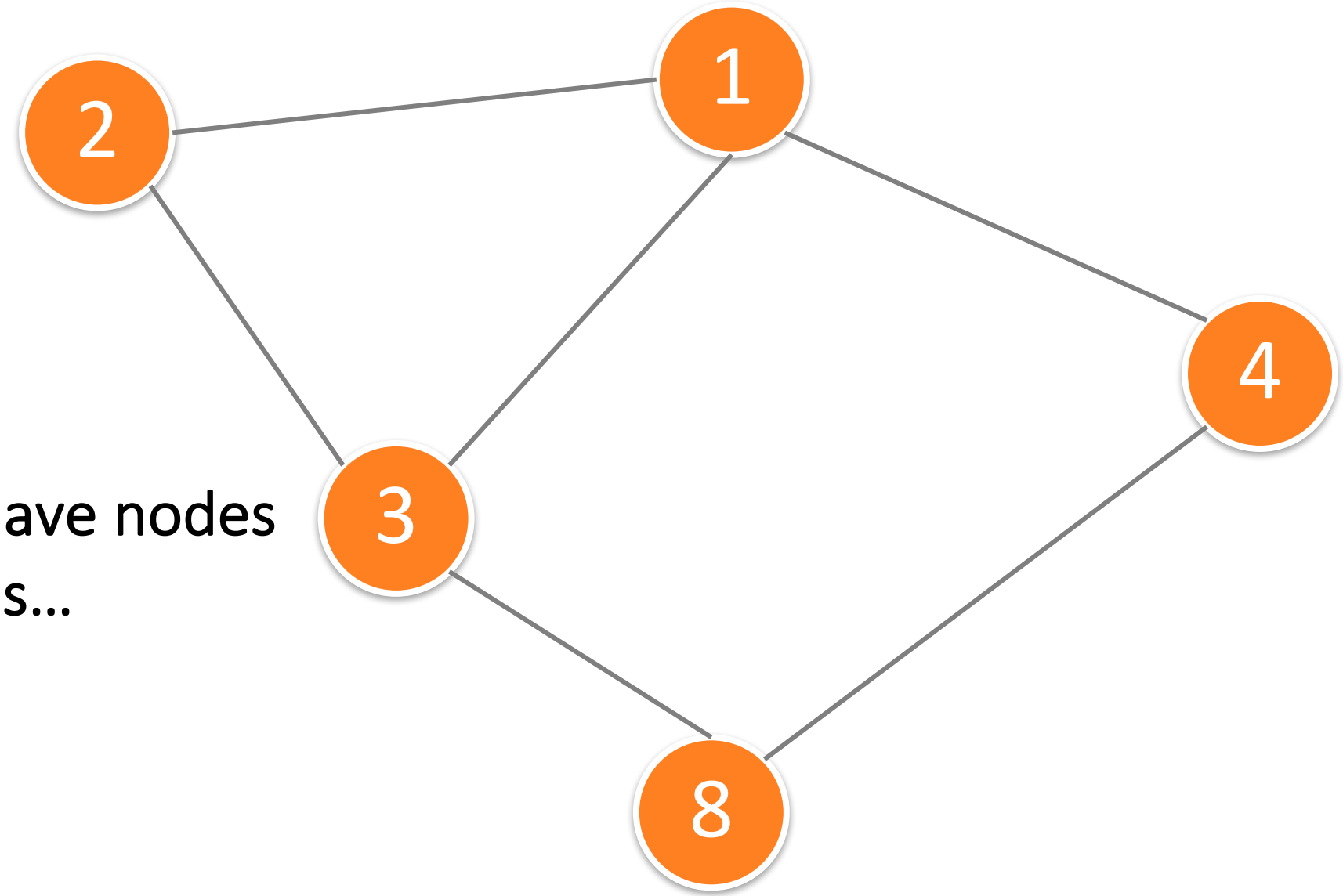
Binary Numbers

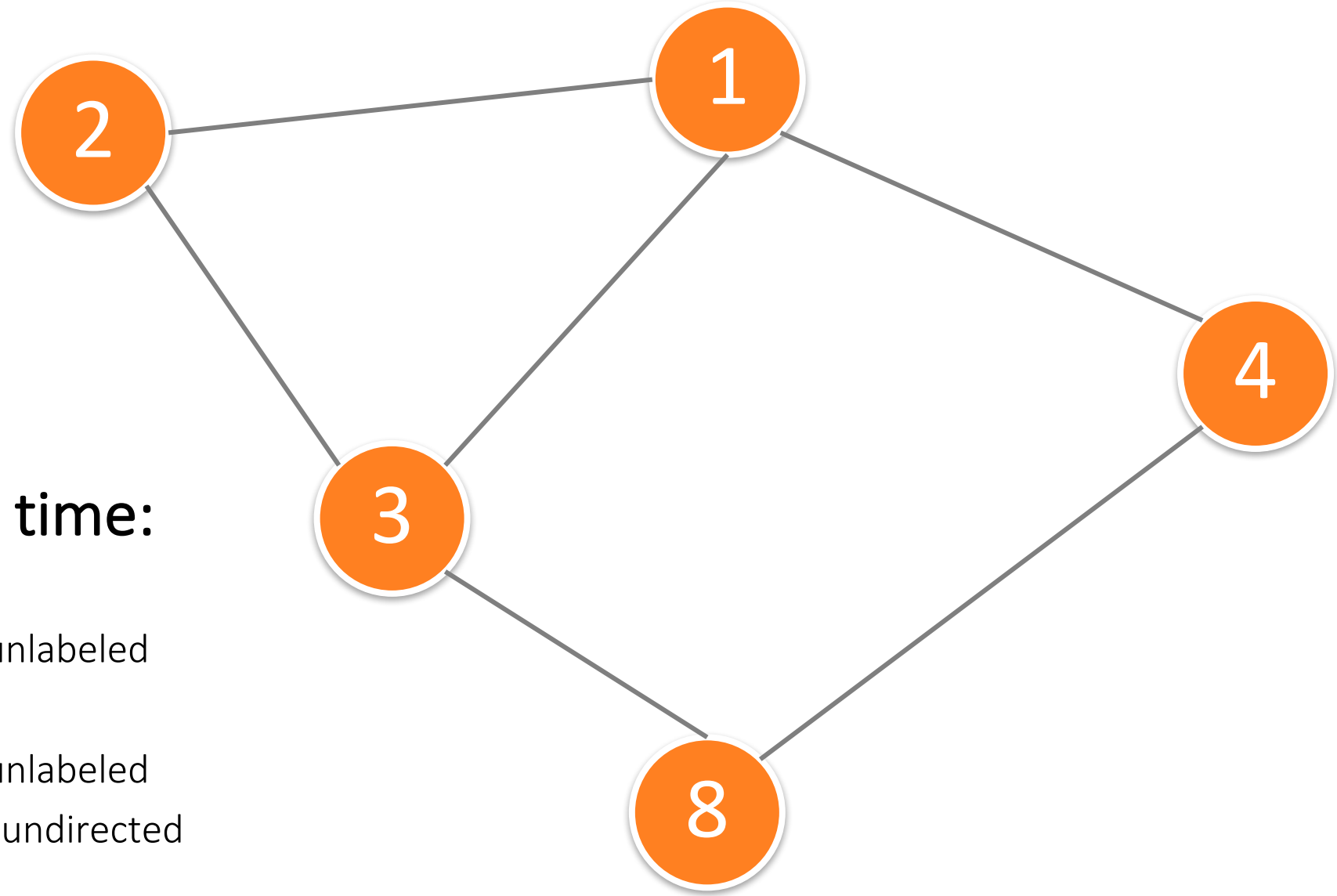


What is a tree?

- A special kind of graph
- Starts at the root
- Each node has at most one parent
  - That is, there are no cycles
- Represents hierarchical data
  - e.g., Organizational structure

We still have nodes and edges…





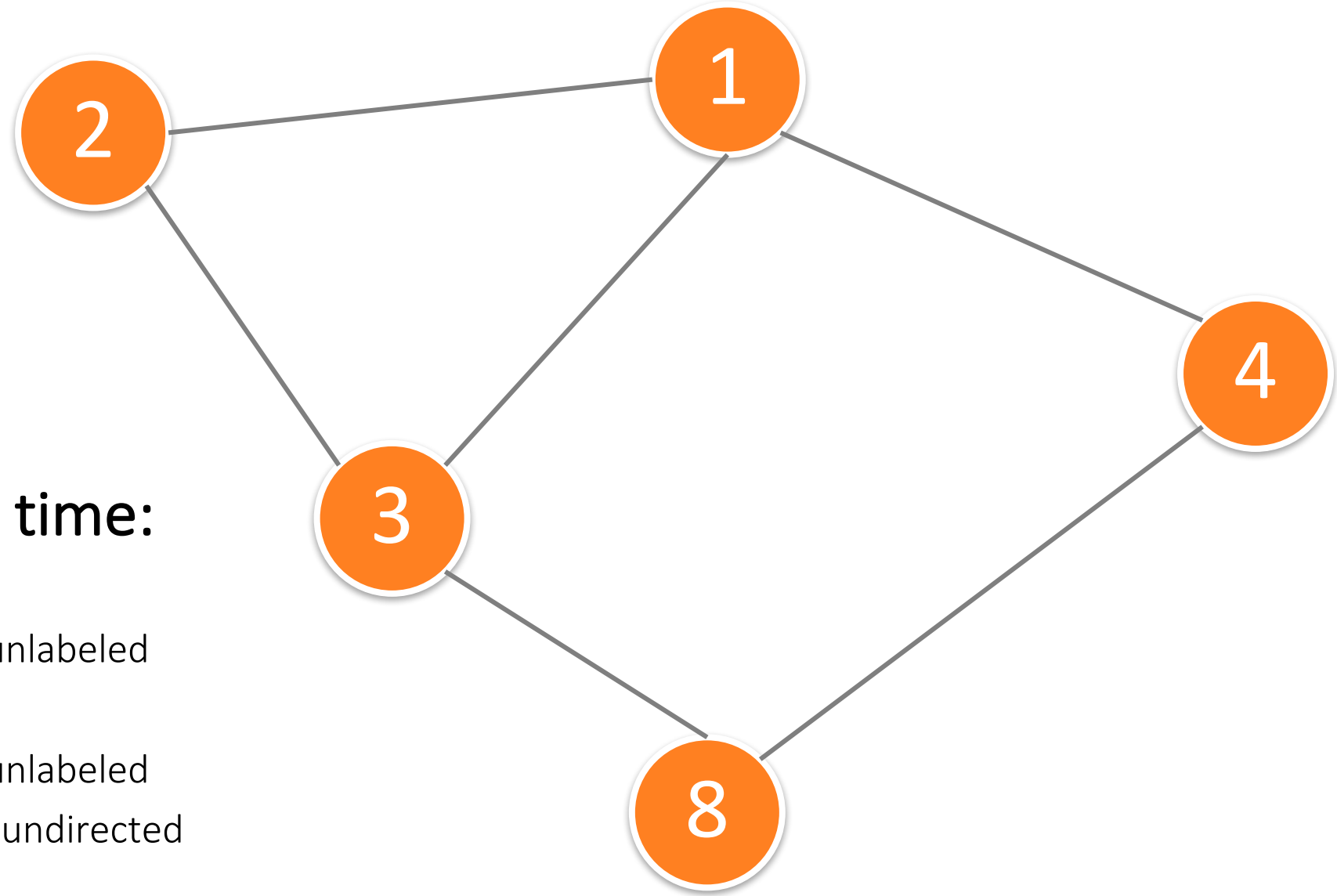
…but this time:

Nodes:

- labeled or unlabeled

Edges:

- labeled or unlabeled
- directed or undirected



…but this time:

Nodes:

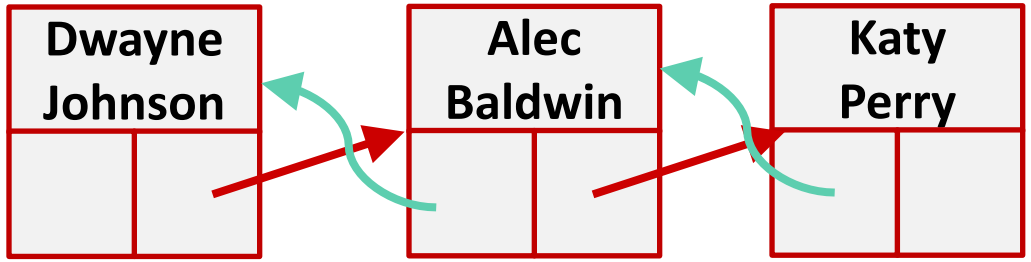
- labeled or unlabeled

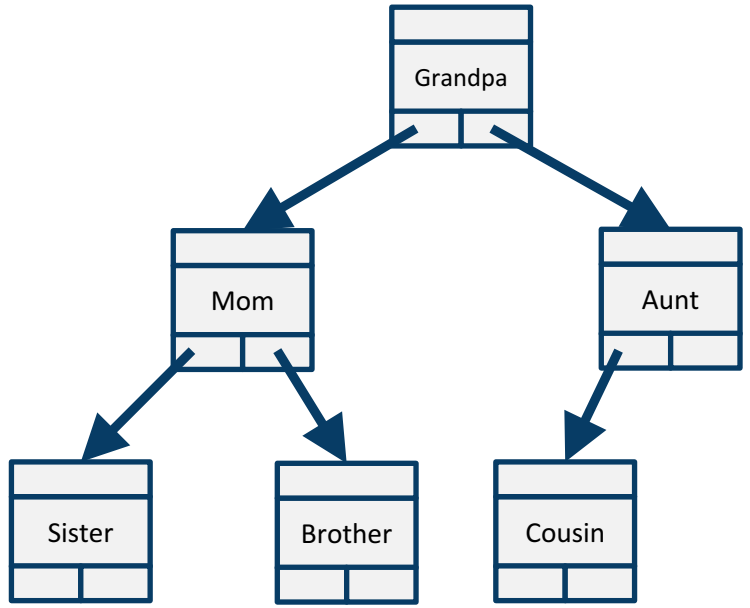
Edges:

- labeled or unlabeled
- directed or undirected

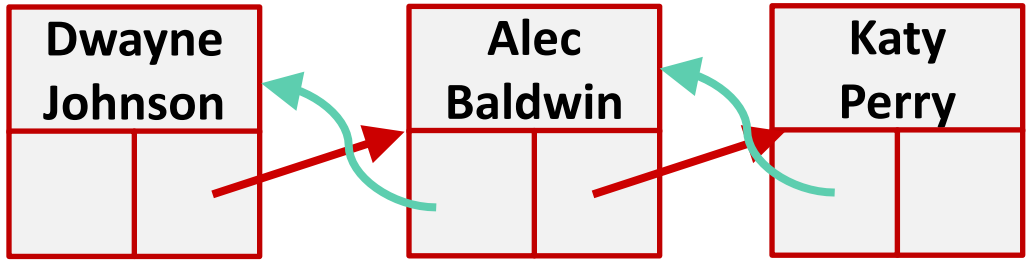
Previous data structures

- Linked List
- Trees
  - Binary Trees
  - Heaps



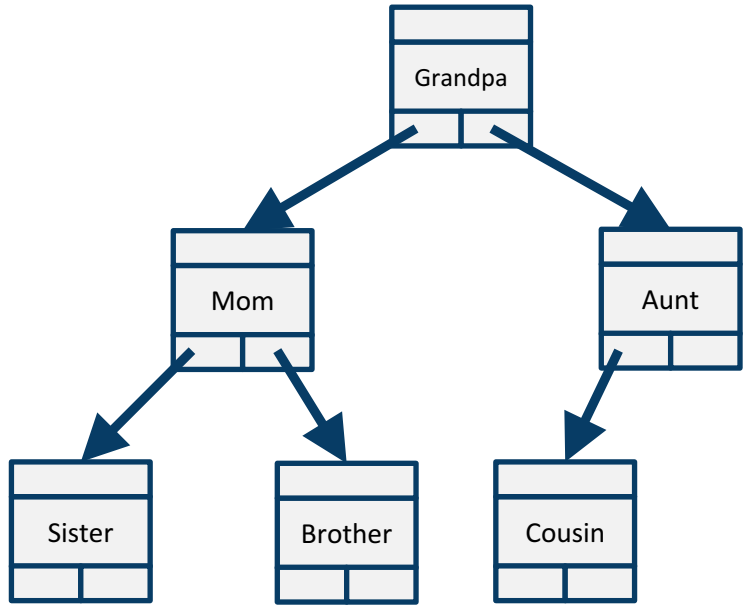


Previous data structures



- Linked List
- Trees
  - Binary Trees
  - Heaps

What do these have in common?



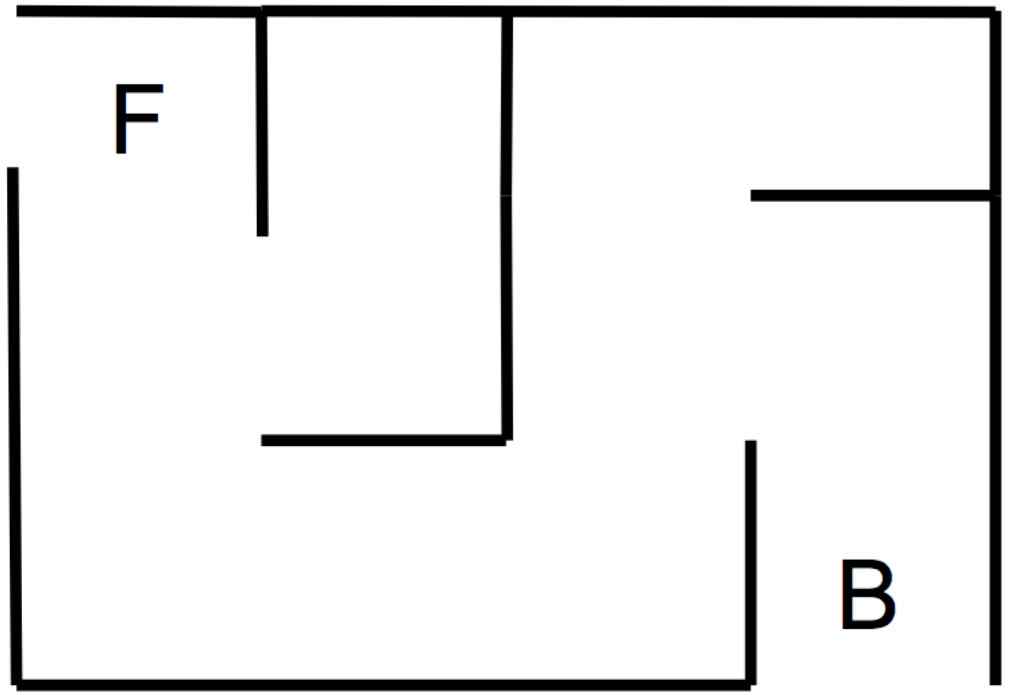
Can we make some generalizations?

How might we model this problem?

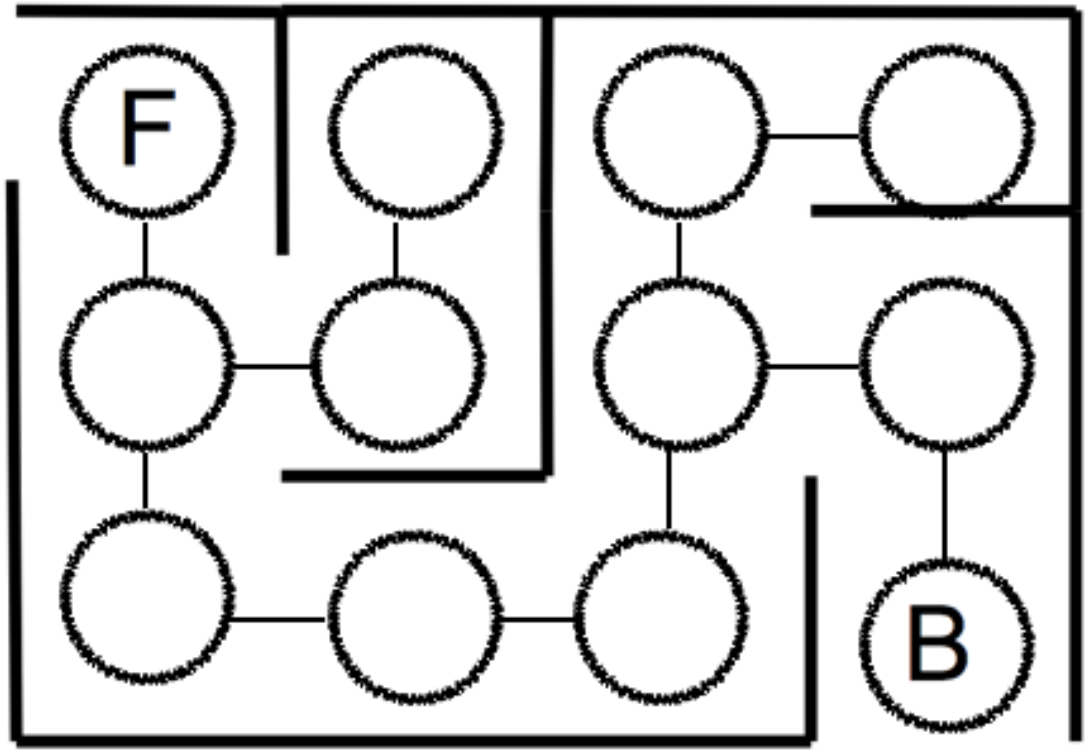
- To meet the requirements for graduation, you must take x number of classes.
- Class Y must be taken before Class Z.
- Class A and B must be taken before everything else.
- …

- <graph goes here>
- What do the nodes represent?
- What do the edges represent?

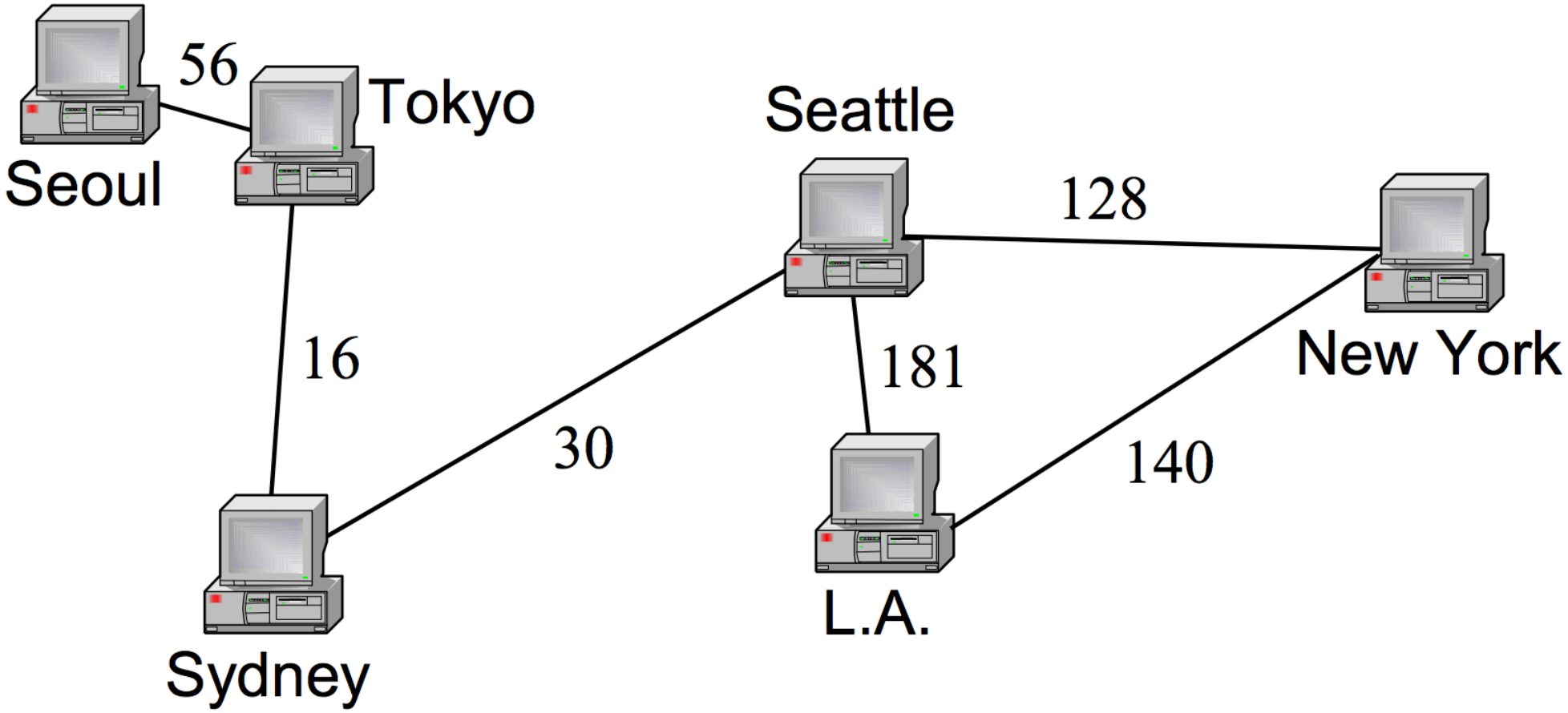
How about this problem?



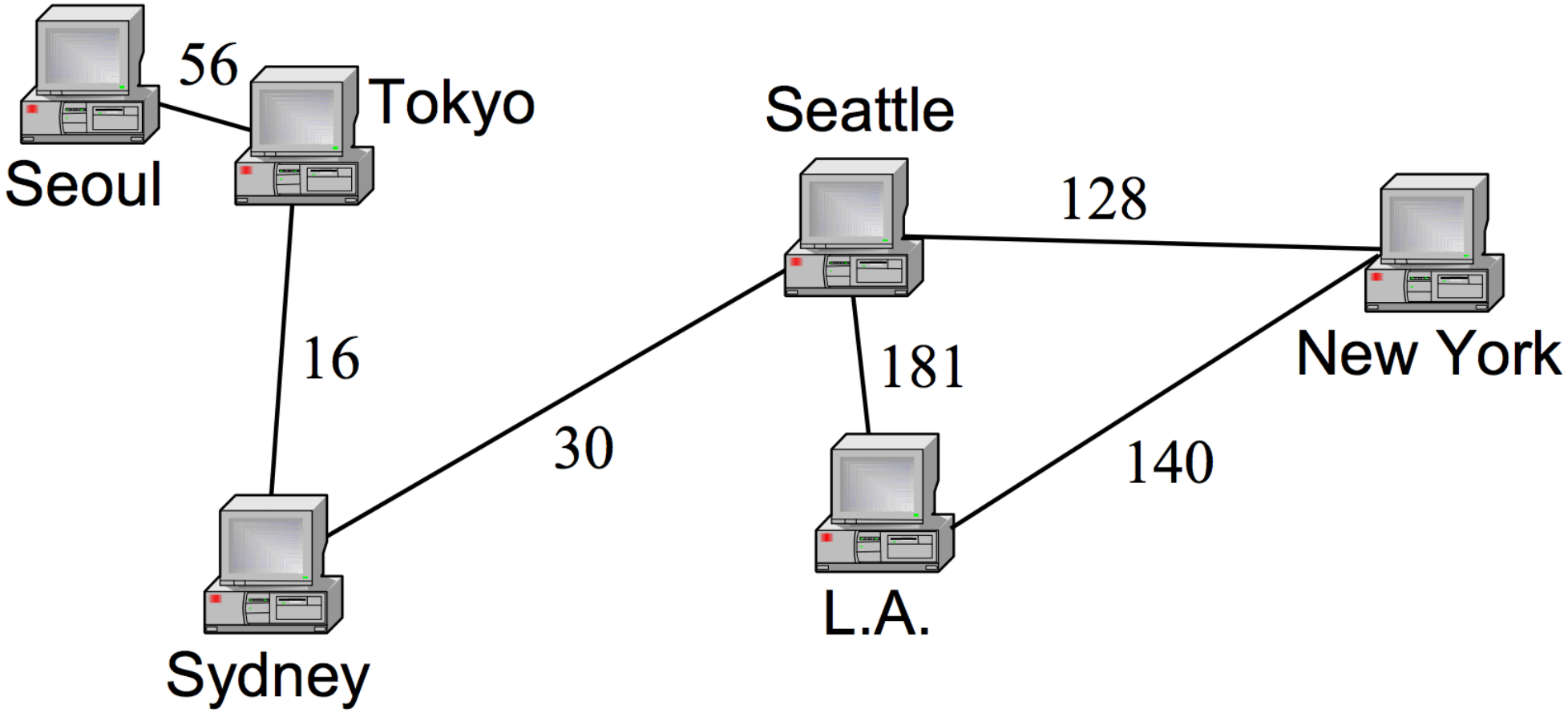
How about this problem?

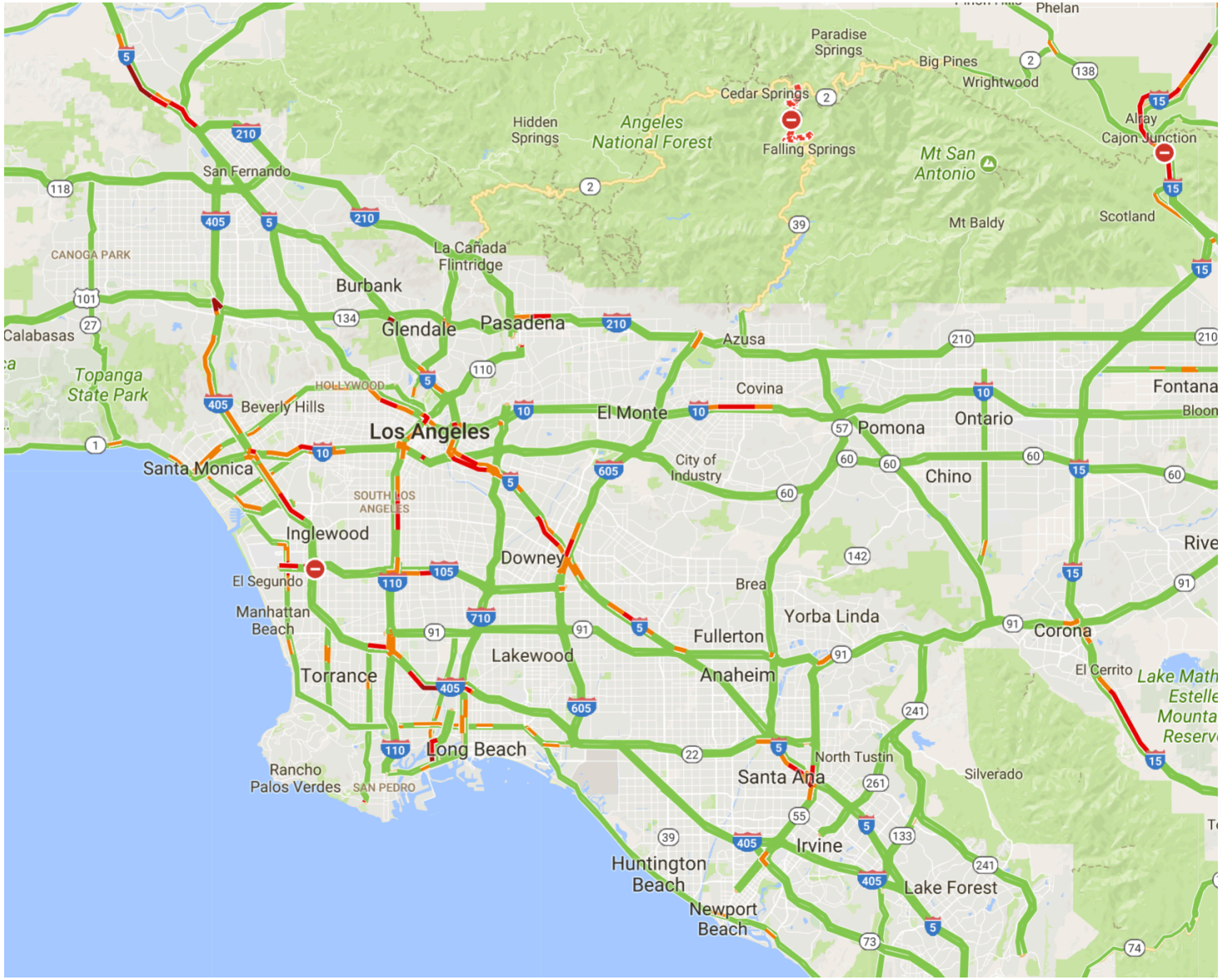


What about a network?

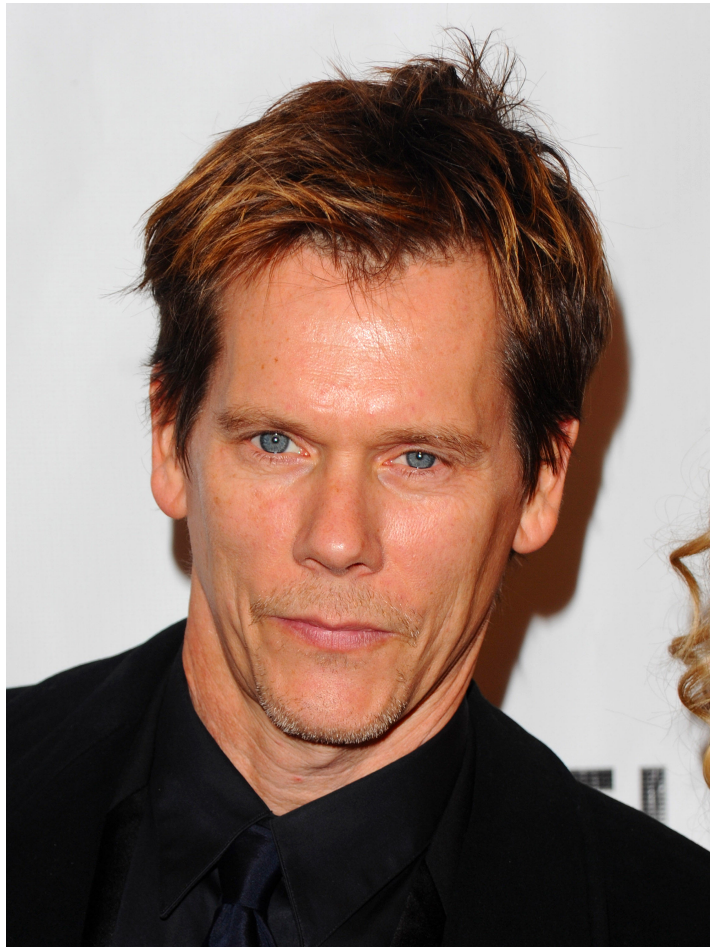


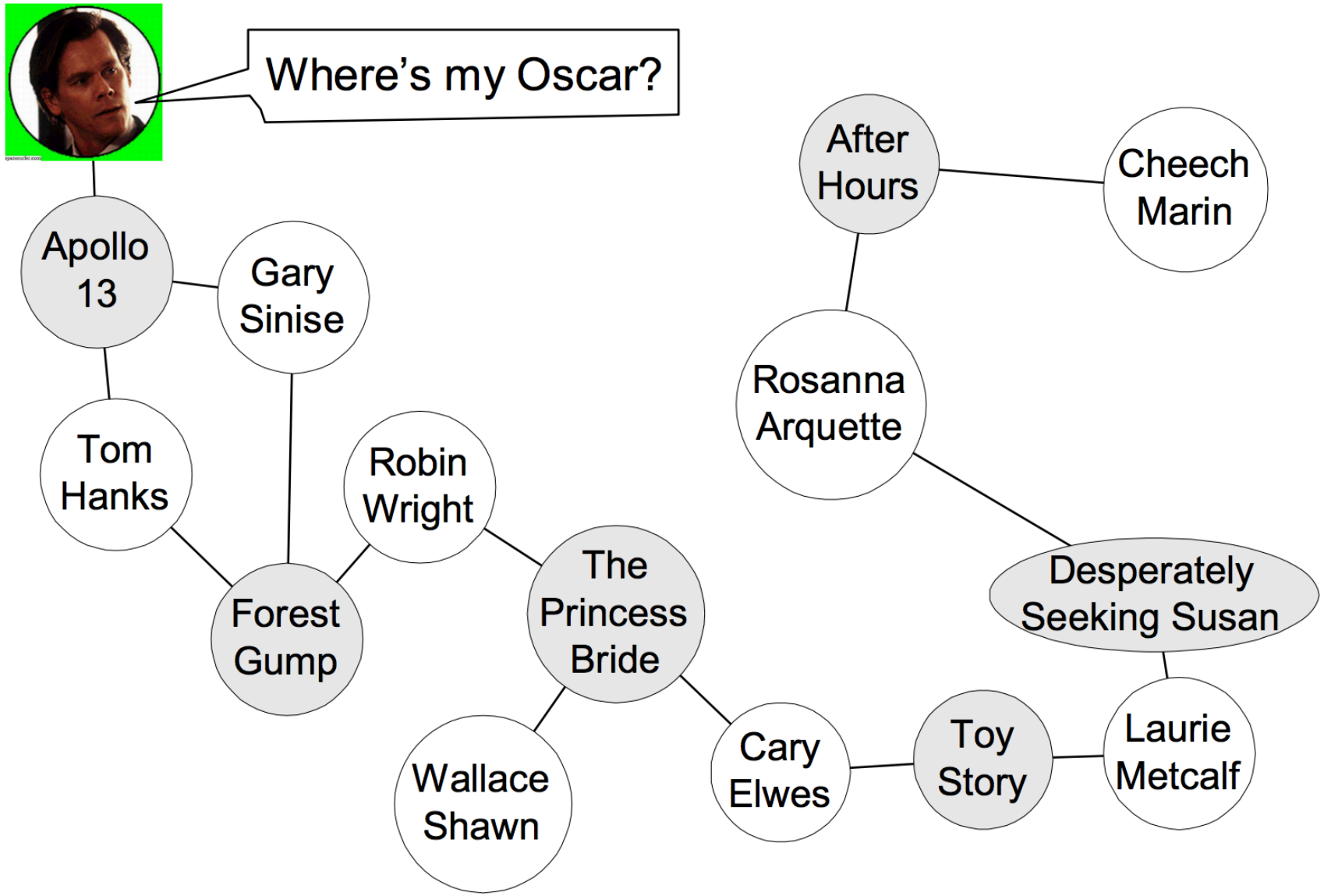
What about a network?

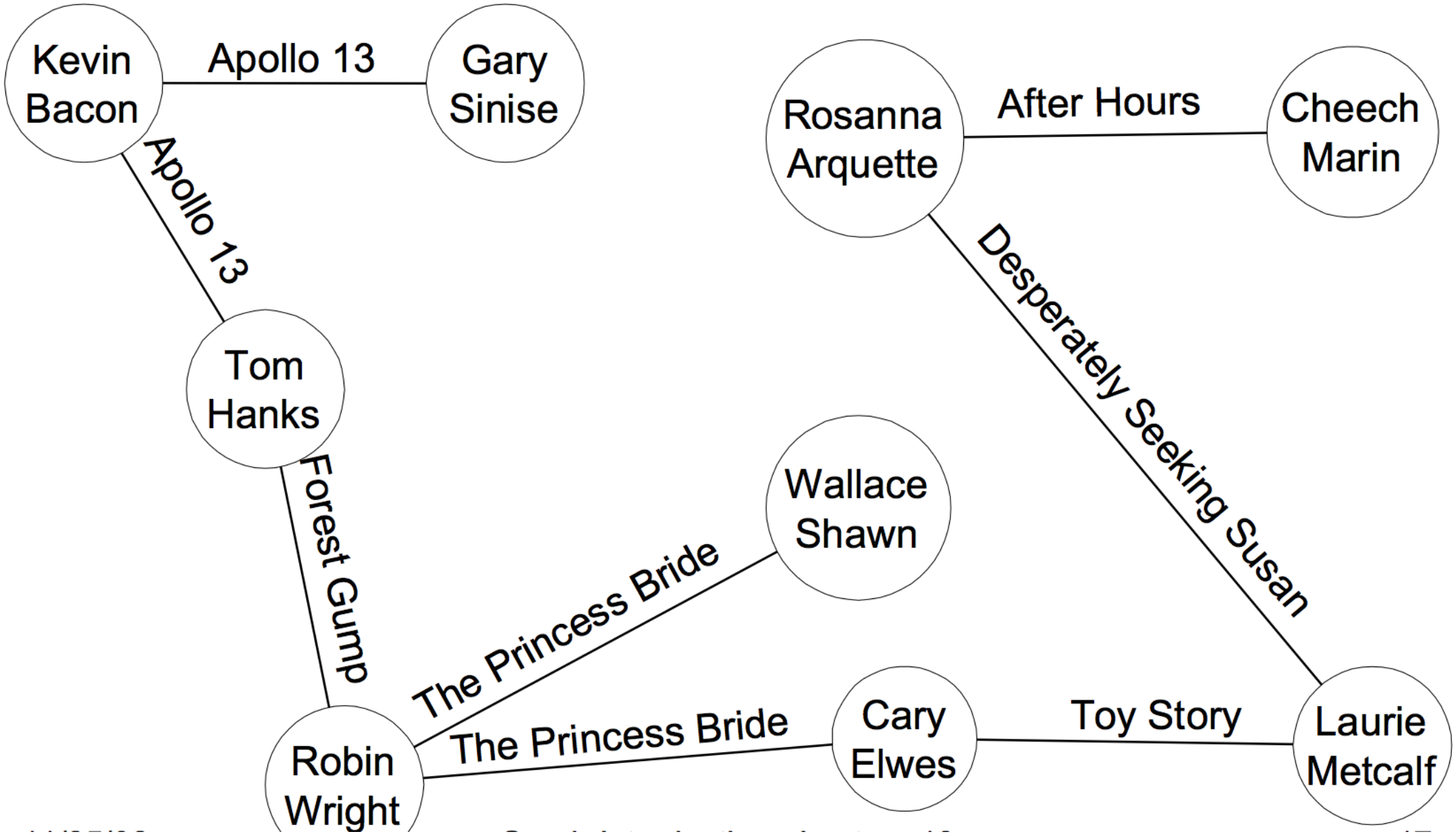



Phelan
Paradise Springs
Big Pines
Wrightwood
Cedar Springs
Hidden Springs
Angeles National Forest
Falling Springs
Mt San Antonio
Cajon Junction
San Fernando
Mt Baldy
Scotland
CANOGA PARK
La Cañada Flintridge
Burbank
Glendale
Pasadena
Azusa
Calabasas
Topanga State Park
HOLLYWOOD
Beverly Hills
Los Angeles
El Monte
Covina
Pomona
Ontario
Fontana
Santa Monica
City of Industry
Chino
SOUTH LOS ANGELES
Inglewood
Downey
El Segundo
Brea
Yorba Linda
Manhattan Beach
Fullerton
Corona
Lakewood
Torrance
Anaheim
El Cerrito
Long Beach
North Tustin
Rancho Palos Verdes
SAN PEDRO
Santa Ana
Silverado
Irvine
Huntington Beach
Lake Forest
Newport Beach

…and let's not forget:




Where's my Oscar?
Apollo 13
Gary Sinise
Tom Hanks
Forest Gump
Robin Wright
The Princess Bride
Wallace Shawn
Cary Elwes
Toy Story
Laurie Metcalf
Desperately Seeking Susan
Rosanna Arquette
After Hours
Cheech Marin


Kevin Bacon
Apollo 13
Gary Sinise
Apollo 13
Tom Hanks
Forest Gump
Rosanna Arquette
After Hours
Cheech Marin
Desperately Seeking Susan
Wallace Shawn
The Princess Bride
Robin Wright
The Princess Bride
Cary Elwes
Toy Story
Laurie Metcalf

Defining a Graph

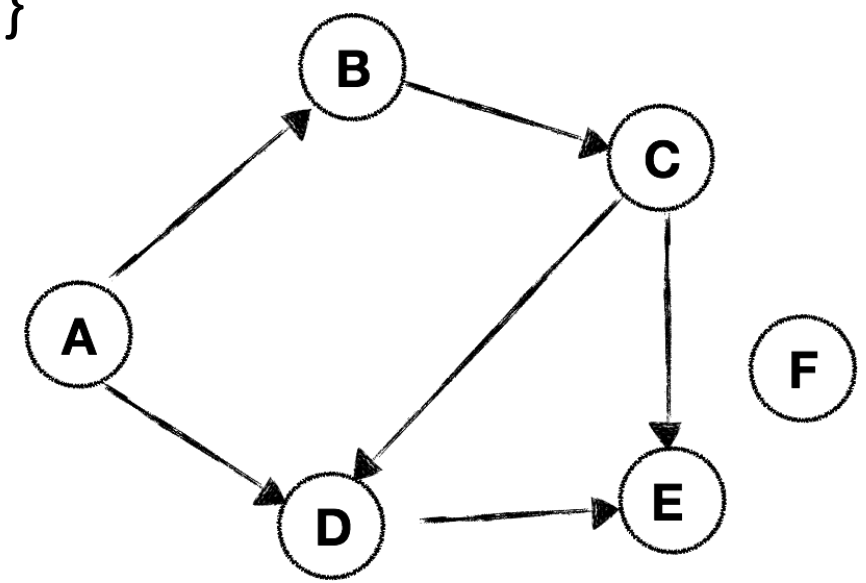
- A graph is simply a collection of nodes plus edges
  - Linked lists, trees, and heaps are all special cases of graphs
  - The nodes are known as vertices (node = "vertex")
- Formal Definition:
  - A graph *G* is a pair (*V, E*) where
    - *V* is a set of vertices or nodes
    - *E* is a set of edges that connect vertices

Defining a Graph, Formally

- 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

An Example

- Here is a directed graph G = (V, E)
  - Each edge is a pair (v1, v2), where v1, v2 are vertices in V
  - V = {A, B, C, D, E, F}
  - E = {(A,B), (A,D), (B,C), (C,D), (C,E), (D,E)}

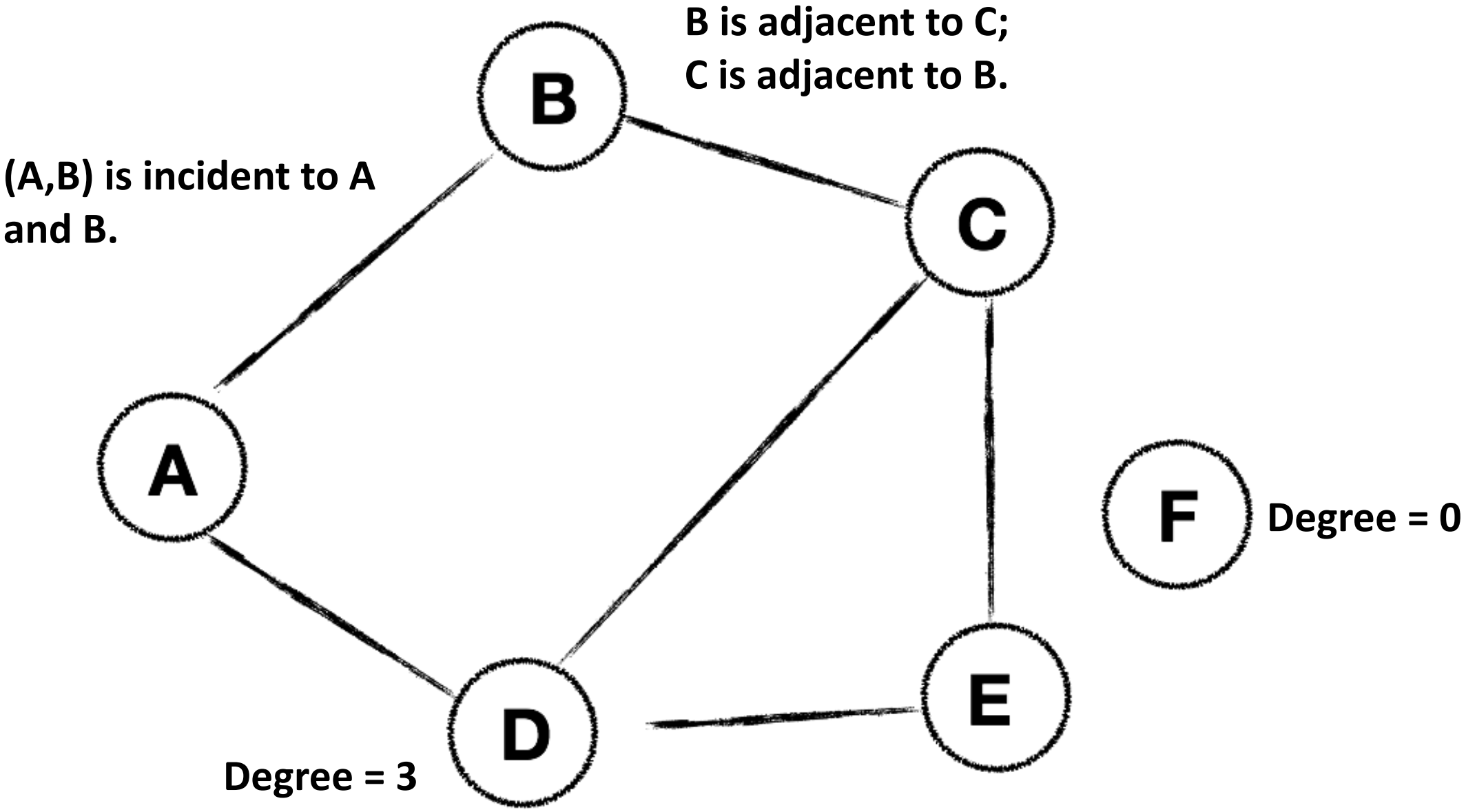


Directed vs Undirected Eduges

- If the order of edge pairs (v1, v2) matters, the graph is directed (also called a **digraph**):
  - (v1, v2) != (v2, v1)
- If the order of edge pairs (v1, v2) does not matter, the graph is called an **undirected graph**: in this case,
  - (v1, v2) = (v2, v1)

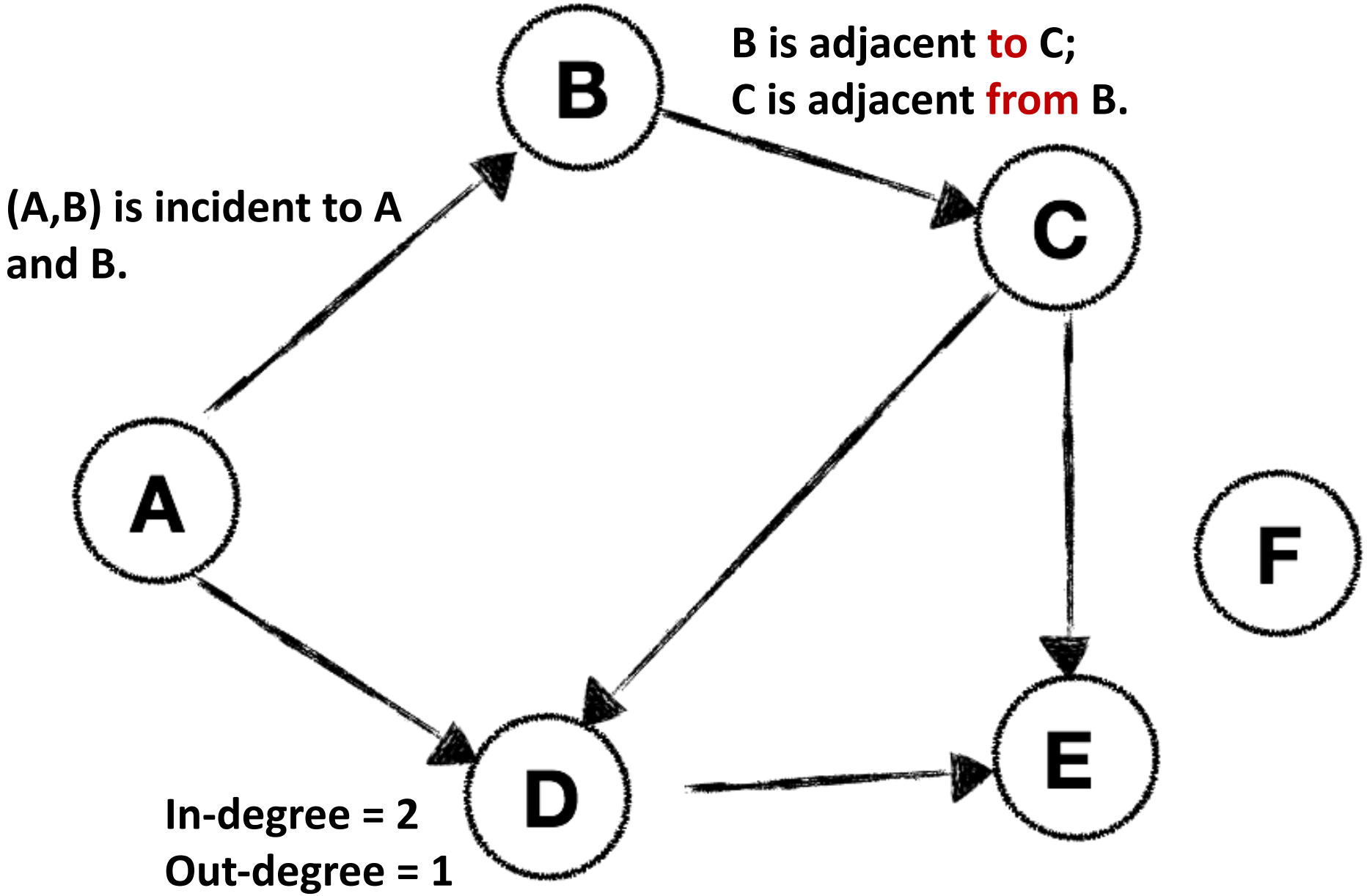
Terminology

- 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*)

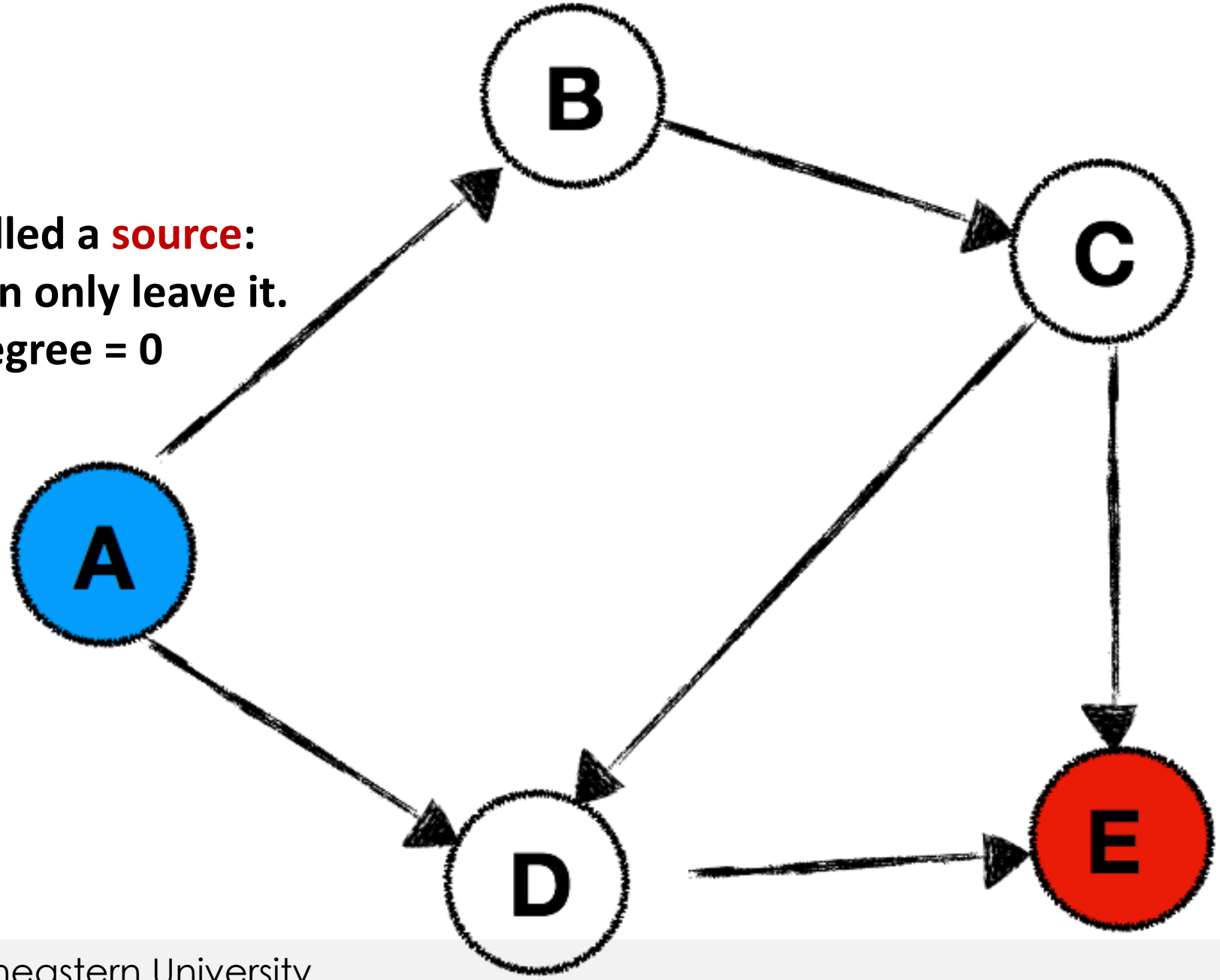

B is adjacent to C;
C is adjacent to B.
(A,B) is incident to A and B.
B
C
A
F
Degree = 0
D
E
Degree = 3

More terminology…

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

A is called a source:
one can only leave it.
* In-degree = 0

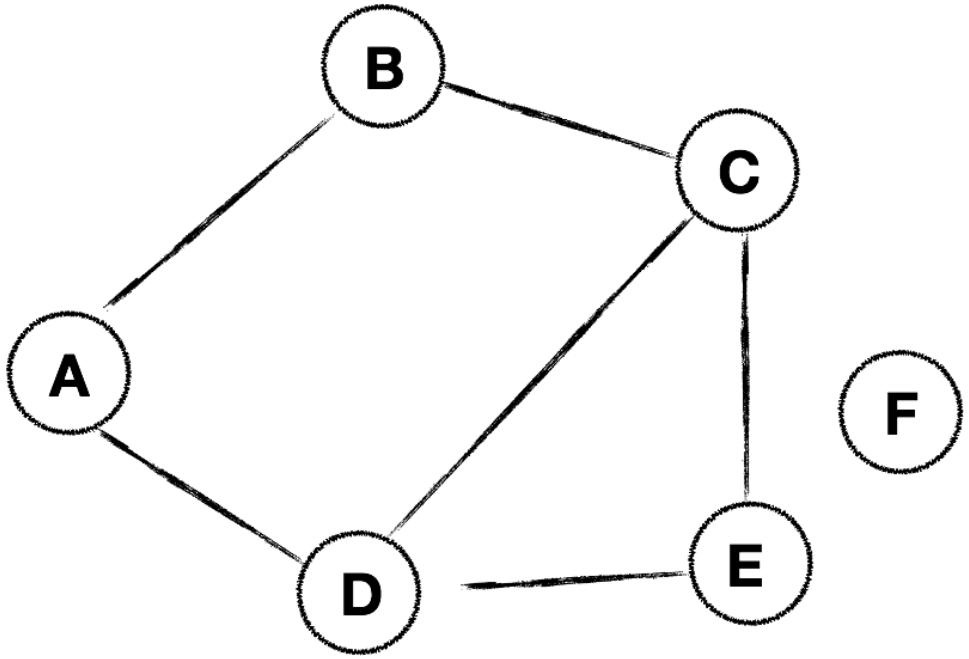


E is called a sink:
one can only arrive, and never leave.
* Out-degree = 0

Graph Representations

- Space and time are analyzed in terms of:
  - Number of vertices = |V| and
  - Number of edges = |E|
- There are at least two ways of representing graphs:
  - The adjacency matrix representation
  - The adjacency list representation

Adjacency Matrix

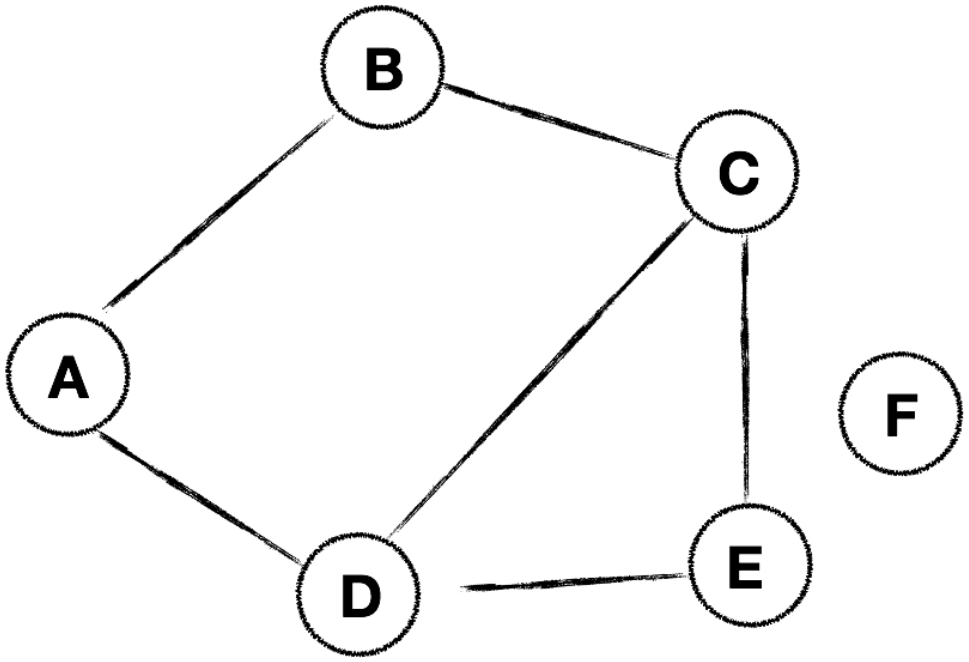


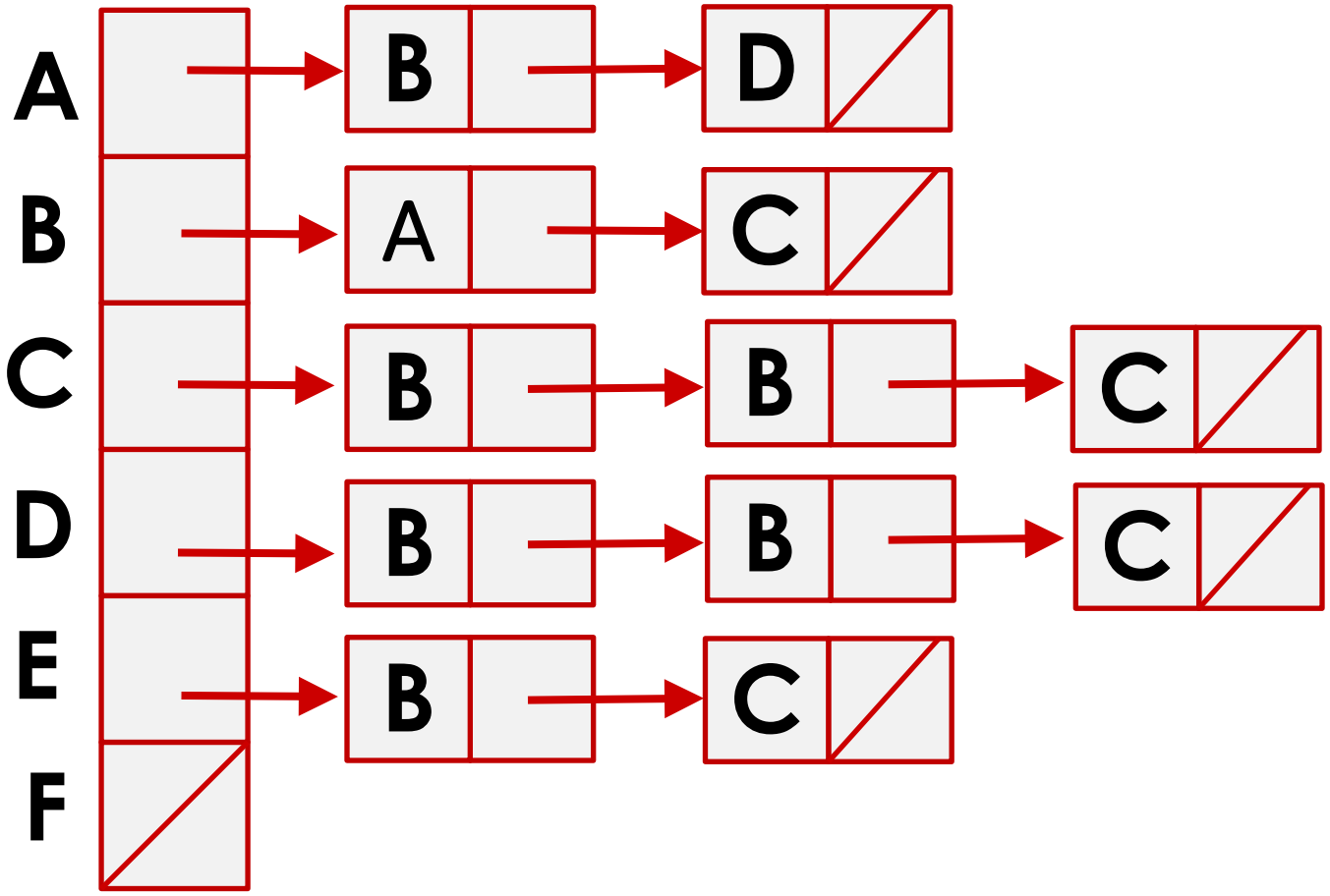
$$M(v, w) = \begin{cases} 1 \text{ if } (v,w) \text{ is in } E \\ 0 \text{ otherwise} \end{cases}$$

	A	B	C	D	E	F
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

Space = $|V|^2$

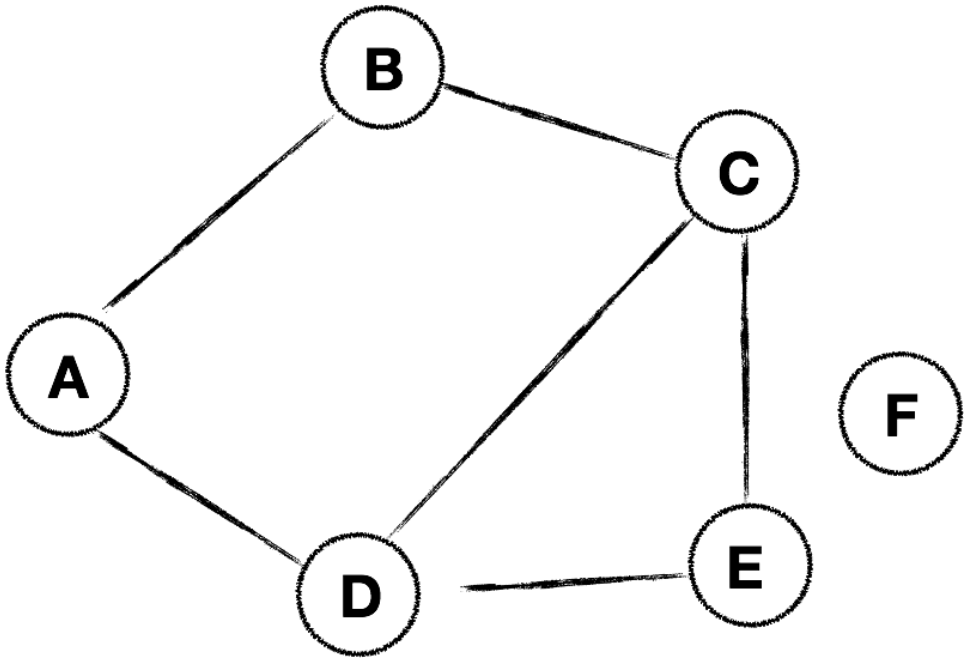
Adjacency List

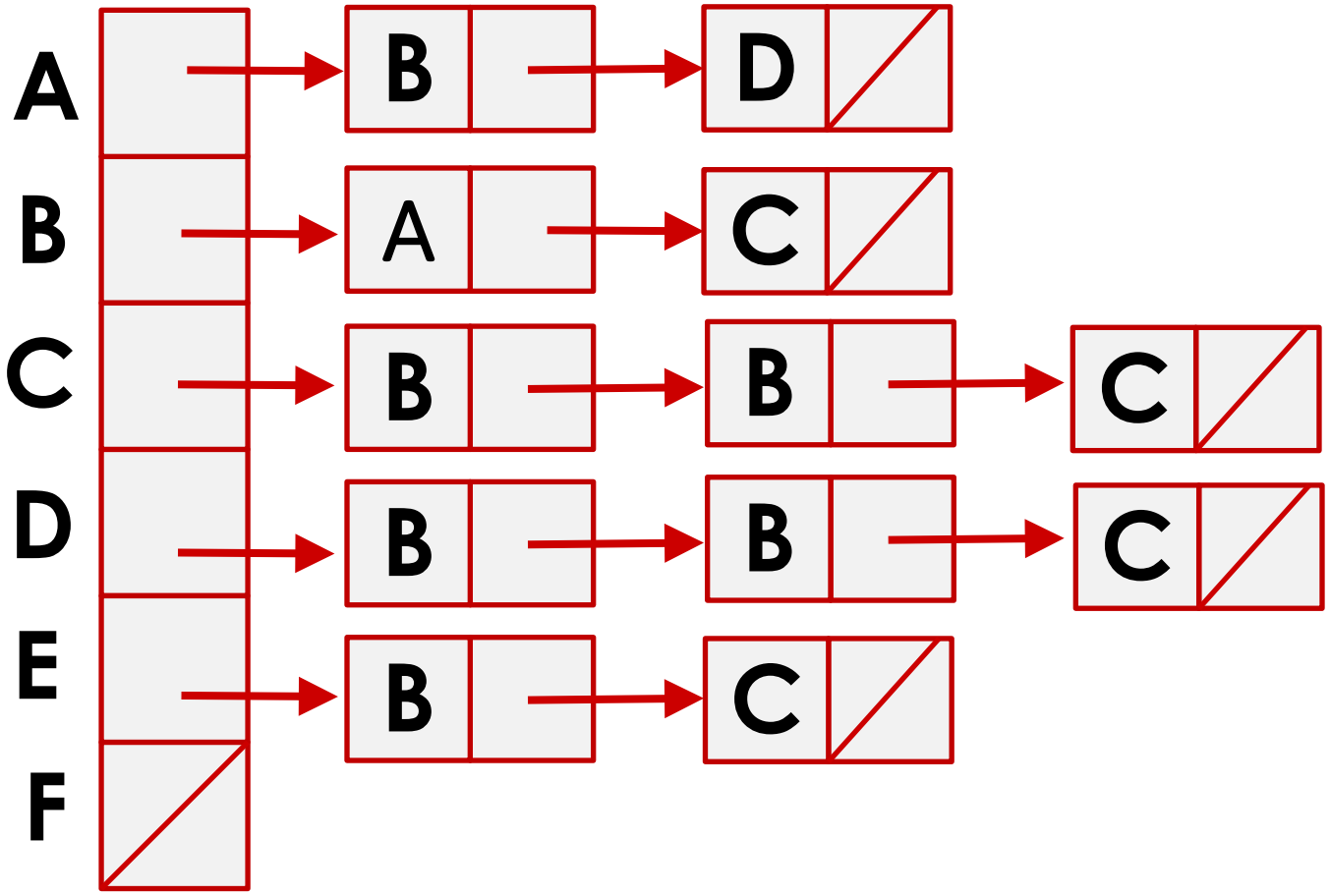




Adjacency List

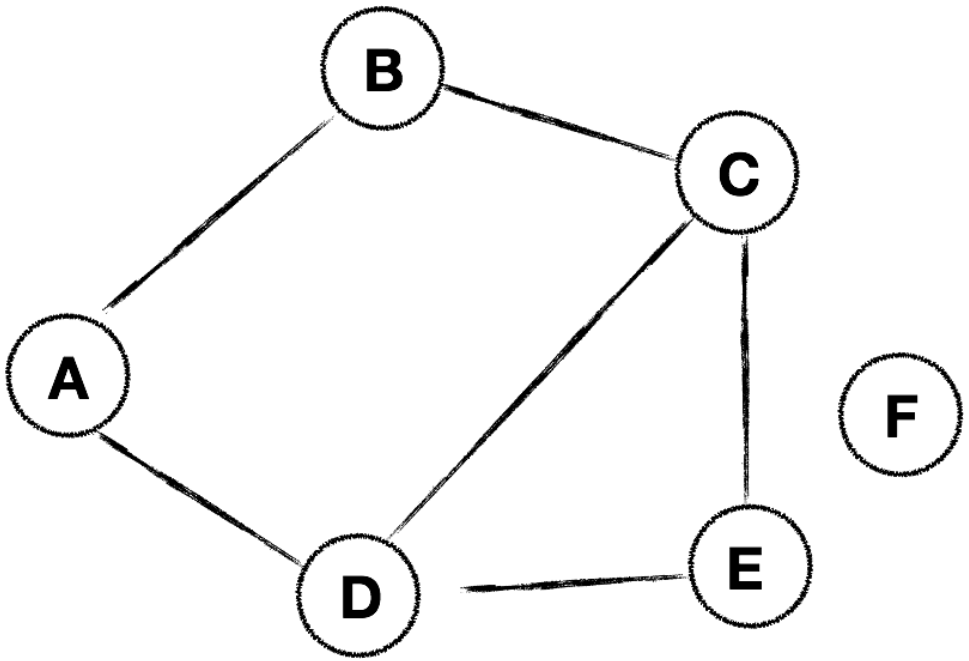
Formally:
For each *v* in *V*, *L(v)* = list of *w* such that (*v*, *w*) is in *E*

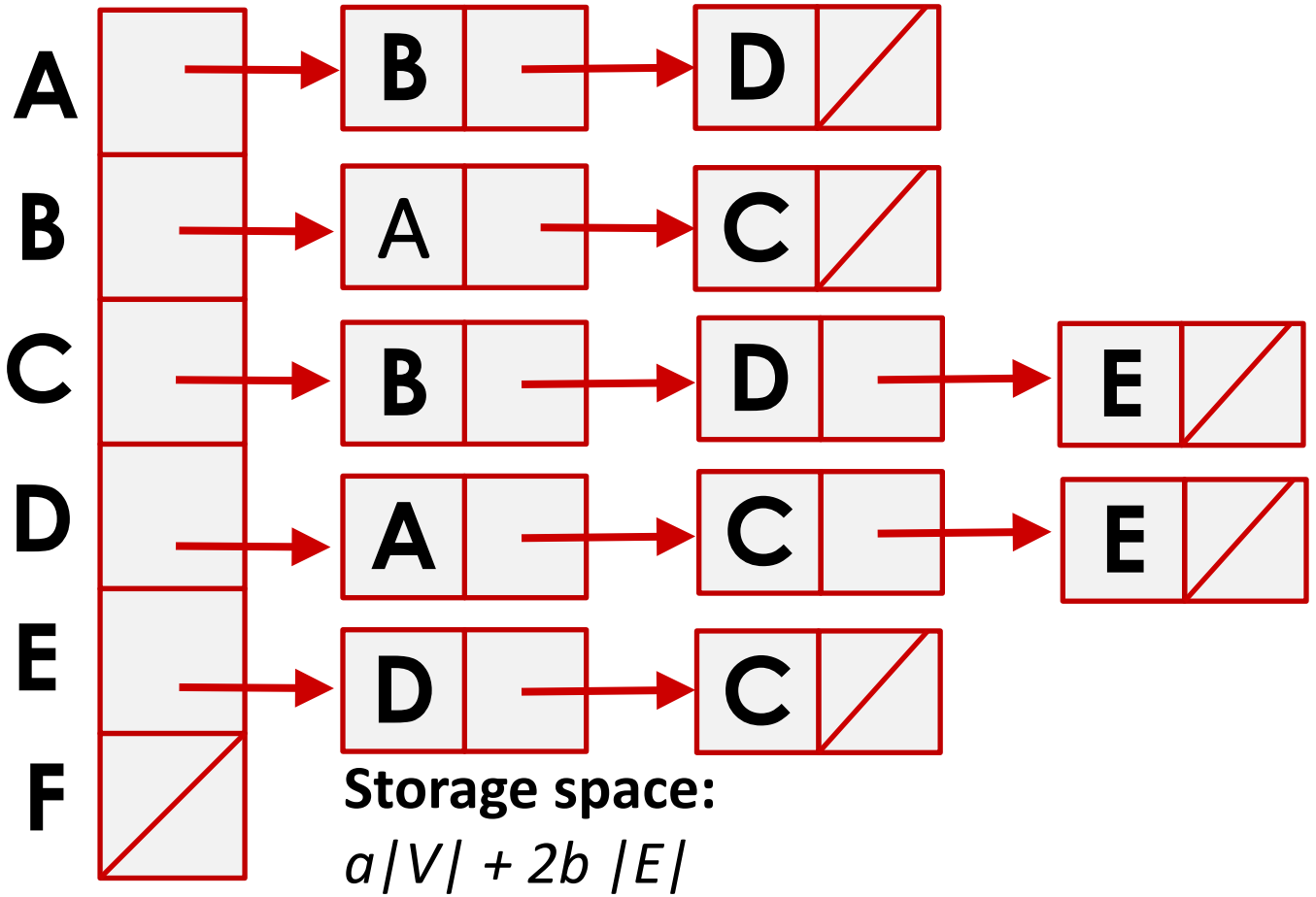




Adjacency List

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

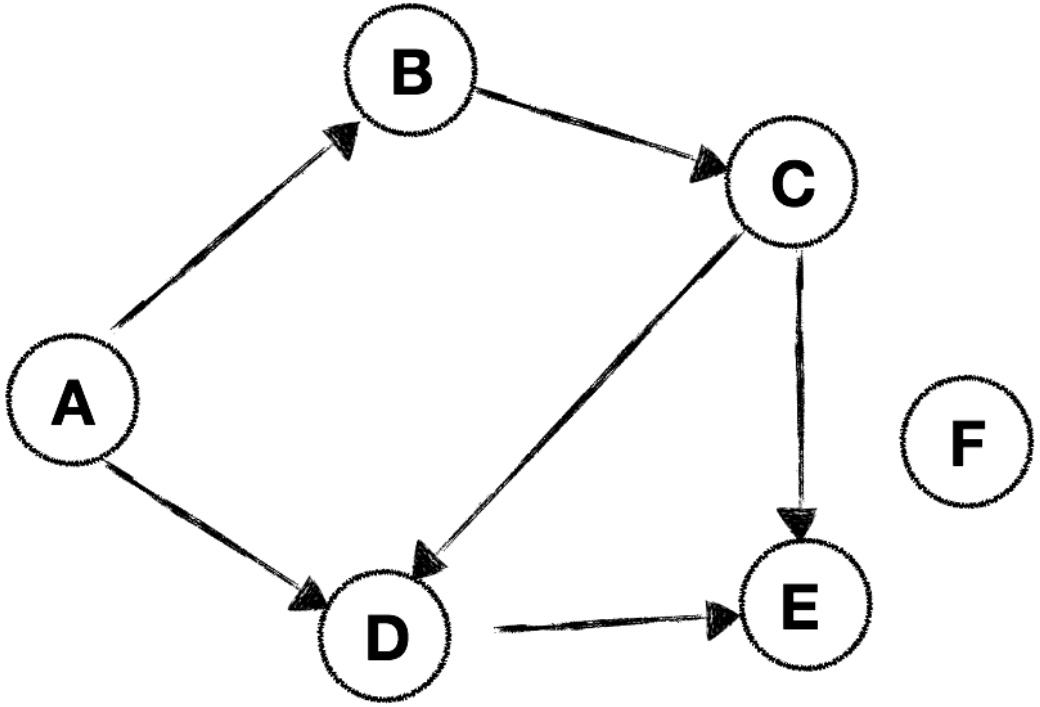


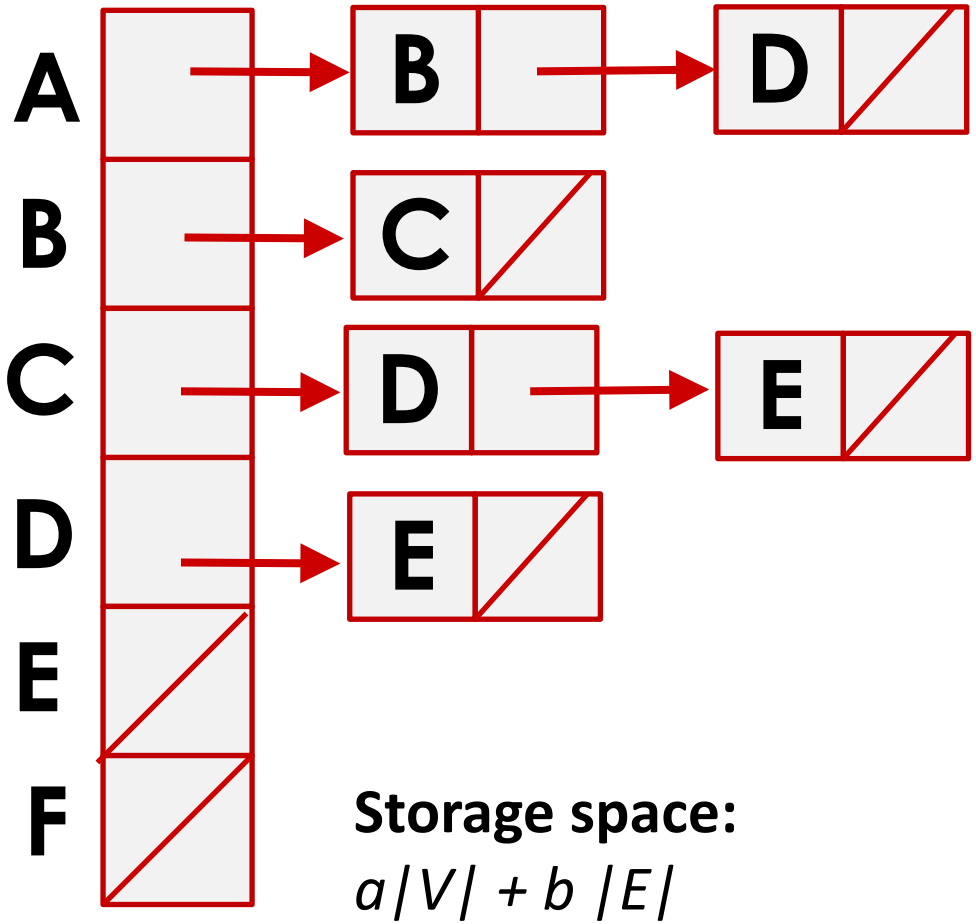


Storage space:
$a|V| + 2b|E|$
a = sizeof (node)
b = size of linked list element

Adjacency List: Directed Graph

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





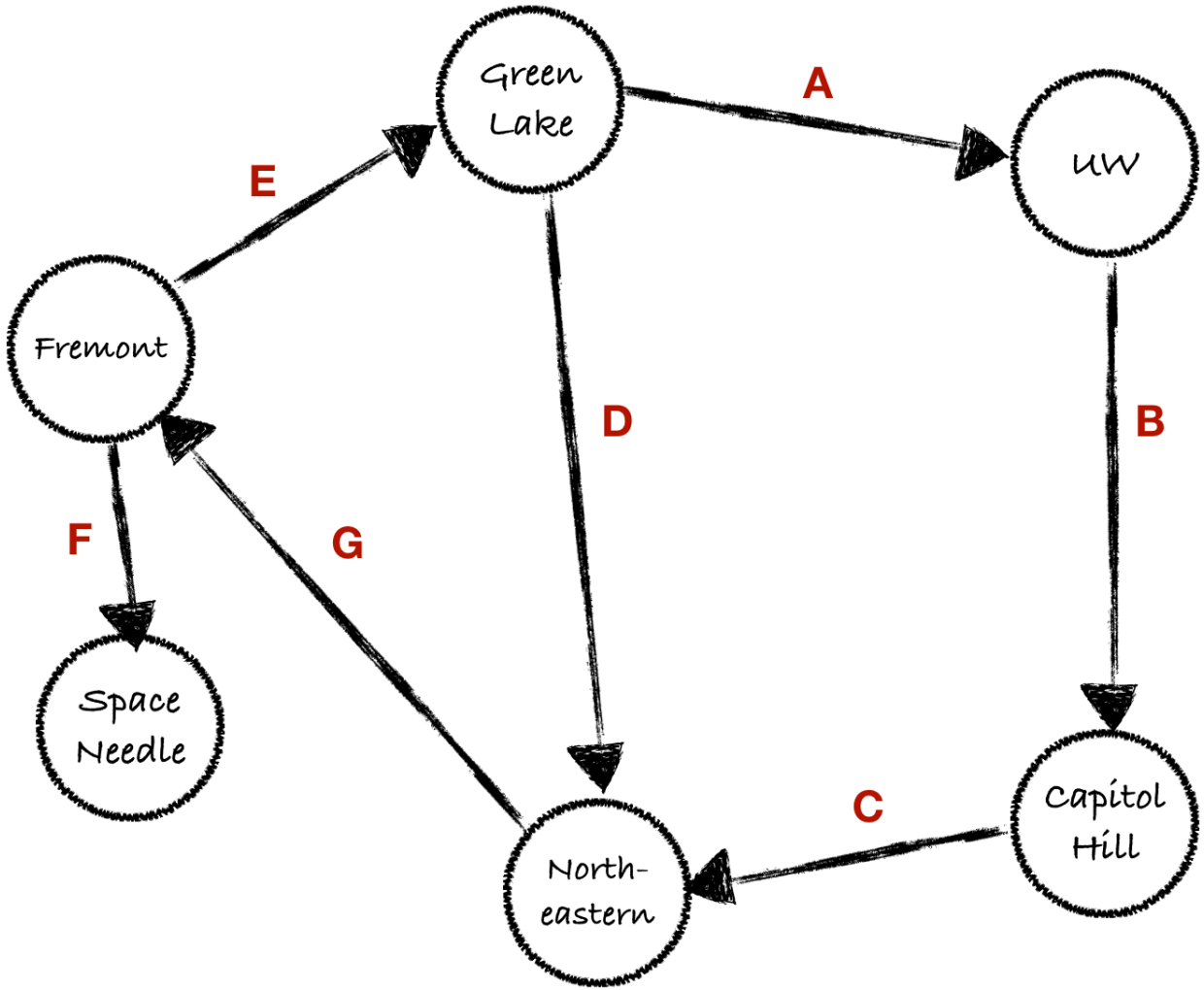
Storage space:
$a|V| + b|E|$
a = sizeof (node)
b = size of linked list element

Problem Solving with Graphs

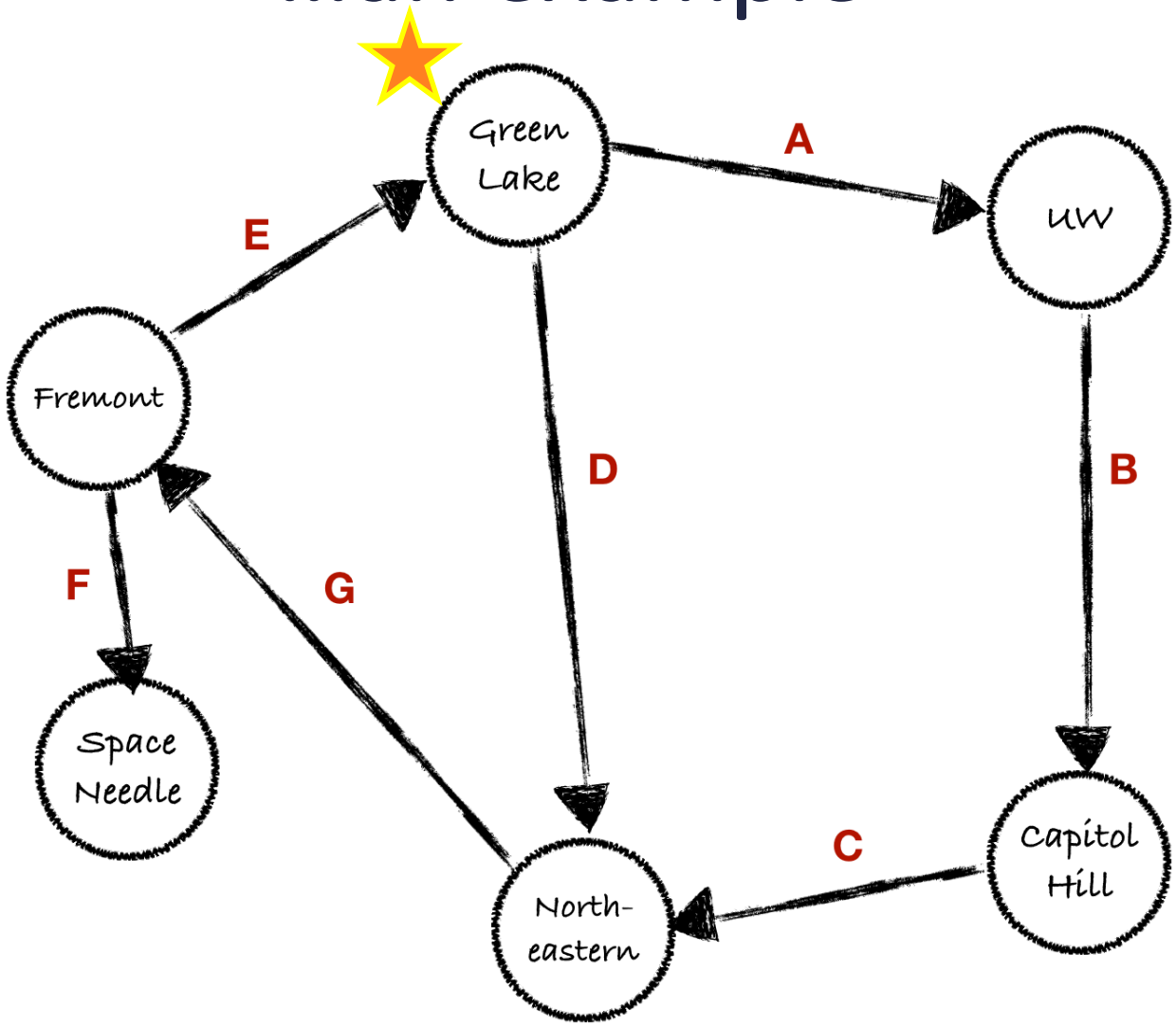
- Abstract the problem as a graph problem.
- Find an algorithm for solving the graph problem.
- Design data structures and algorithms to implement the graph solution.
- Write code

...an example

...an example

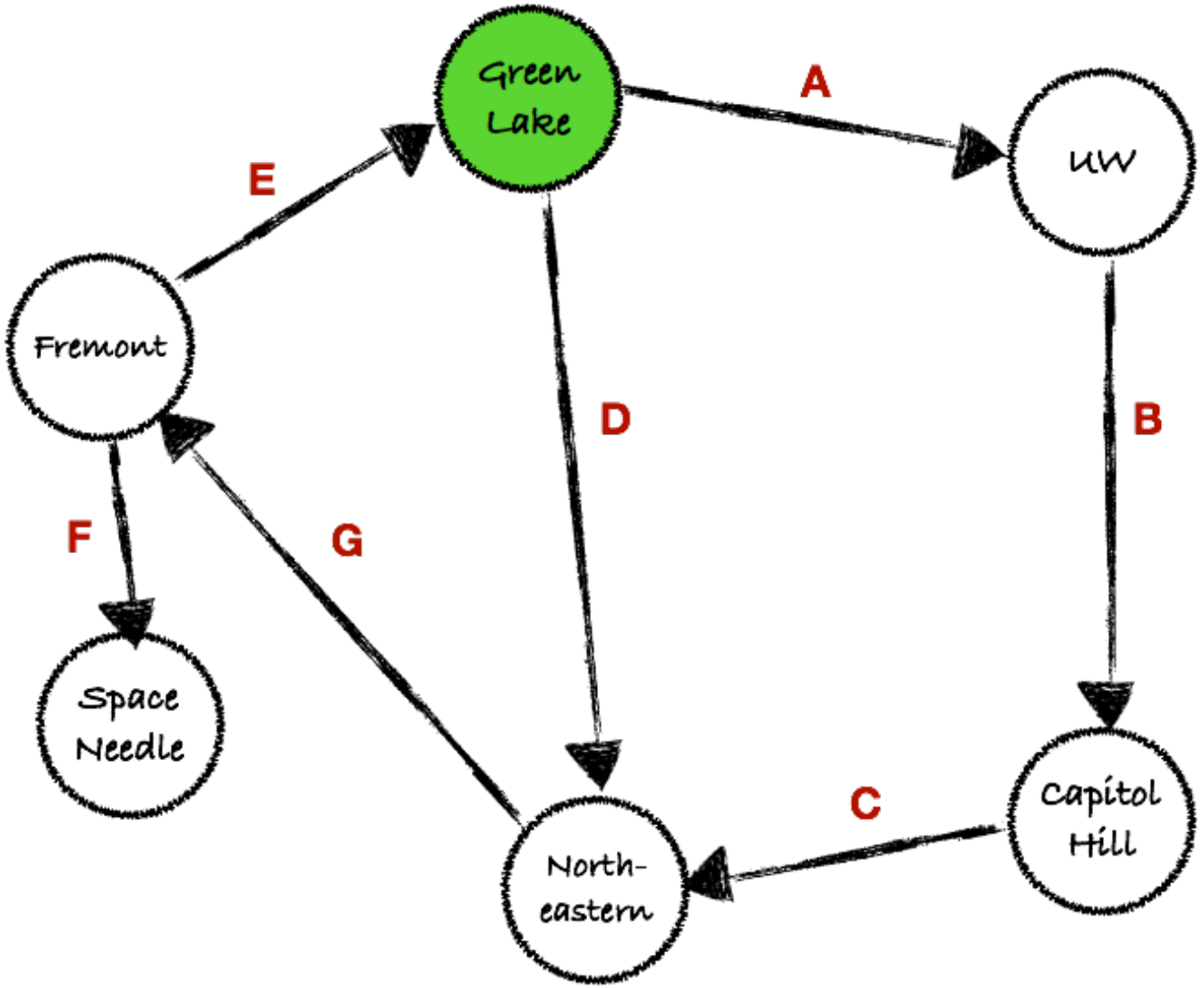

Green Lake
UW
Fremont
Space Needle
North-eastern
Capitol Hill
A
B
C
D
E
F
G

...an example



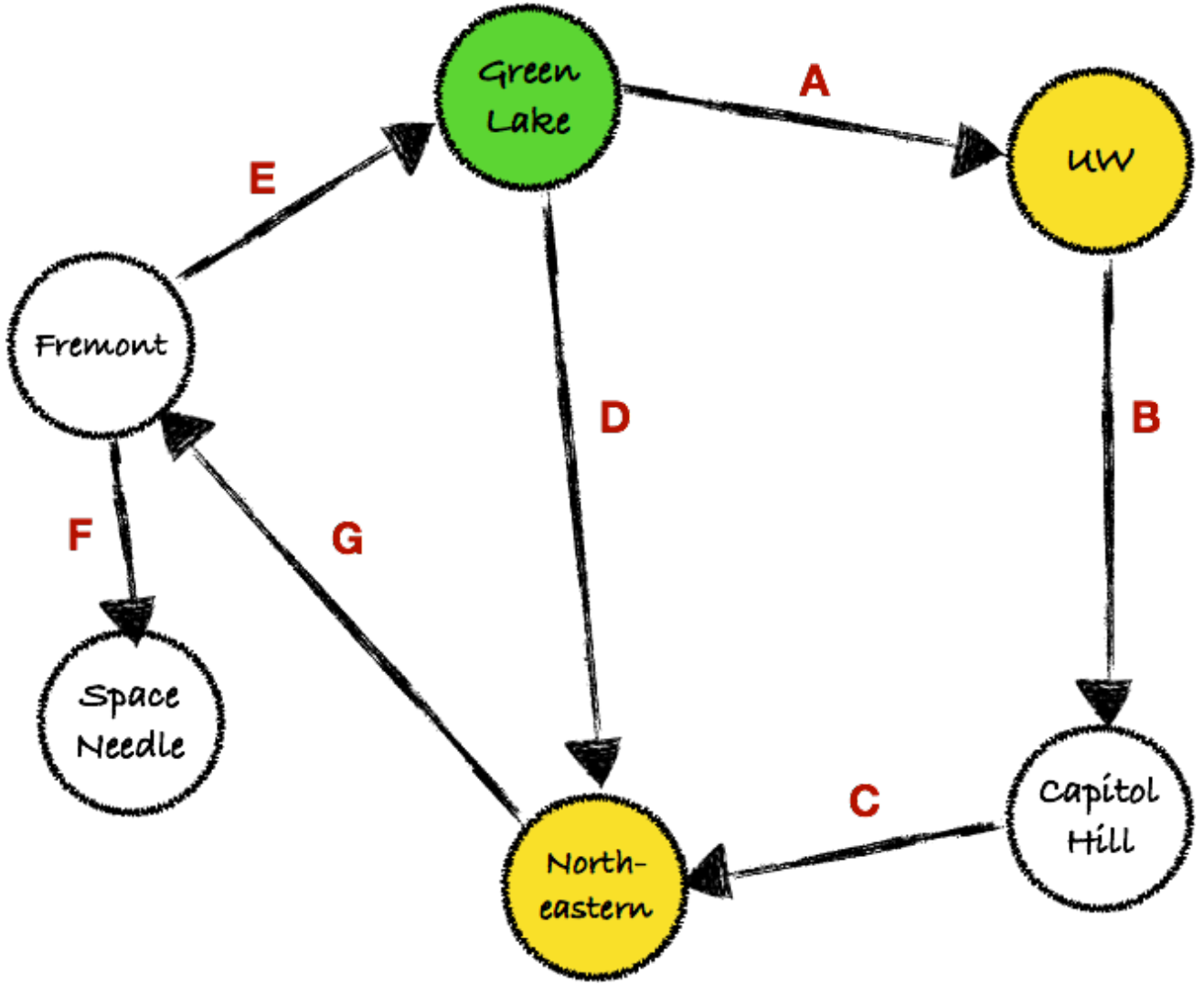
What's the best way for me to get from Green Lake to the Space Needle?

...an example



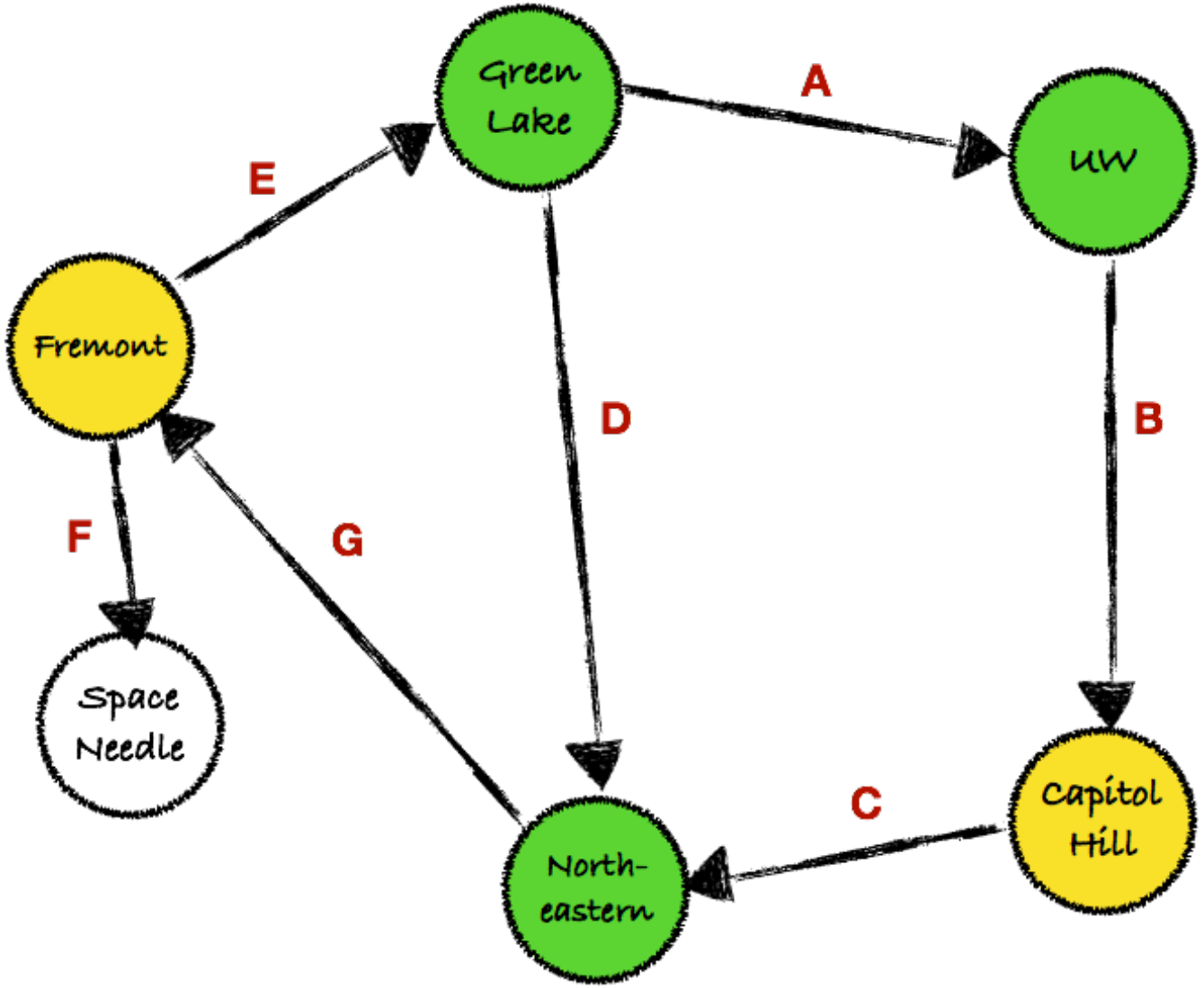
What's the best way for me to get from Green Lake to the Space Needle?

...an example



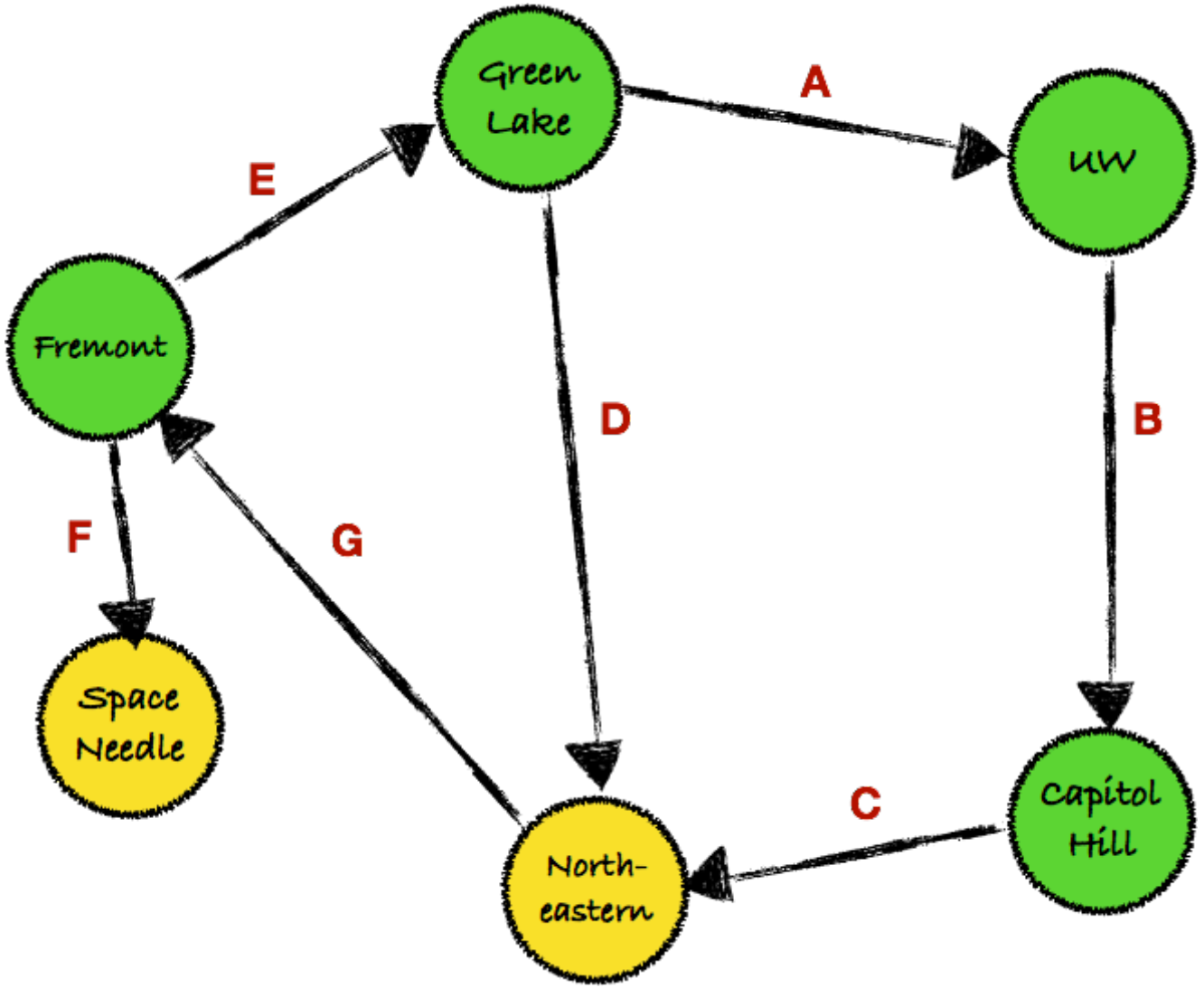
What's the best way for me to get from Green Lake to the Space Needle?

...an example



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

...an example



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

What's the algorithm?

- Start at the start.
- Look at all the neighbors. Are any of them the destination?

What's 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?

What's 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?
  - If no:
    - Look at all the neighbors of the neighbors of the neighbors. Are any of them the destination?

What's the algorithm?

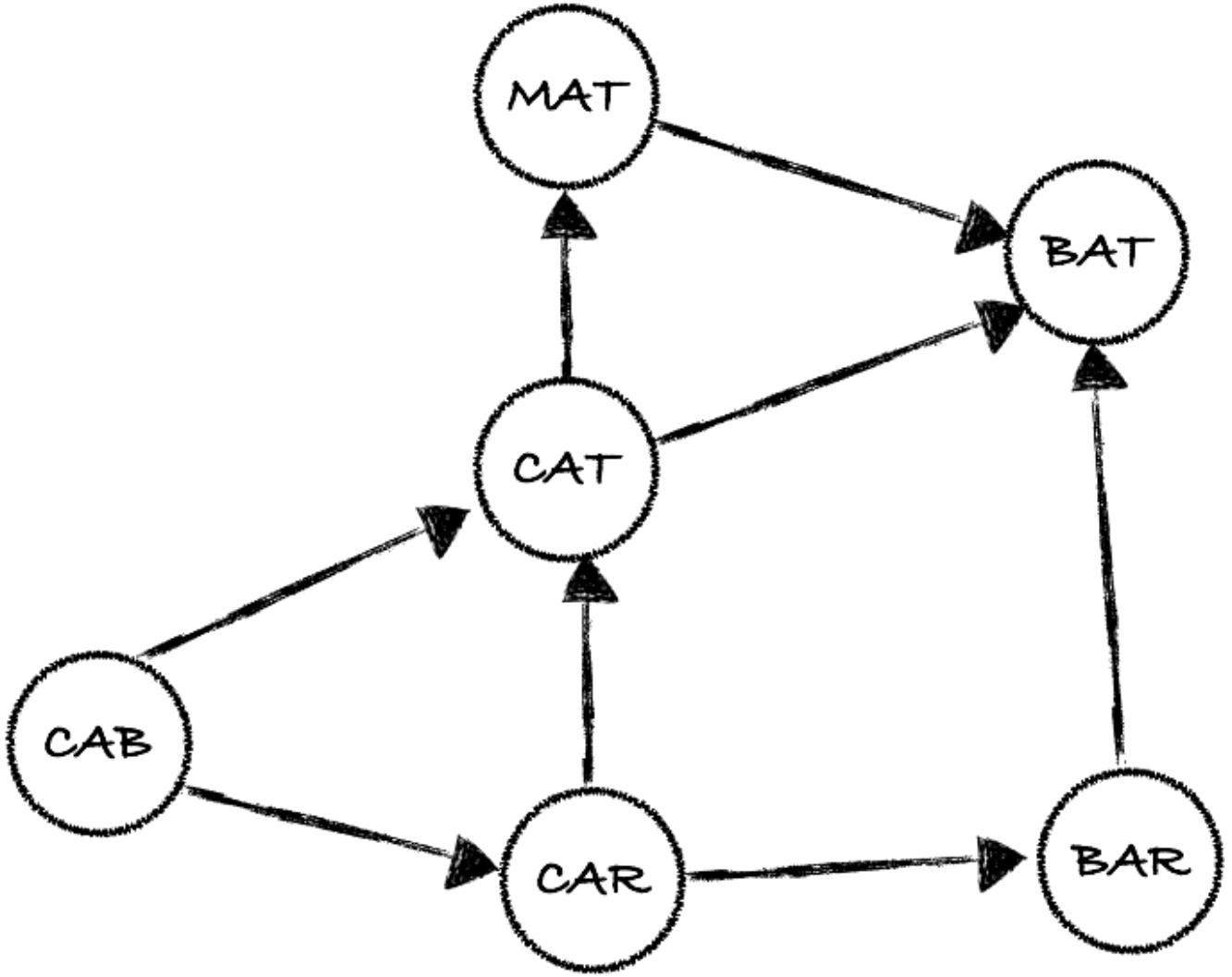
- Start at the start.
- Look at all the neighbors. Are any of them the destination?
- If no:
  - Look at all the neighbors of the neighbors. Are any of them the destination?
  - If no:
    - Look at all the neighbors of the neighbors of the neighbors. Are any of them the desitination?

This is breadth-first search!

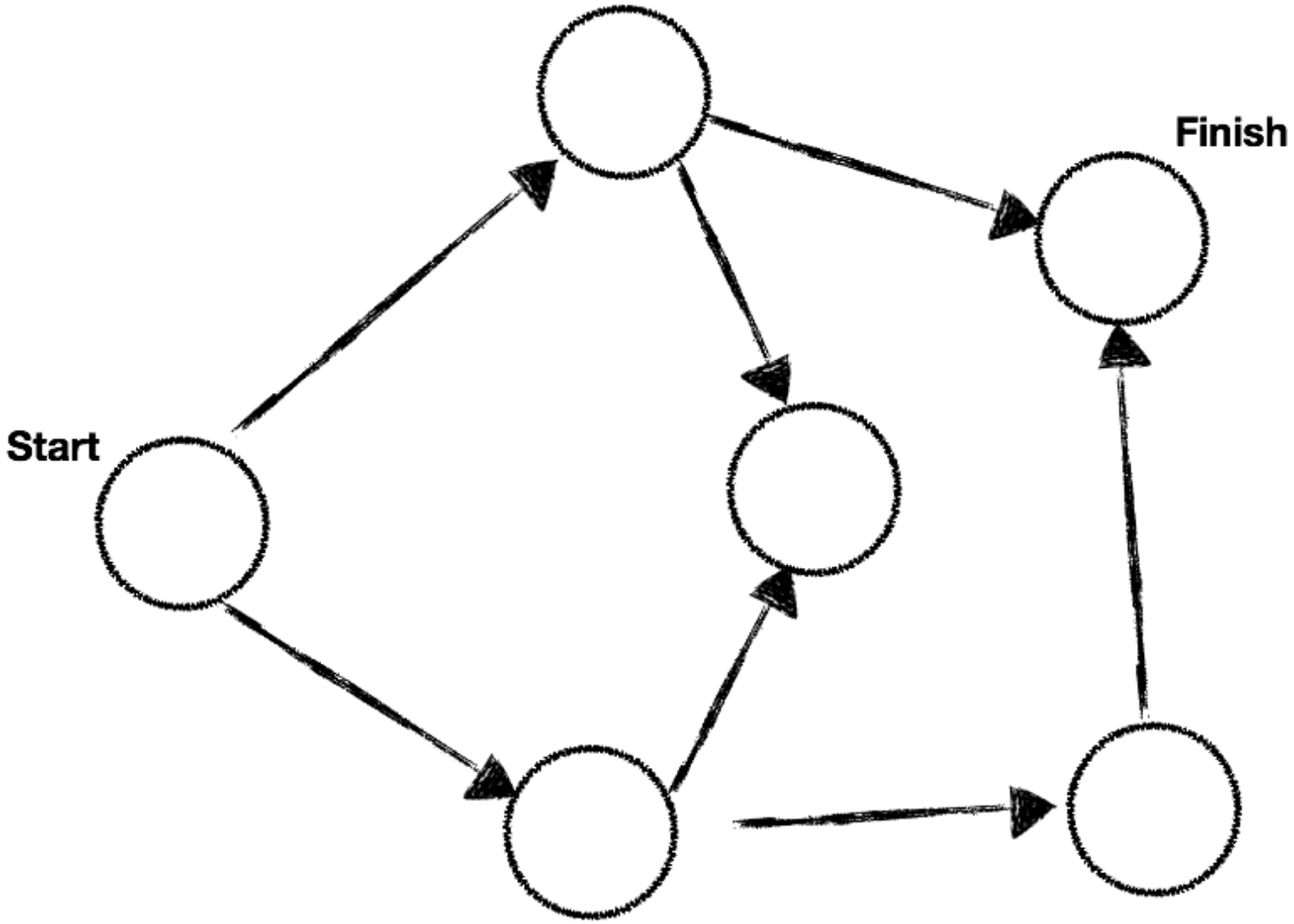
Breadth-first Search with Graphs

- 2 types of questions:
  - Is there a path from A to B?
  - What is the shortest path from A to B?

What's the shortest path from "CAB" to "BAT"?



What's the shortest path from "start" to "finish"?



Breadth-First Search Algorithm

- Let's look at some code.

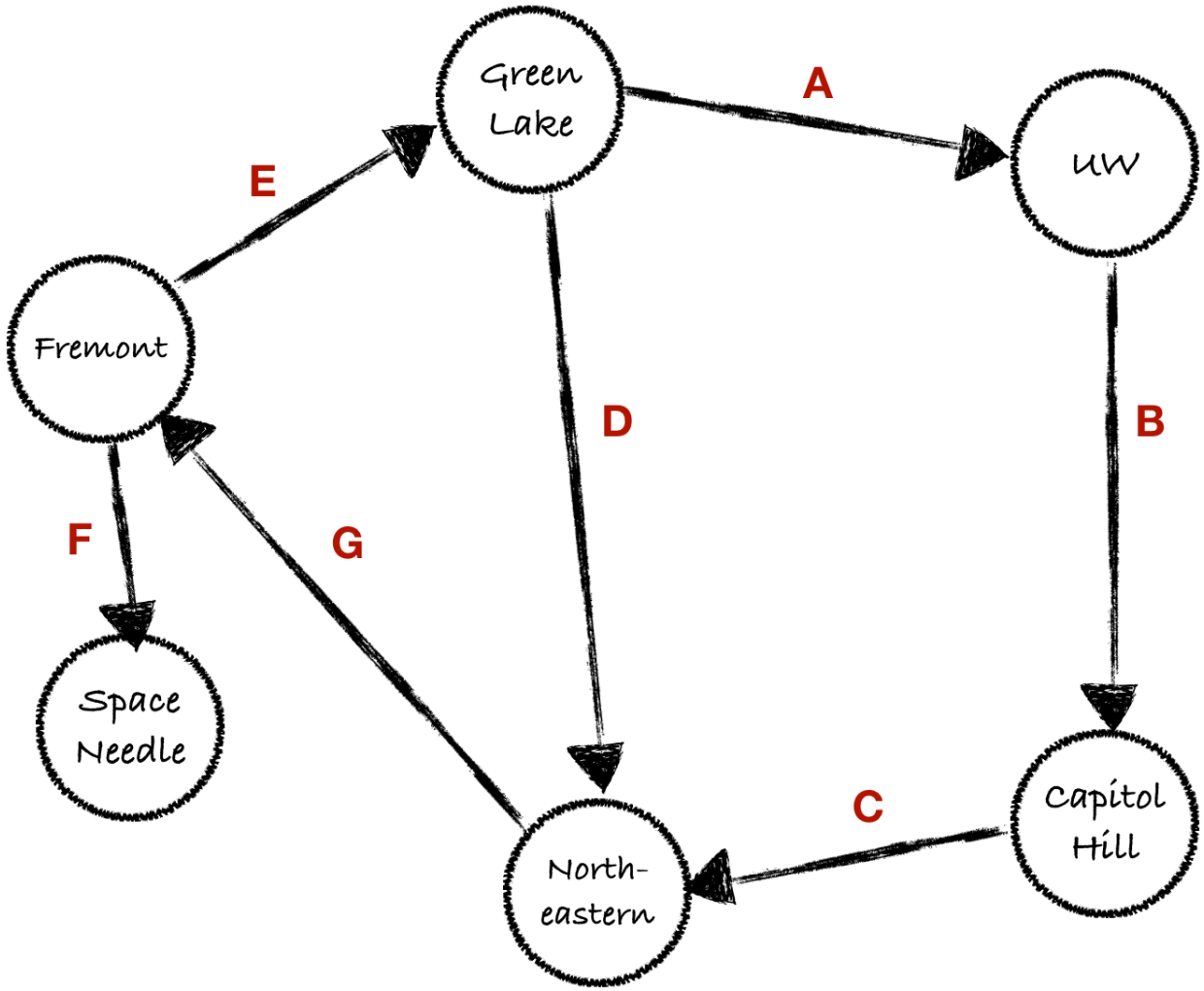
Things to add to the code

- Can the function return how long the path is?
- Can the function print out the path?

Running Time

- 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)

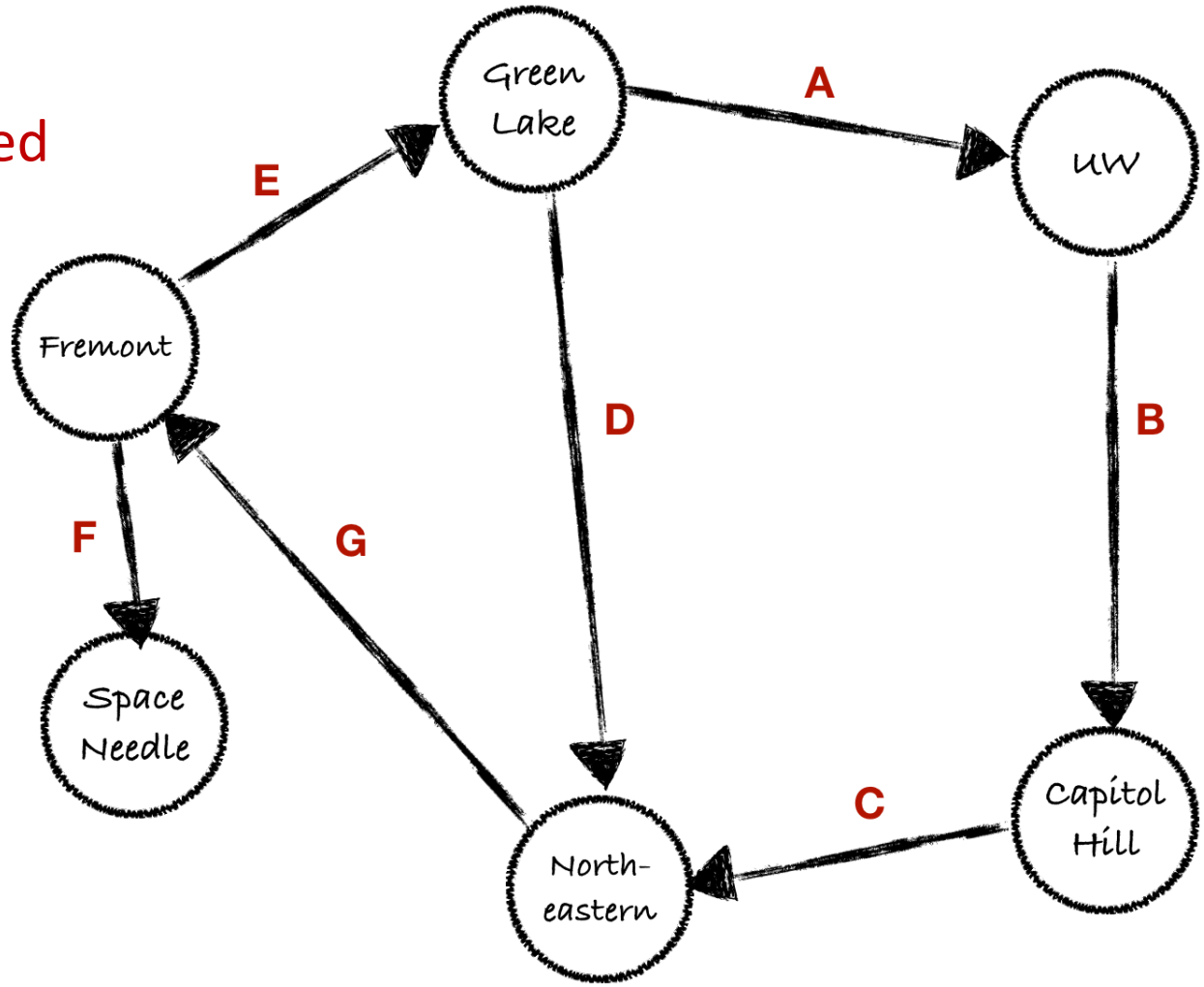
Back to this:


Green Lake
UW
Fremont
Space Needle
North-eastern
Capitol Hill
A
B
C
D
E
F
G

Back to this:

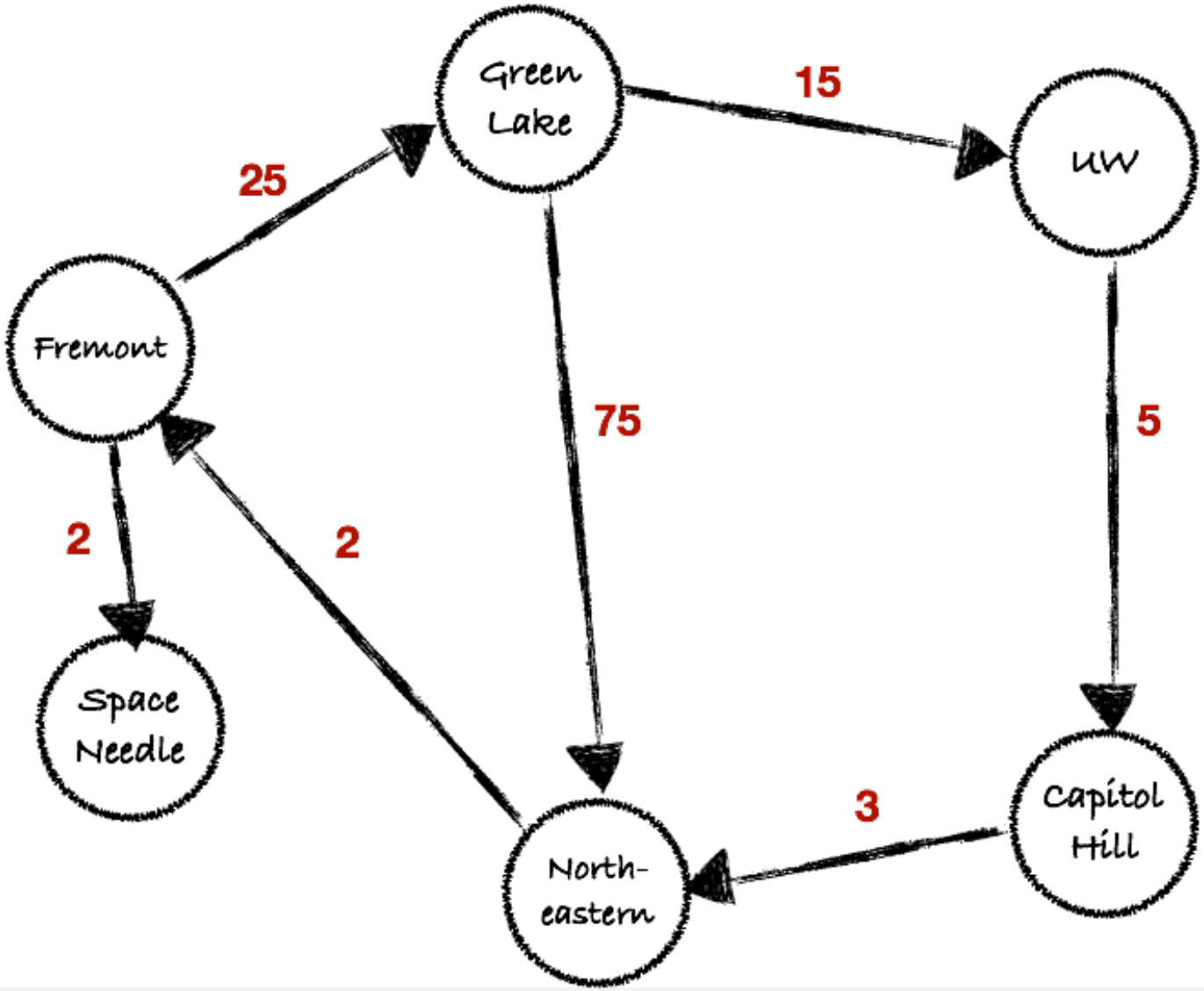
We know…
Not all edges are created equal!

Just because it's the shortest number of connections, doesn't mean it's the fastest to get there.



Weight the edges:

How do we find the fastest path?

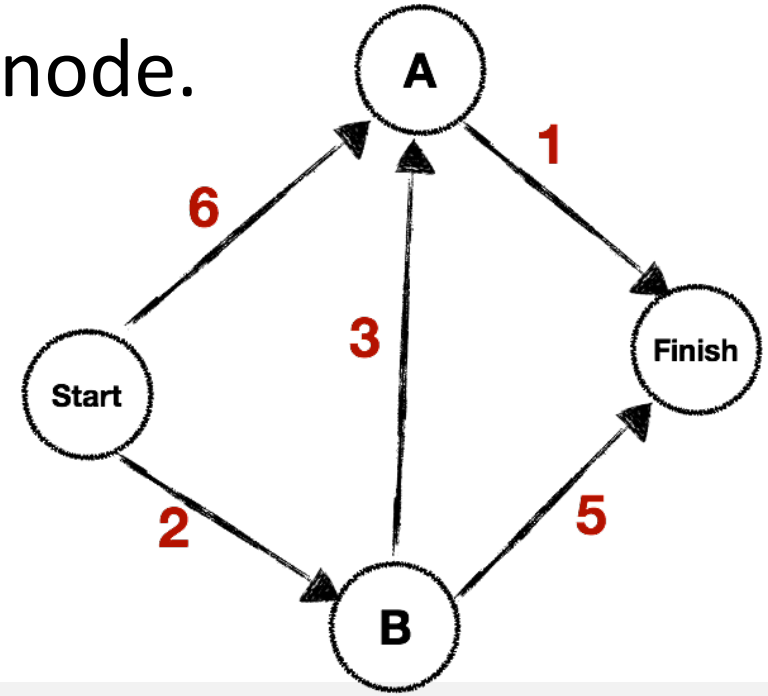


Djikstra's Algorithm (Overview)

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

Djikstra's Algorithm (Overview)

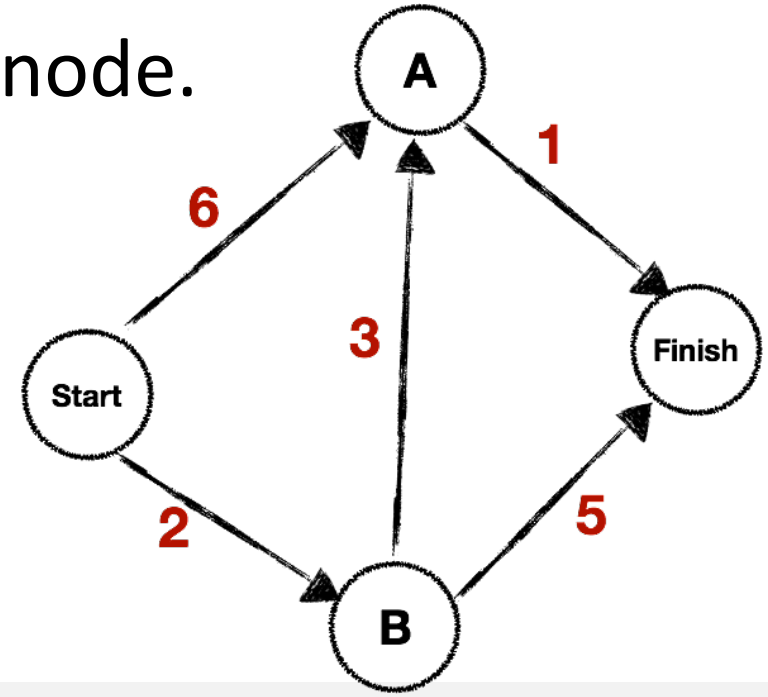
1. Find the "cheapest" node– the node you can get to in the shortest amount of time.
2. Update the costs of the neighbors of this node.
3. Repeat until you've done this for each node.
4. Calculate the final path.



Djikstra's Algorithm (Overview)

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

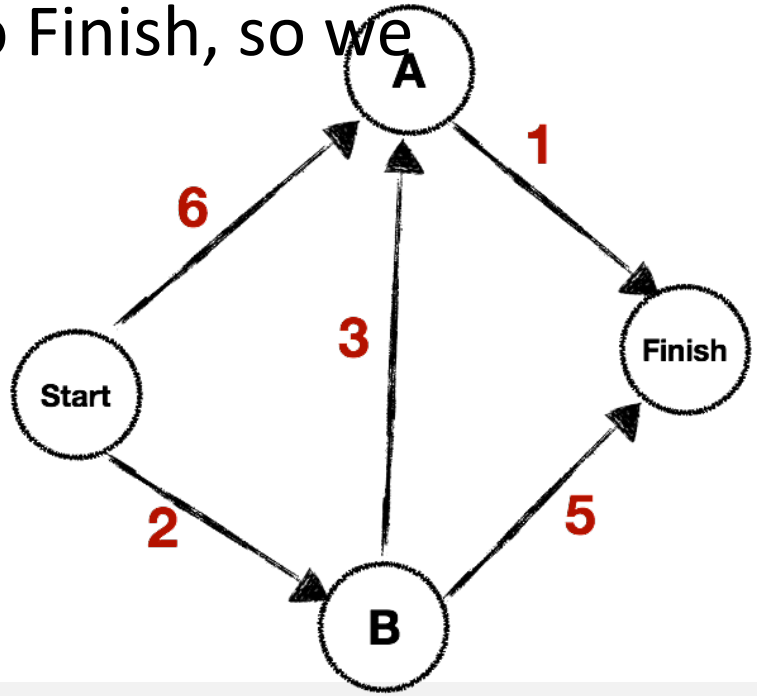
Breadth-search first:
distance = 7



Step 1: Find the cheapest node

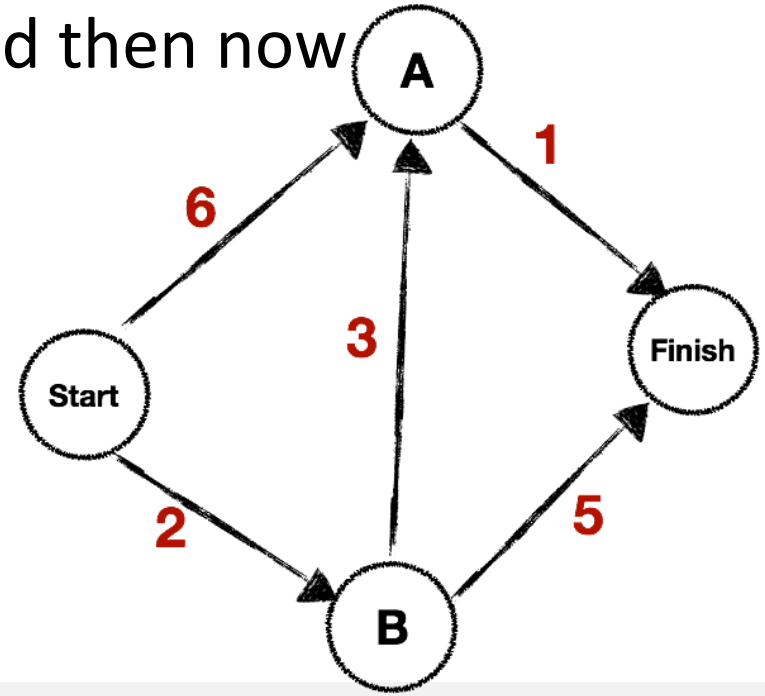
1. Should we go to A or B?
   - Make a table of how long it takes to get to each node from this node.
   - We don't know how long it takes to get to Finish, so we just say infinity for now.

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.

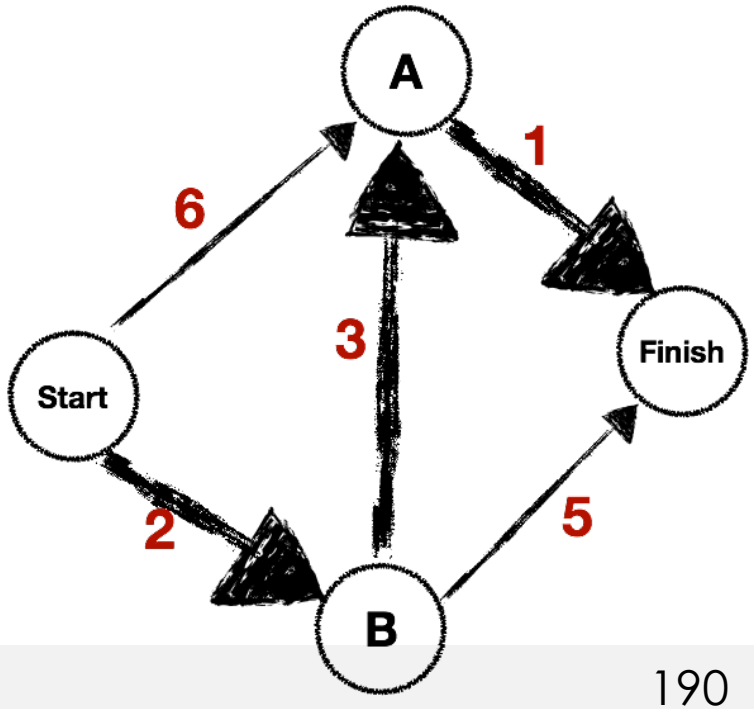


Node	Time to Node
A	~~6~~ 5
B	2
Finish	7

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	~~6~~ 5
B	2
Finish	~~7~~ 6



Djikstra's Algorithm (Review)

1. Find the "cheapest" node– the node you can get to in the shortest amount of time.
2. Update the costs of the neighbors of this node.
    1. Is there a cheaper way to get to the neighbors of this node than we know about so far?
3. Repeat until you've done this for each node.
4. Calculate the final path.
    1. Soon!

Back to the code: How to practice Djikstra?

- Add edge weights/labels to the edges.
- Then…?

Summary

- Trees!
  - Terminology
- Tree Operations:
  - insert() and remove(): Depends on use of tree
  - traverse(): Depth-first, Breadth-first
    - Can easily implement recursively, but can be problematic with large trees
    - Can also implement iteratively, by using either a stack (DFS) or queue (BFS).

Summary, cont.

- Binary Search Tree
  - Inserting nodes such that the left is always smaller than the right
  - Makes it easier to find values in a tree
- Priority Queue ADT
  - Like a queue, but rather than FIFO, always dequeue the highest priority element
- Heap
  - Frequently used to implement Priority Queue
  - Balanced Binary tree
  - Highest priority element is always at the root

Next time:

- Graphs
- In 2 weeks:
  - Final lecture
  - Fun topics
  - Games!

Comparing Data Structures

	Stack	Linked List	Queue	Tree	Graph
Type	Linear	Linear	Linear	Tree/graph	
Insert action	Puts element at top	Inserts element where specified	Puts element at back	…	
Insert operation	push()	insert()	enqueue()	…	
Remove action	Returns element at top	Removes element as specified	Returns element at front	…	
Remove operation	pop()	remove()	dequeue()	…	
Notes	LIFO	Arbitrary length (not constrained by array size)	FIFO	…	

Runtime, Memory

	Stack (Array)	Linked List	Queue (Llist)
Memory	O(k >= n) *[k = max size]*	O(n)	O(n)
Insert	O(1)	O(1) *[insert beginning]* O(n) *[insert elsewhere]*	O(1)
Remove	O(1)	O(1)	O(1)
Find/ Contains	N/A	O(n)	N/A