Trees

Amrita Kaur

August 2, 2023

Contributions made from previous CS106B Instructors

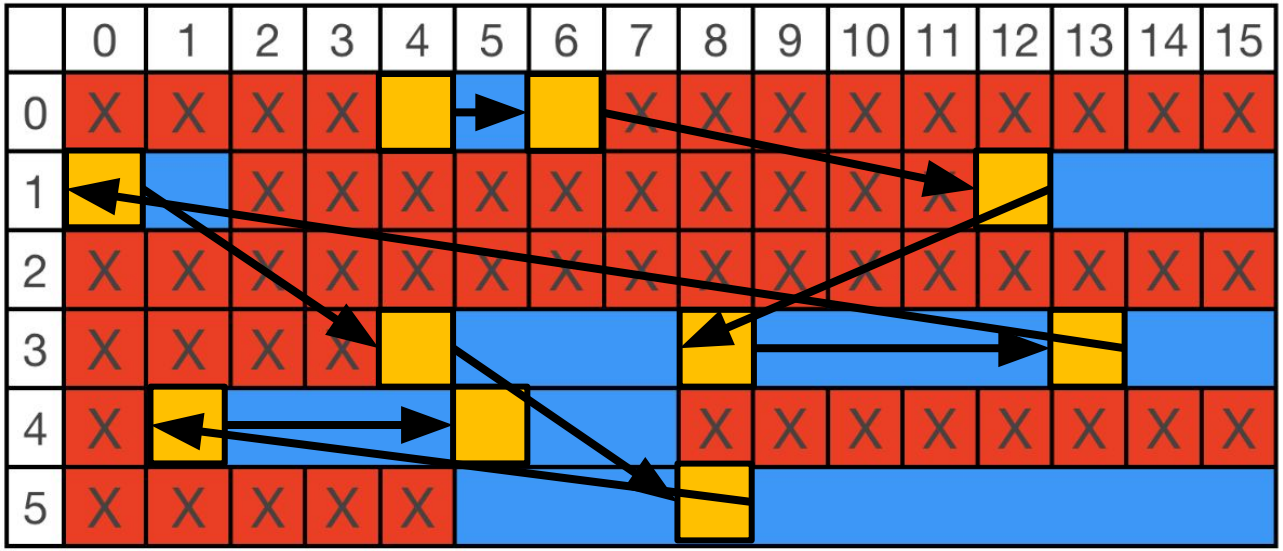
Announcements

- Assignment 4 due tonight at 11:59pm
- Assignment 5 released today
  - YEAH Hours from 3-4pm with Bryant
- Change of grading basis deadline is this Friday at 5pm PT
  - Come chat with us (or check out this resource) if you're considering whether to take for letter grade or credit/no credit

Recap: LinkedLists

What are Linked Lists?

- A way we can use pointers to organize non-contiguous memory on the heap



Redefining Linked Lists

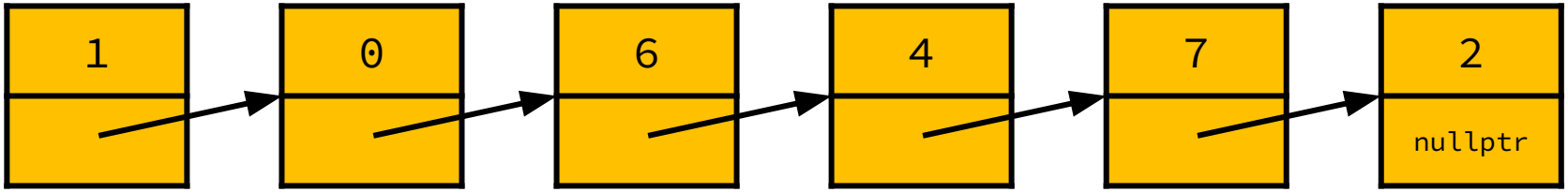
A **linked list** is either:

An empty list (`nullptr`)

Or a single node that points to another **linked list**

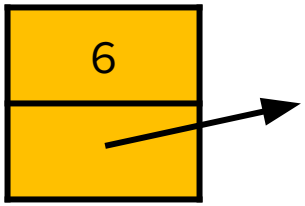
Benefits of Linked Lists

- Easily resizable
- Efficient to insert elements at the beginning



Linked Lists, Structurally

- A linked list is a chain of nodes
- Each node is a struct that contains:
  - A piece of data (like an int, or string)
  - A pointer to the next node



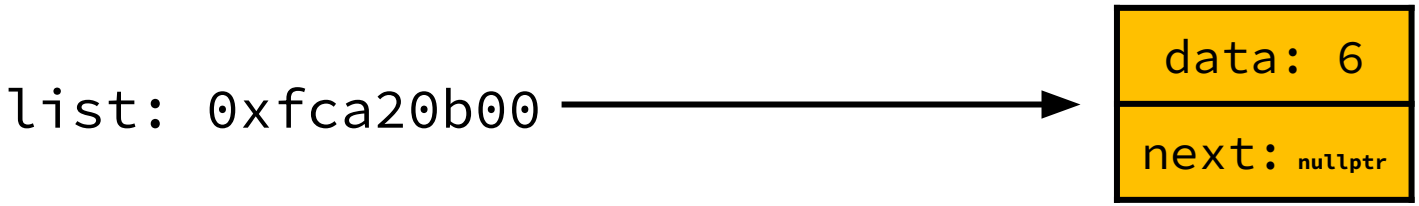

```
struct Node {
   int data;
   Node* next;
};
```

Creating a Linked List

- Create a new Node on the heap and store a pointer to it

```
Node* list = new Node;
list->data = 6;
list->next = nullptr;
```

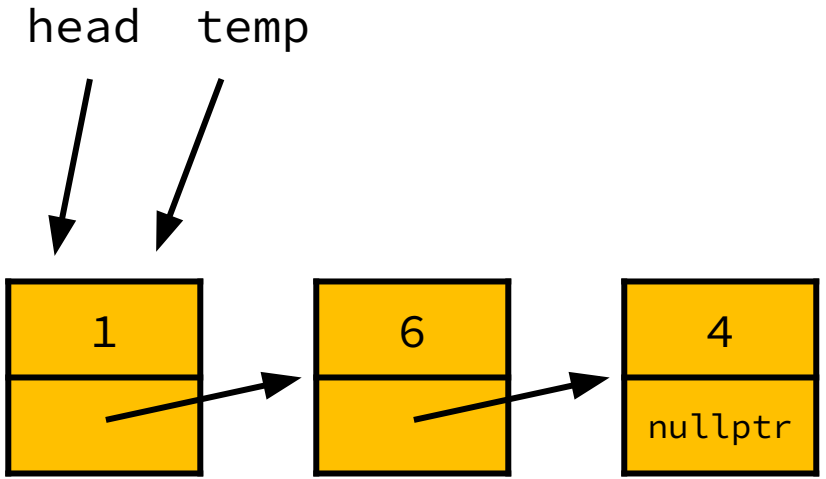
Dereference AND access the field for struct pointers using ->



Lives at `0xfca20b00` on the heap

Freeing a Linked List

```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```



Linked Lists vs. Arrays, Big-O

Linked Lists

- Prepend - O(1)
- Append - O(n)
- Insert - O(n)
- Delete - O(n)
- Traverse - O(n)

Arrays

- Prepend - O(n)
- Append - O(1)
- Insert - O(n)
- Delete - O(n)
- Traverse - O(n)

Passing Pointers by Value

- Unless specified otherwise, parameters in C++ are passed by value – this includes pointers!
- When passed by value, callee function gets a copy of the pointer; **it cannot change where the original pointer points**

Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

head: nullptr

Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

head: nullptr

Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

```
head: nullptr
data: 5
```

```
head: nullptr
```

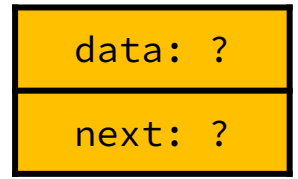
Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

head: nullptr
data: 5
newNode:



head: nullptr

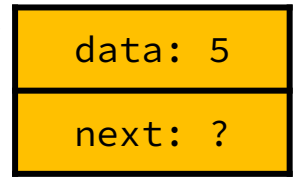
Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

head: nullptr
data: 5
newNode:



head: nullptr

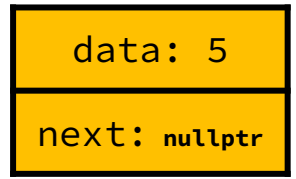
Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

head: nullptr
data: 5
newNode:



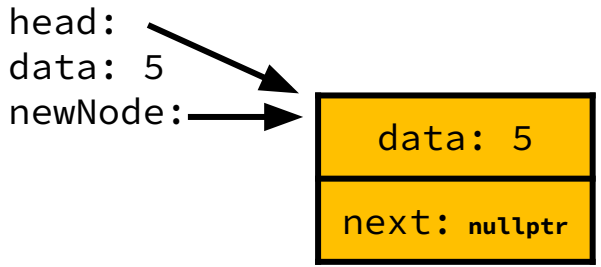
head: nullptr

Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```



head: nullptr

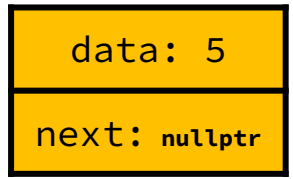
Note: this was a copy of the original head, so head from main doesn't get changed!

Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```



head: nullptr

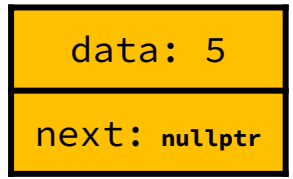
Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

MEMORY LEAK 👎



head: nullptr

Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

head: nullptr

Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

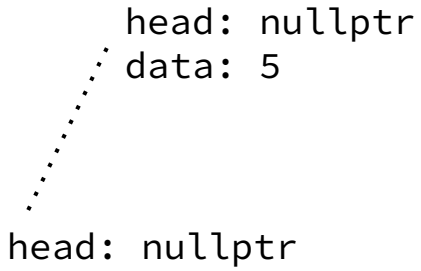
head: nullptr

Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```



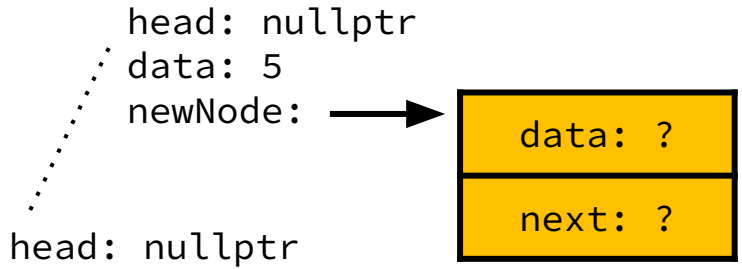
Note: we didn't make a copy of `head`, `prependTo` gets access to the `head` variable from back in `main`!

Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

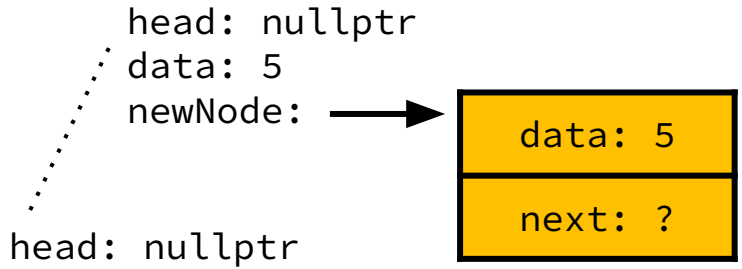


Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

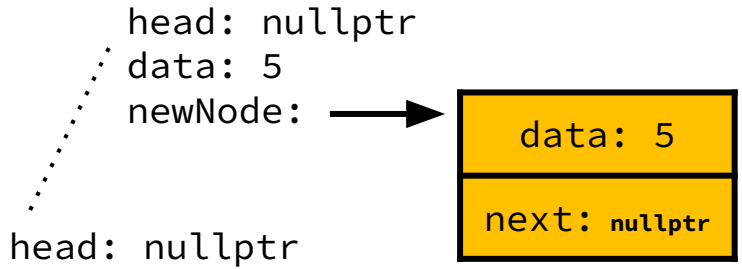


Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

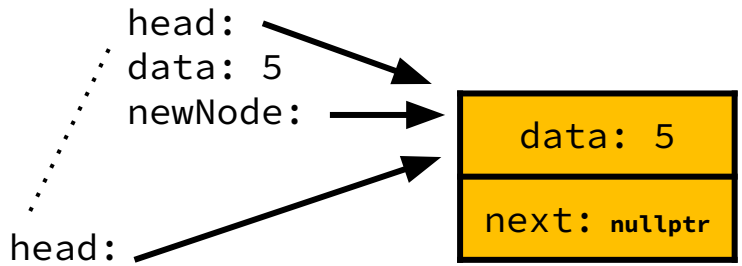


Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

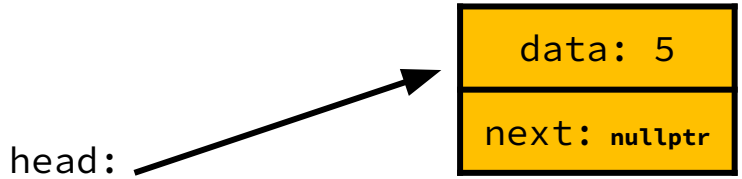


Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```



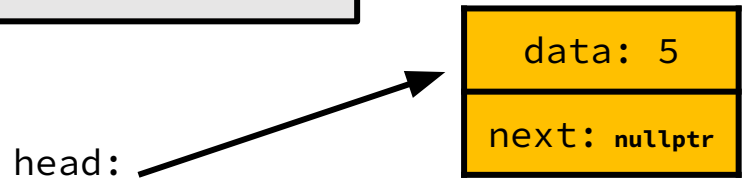
Passing Pointers by Reference

- When passed by reference, the callee function can change where the origina

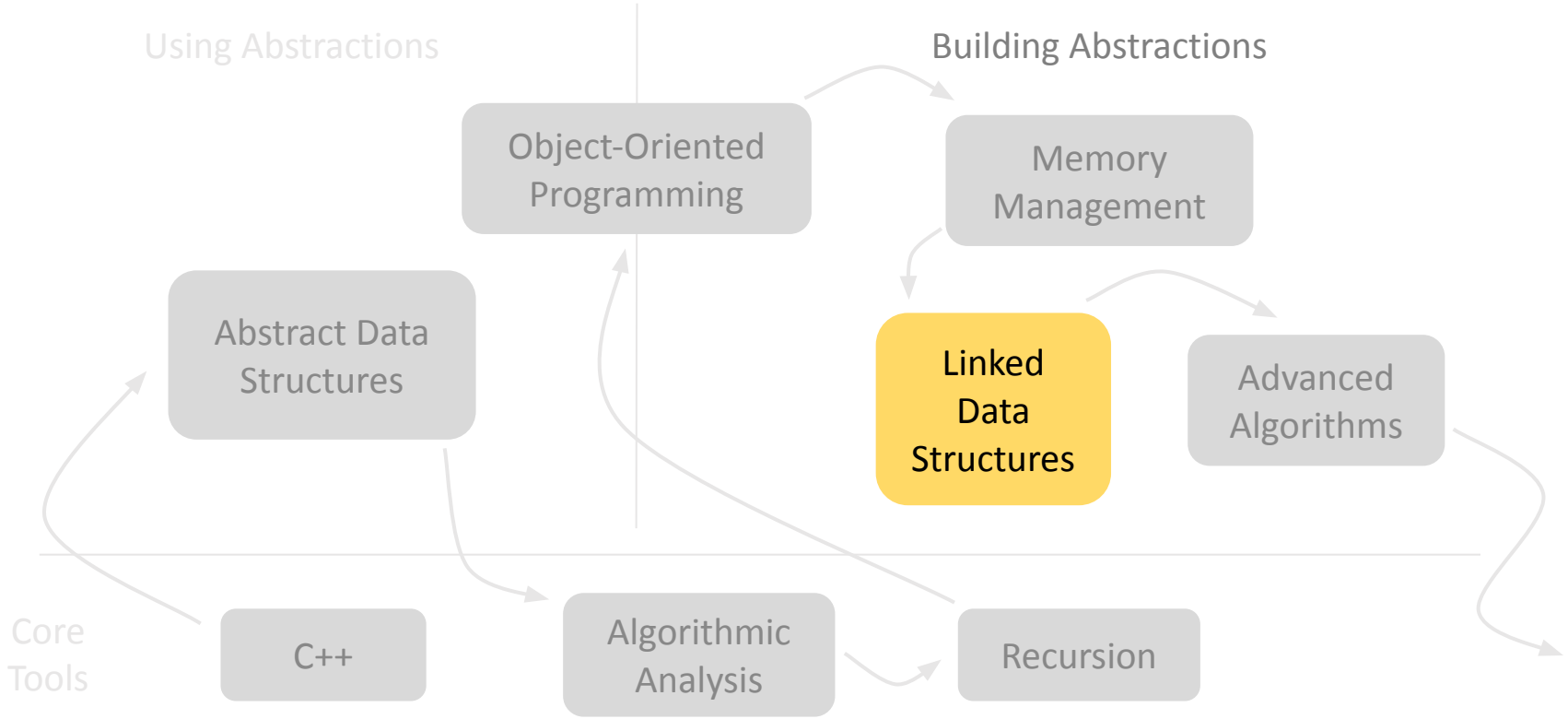
```
void pre
    Node
    newN
    newN
    head
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

When you want a helper function to modify the address a pointer points to, you should pass it by reference.

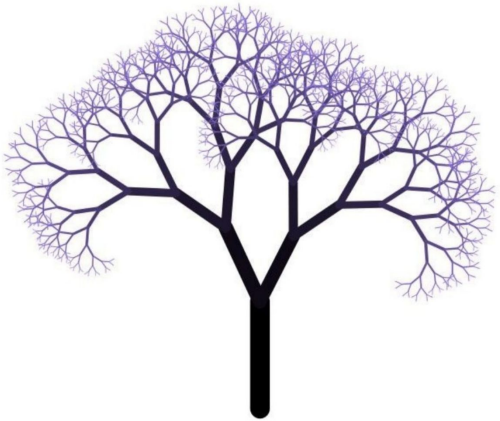


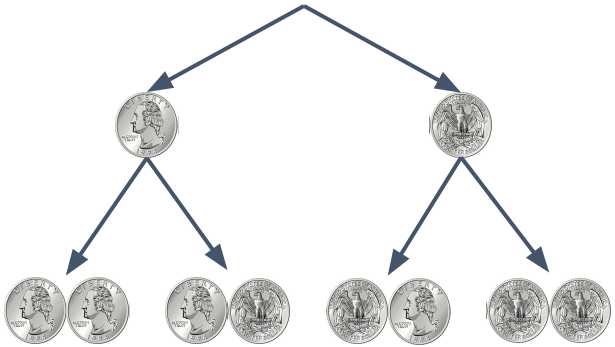
Roadmap

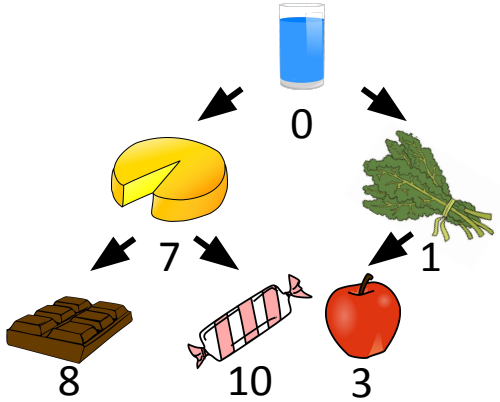

Using Abstractions
Building Abstractions
Object-Oriented Programming
Memory Management
Abstract Data Structures
Linked Data Structures
Advanced Algorithms
Core Tools
C++
Algorithmic Analysis
Recursion

Throwback

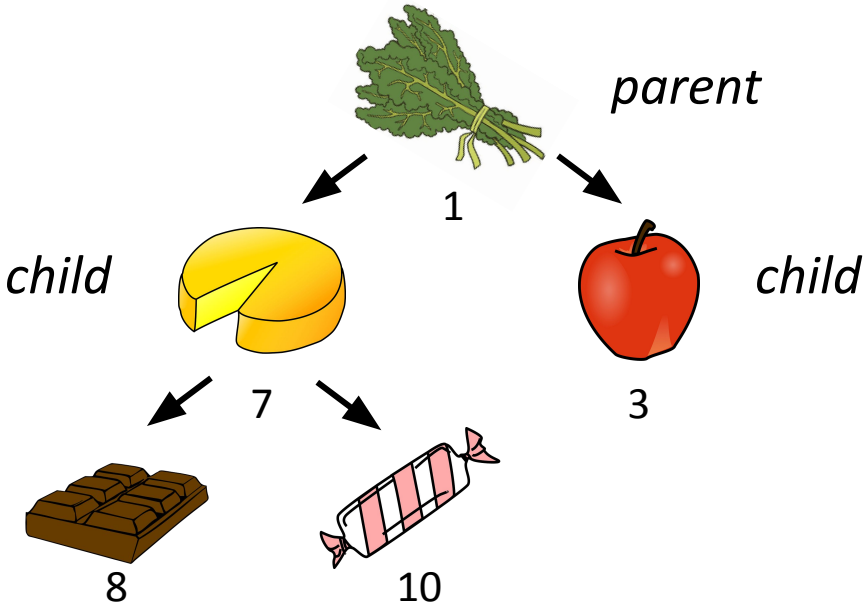
- We've already seen trees before in this class



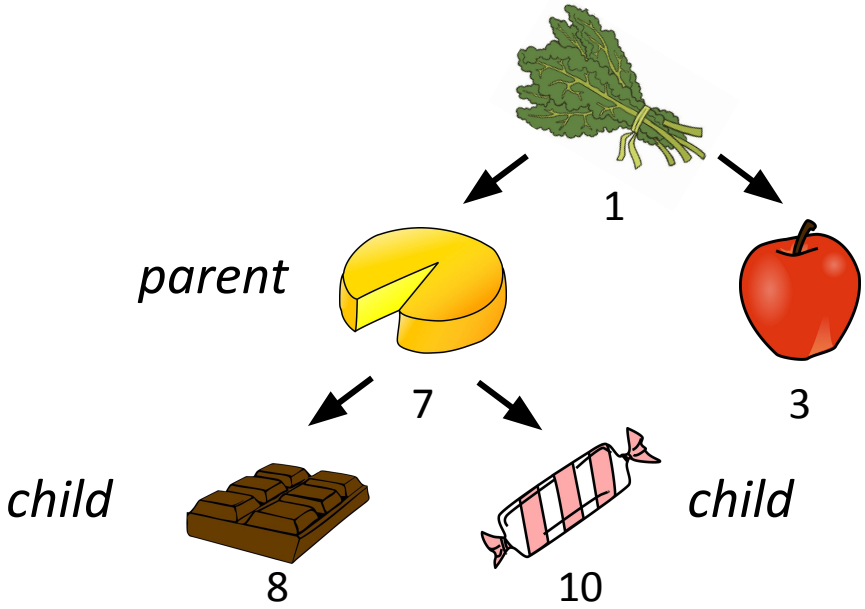




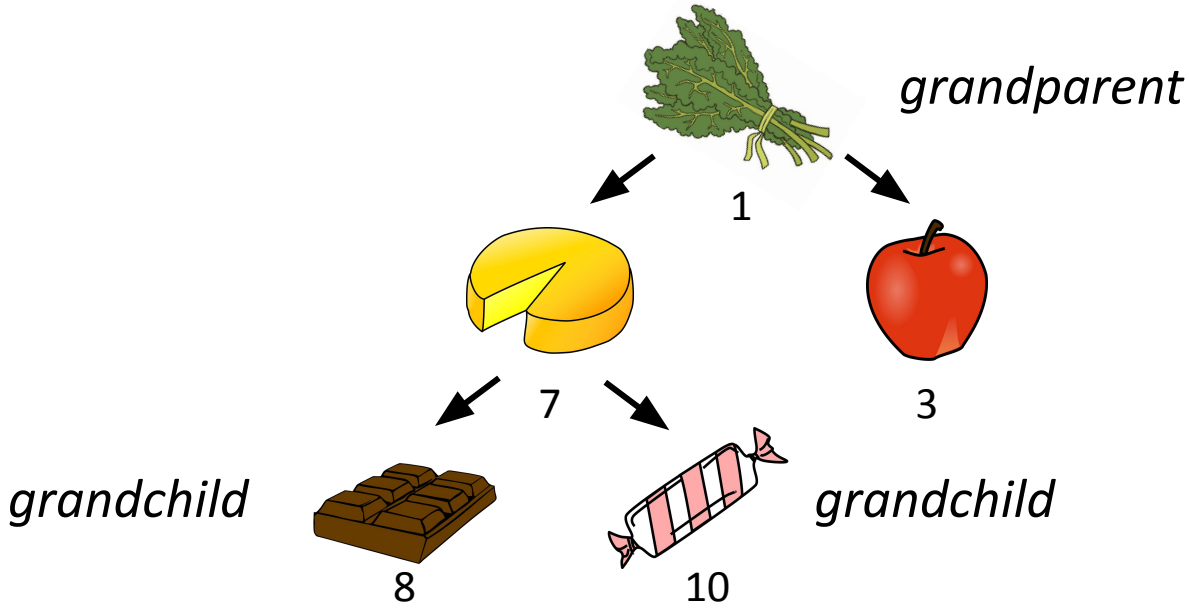
Terminology Recap



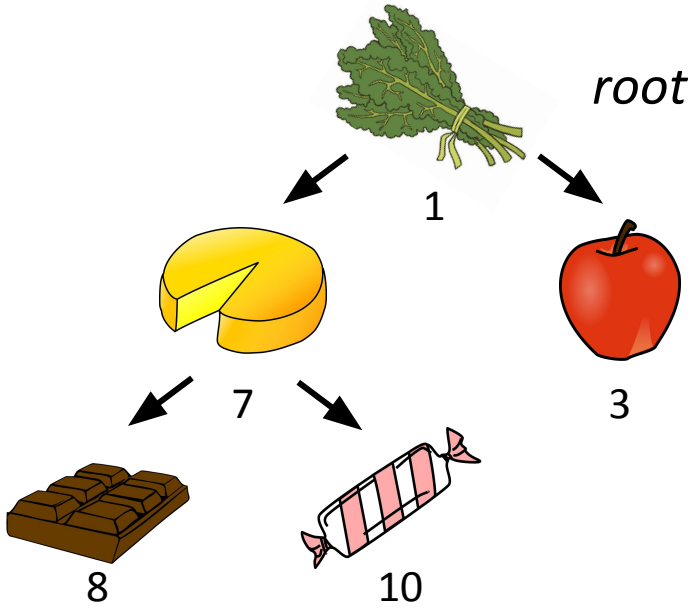
Terminology Recap



Terminology Recap

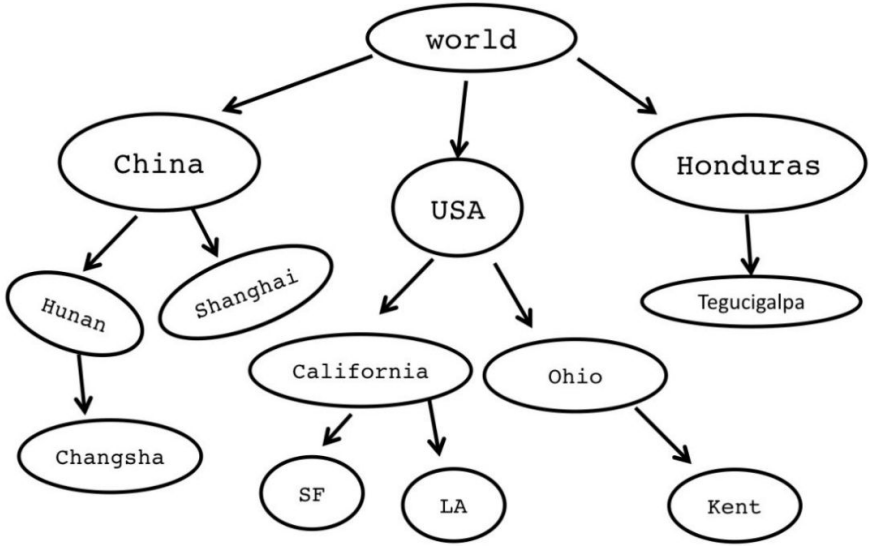


Terminology Recap



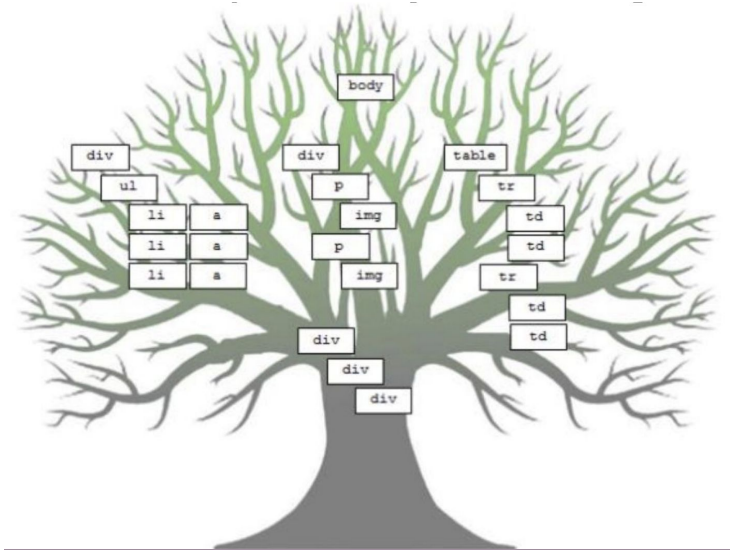
Uses

- Trees are useful in other ways besides visualizing recursion and modeling priority
  - Describe hierarchies



Uses

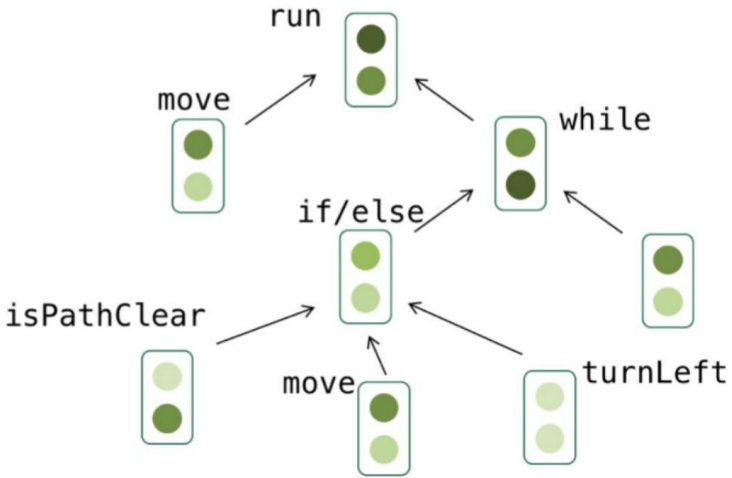
- Trees are useful in other ways besides visualizing recursion and modeling priority
  - Describe hierarchies
  - Model the structure of websites



Uses

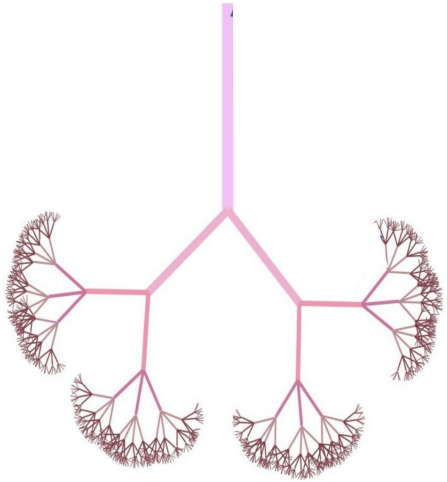
- Trees are useful in other ways besides visualizing recursion and modeling priority
  - Describe hierarchies
  - Model the structure of websites
  - Describe syntax structure of programs

```
def run() {
  move();
  while (notFinished()) {
    if (isPathClear()) {
      move();
    } else {
      turnLeft();
    }
    move();
  }
}
```



Uses

- Trees are useful in other ways besides visualizing recursion and modeling priority
  - Describe hierarchies
  - Model the structure of websites
  - Describe syntax structure of programs
- Distance from each element to the top of the structure is small, even if there are many elements



Uses

- Trees are useful in other ways besides visualizing recursion and modeling priority
  - Describe hierarchies
  - Model the structure of websites
  - Describe syntax structure of programs
- Distance from each element to the top of the structure is small, even if there are many elements
- Really good for working with recursive problems, because trees are inherently defined recursively!

What is a tree?

tree

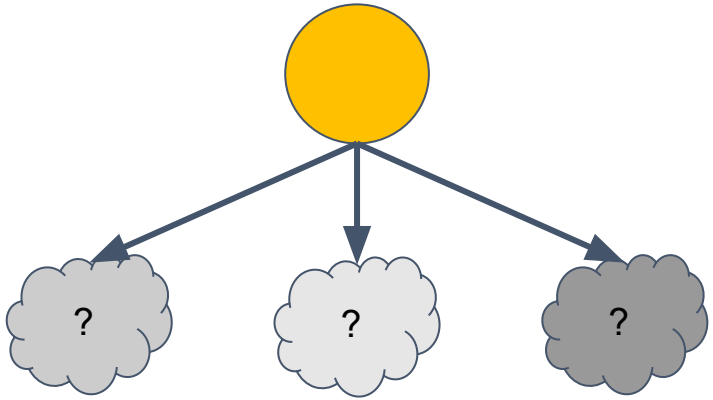
a hierarchical data organization structure composed of a root value linked to zero or more non-empty subtrees

What is a tree?

A tree is either:

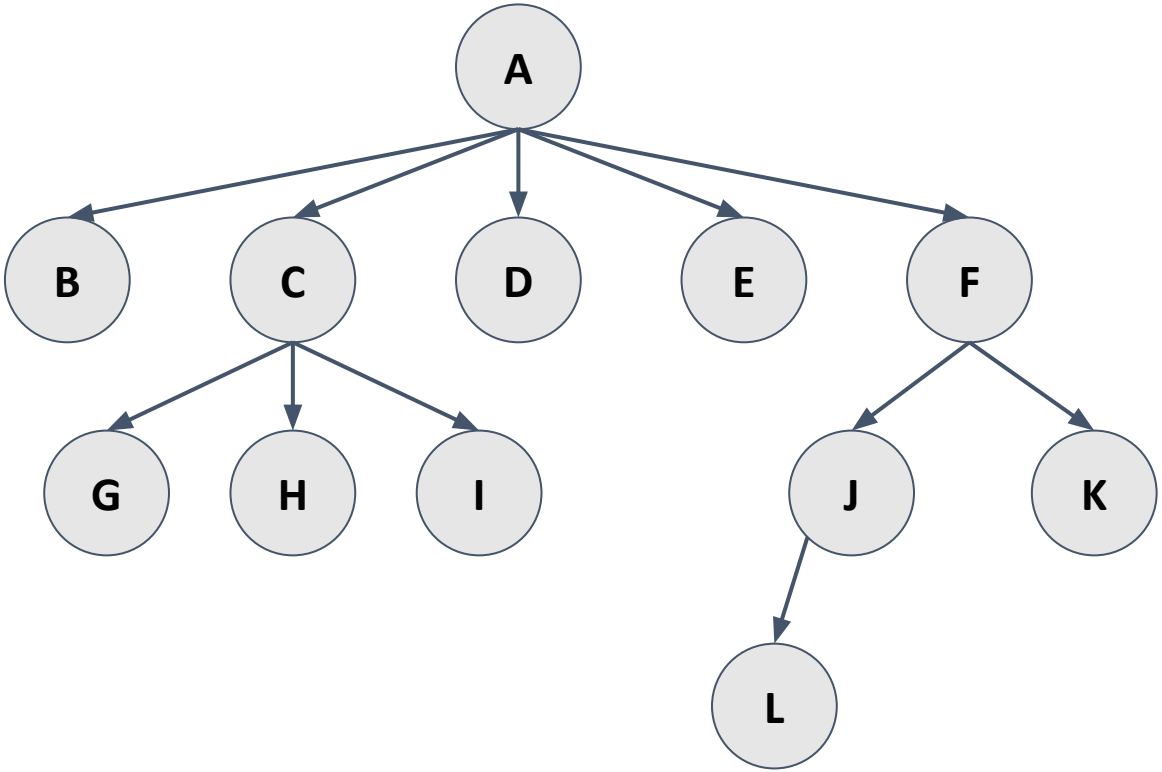
- An empty data structure, or

- A single node with zero or more non-empty subtrees

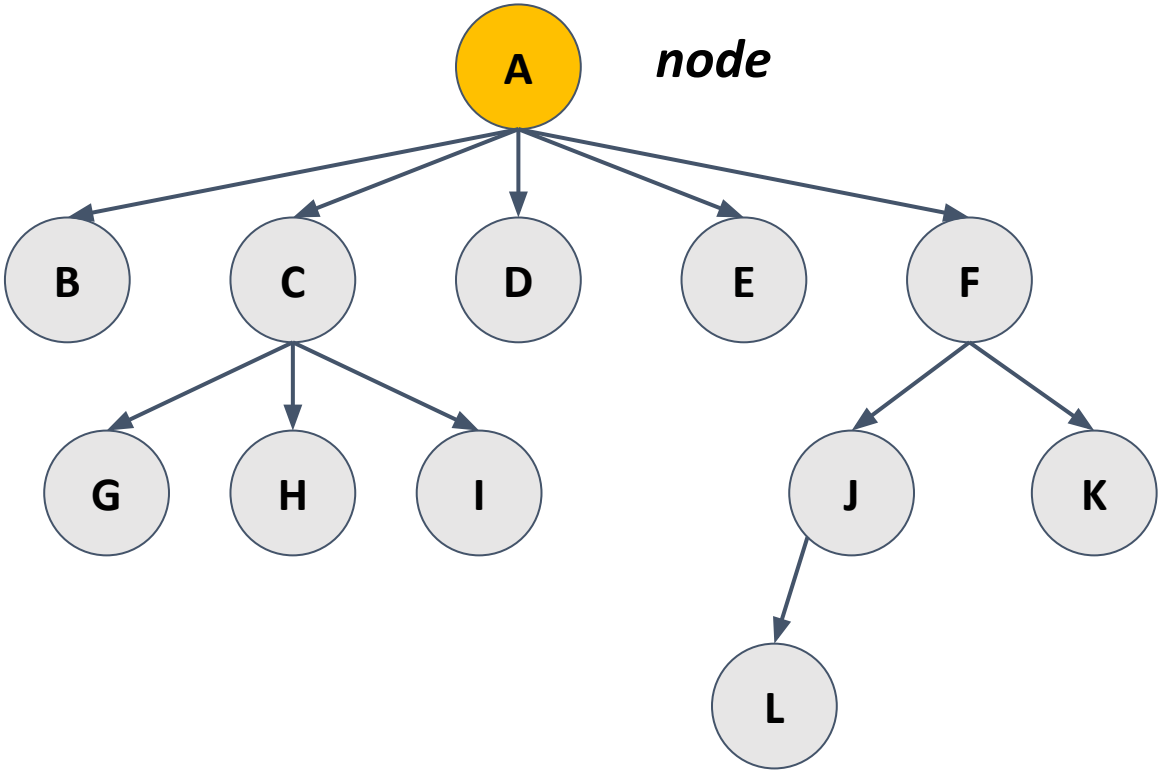


New Tree Terminology

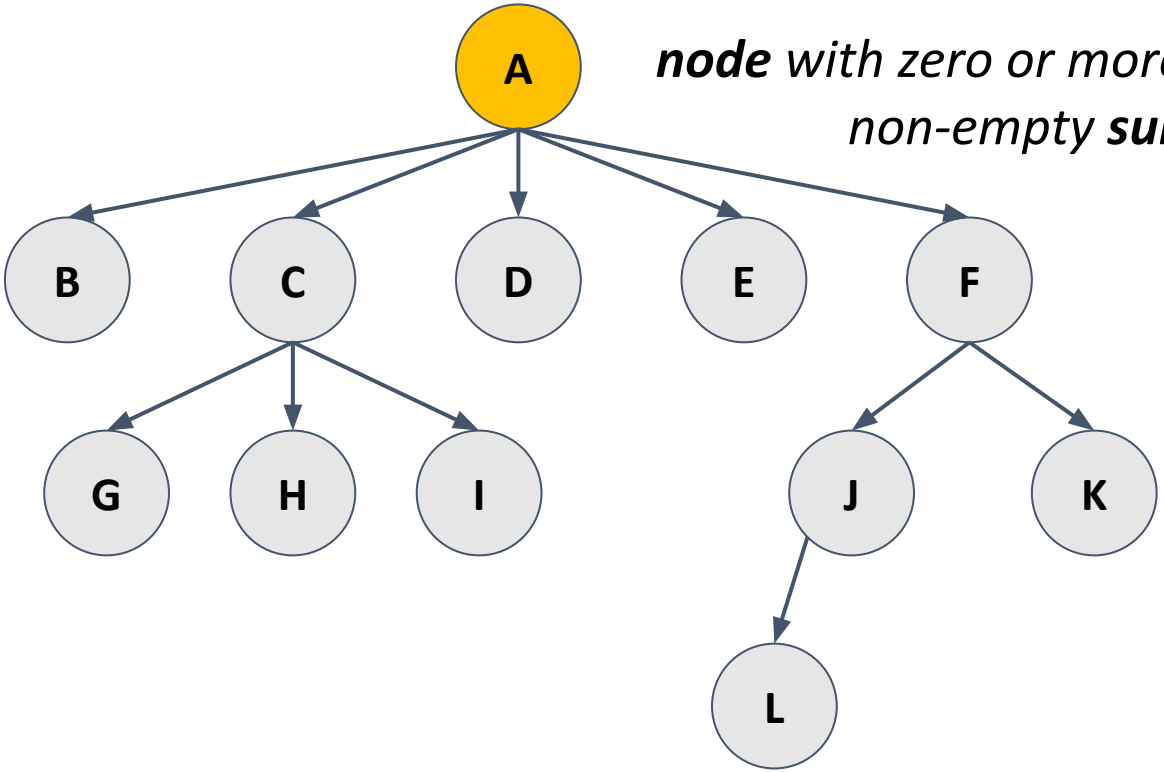
New Tree Terminology



New Tree Terminology

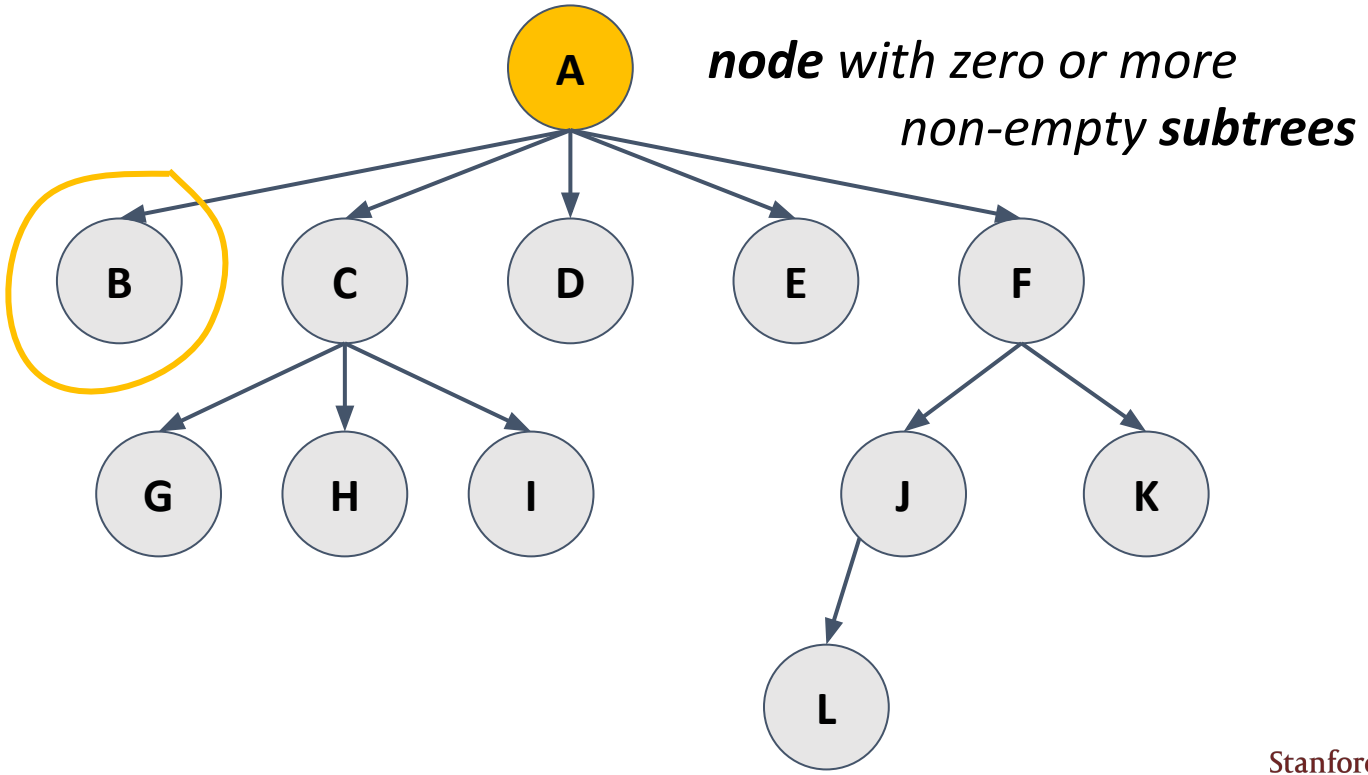


New Tree Terminology



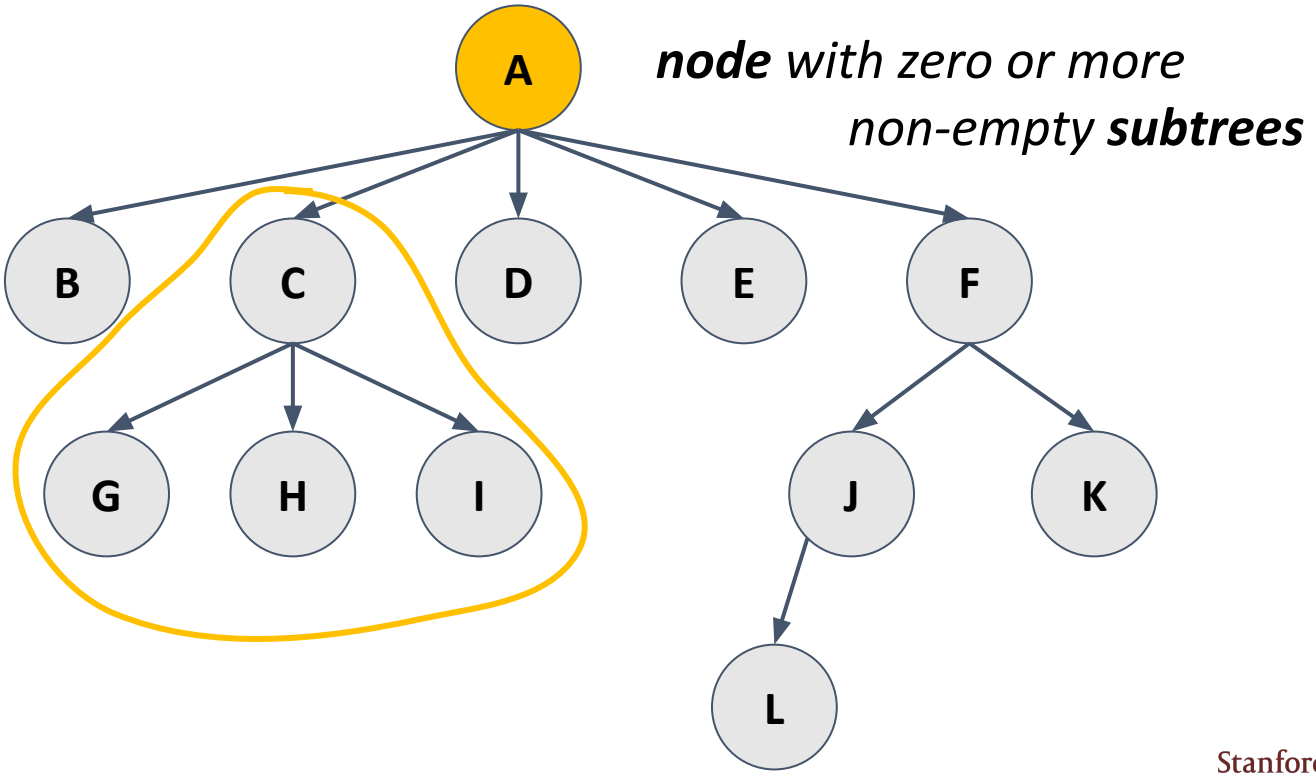
node *with zero or more non-empty* ***subtrees***

New Tree Terminology



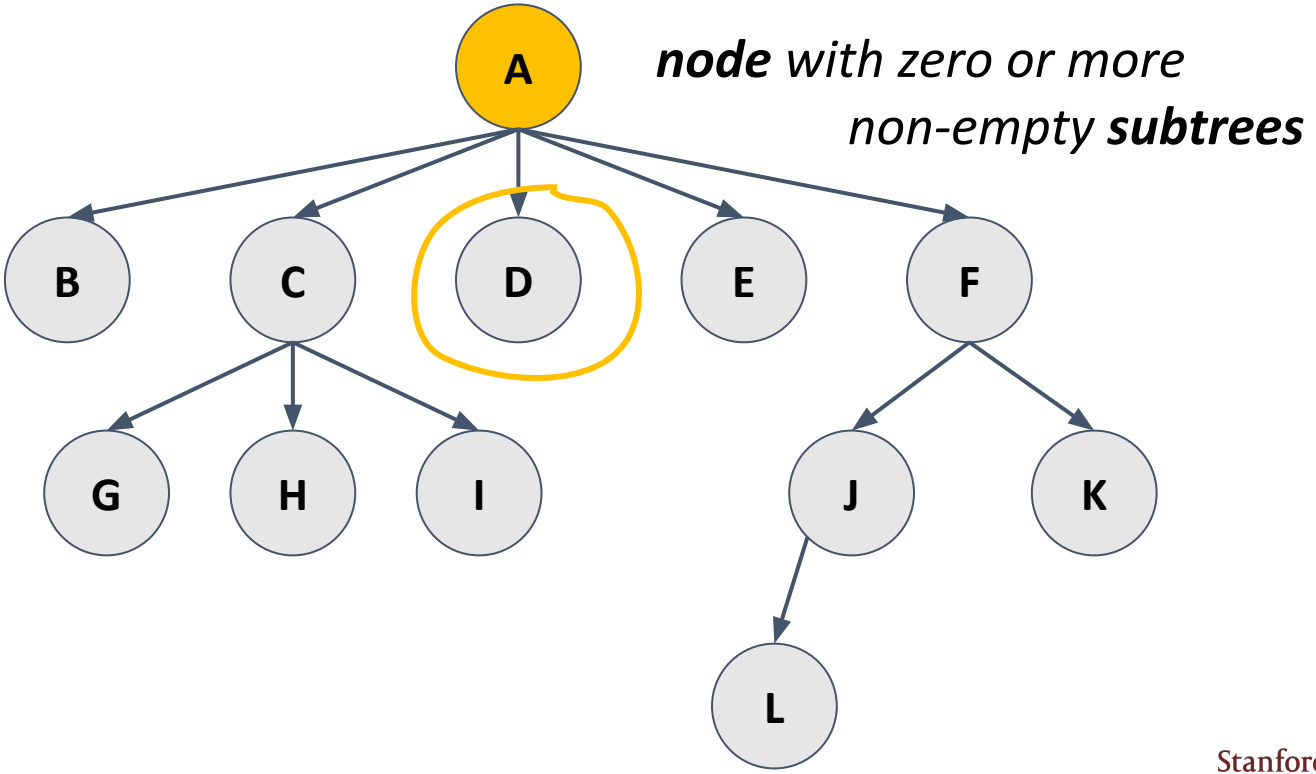
node *with zero or more non-empty* ***subtrees***

New Tree Terminology

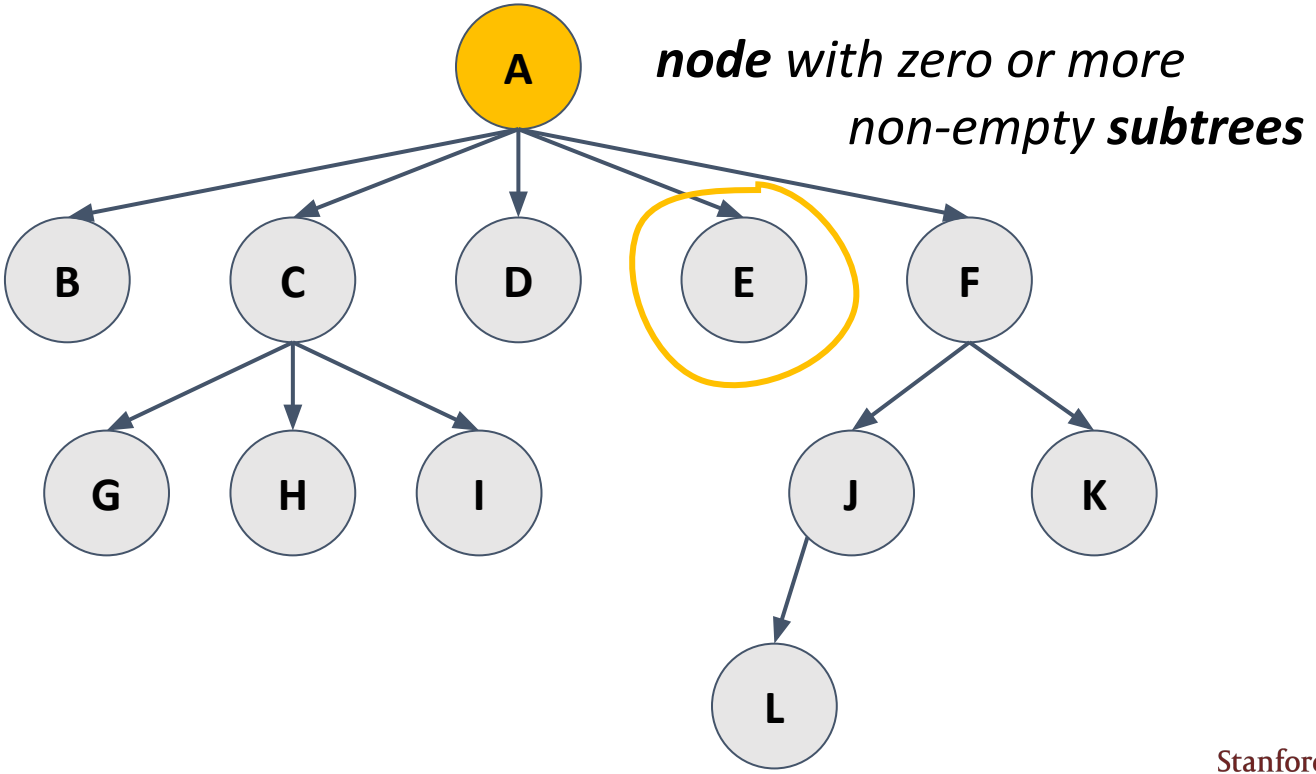


node *with zero or more non-empty* ***subtrees***

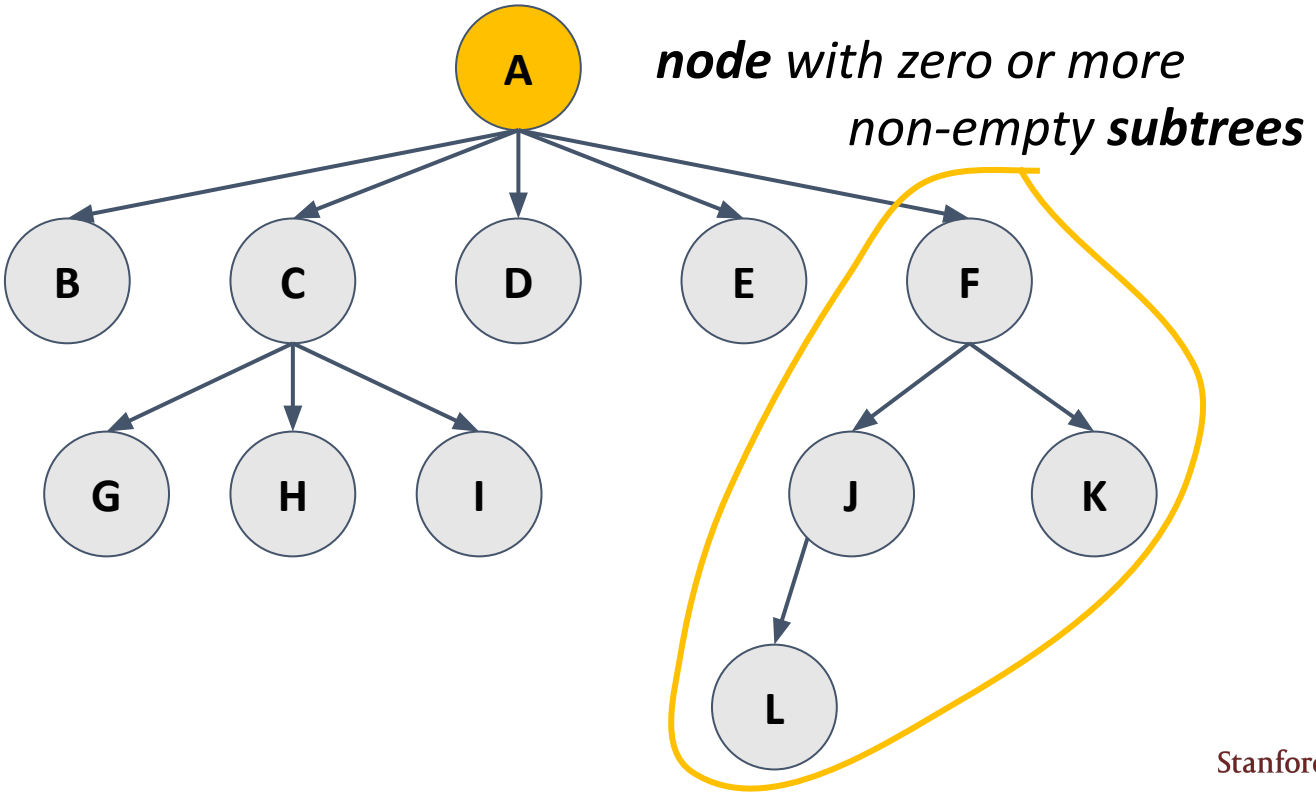
New Tree Terminology



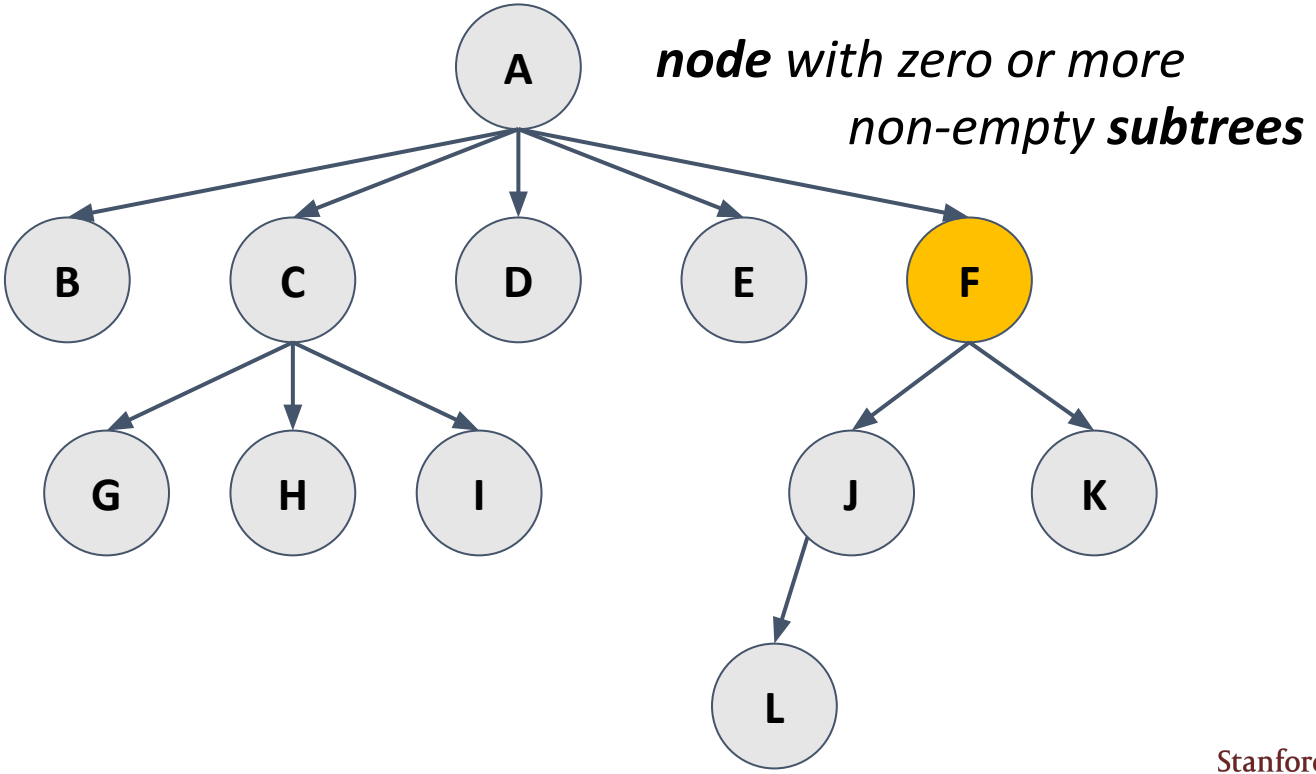
New Tree Terminology



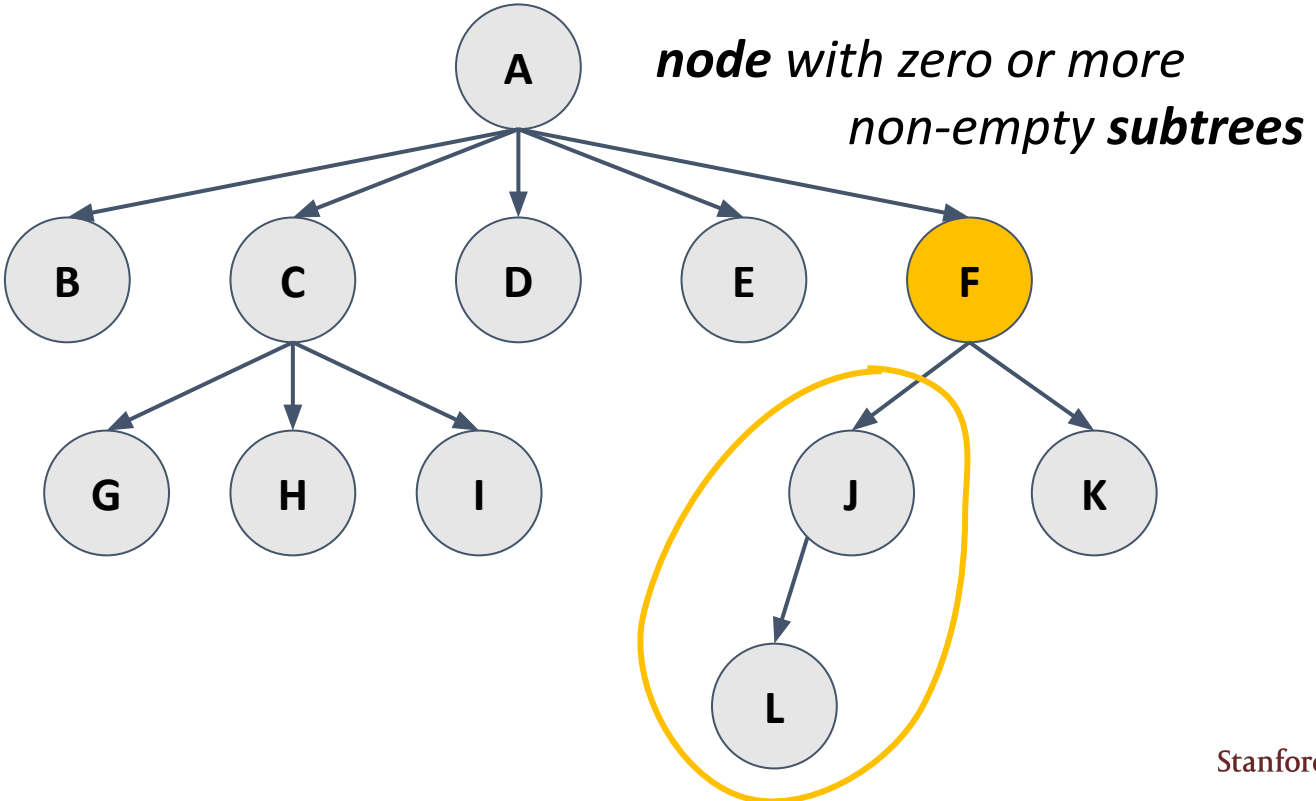
New Tree Terminology



New Tree Terminology

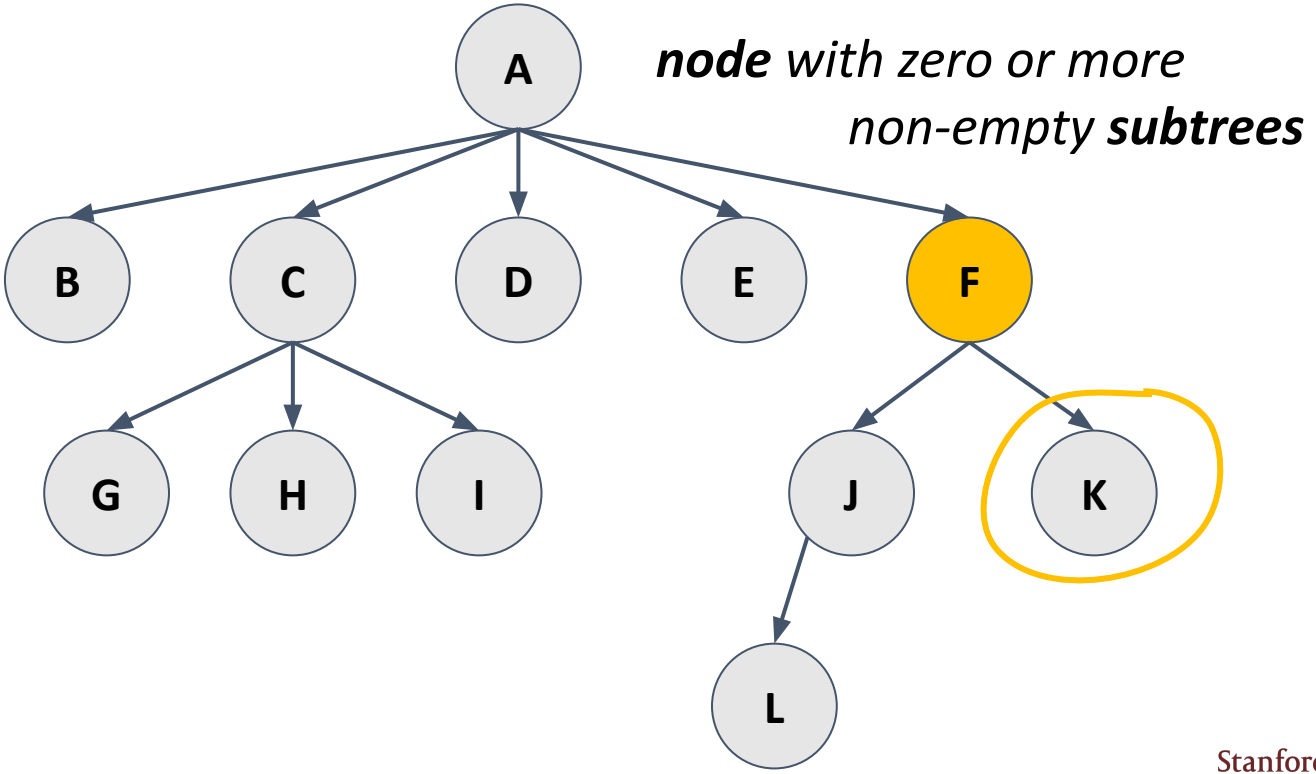


New Tree Terminology

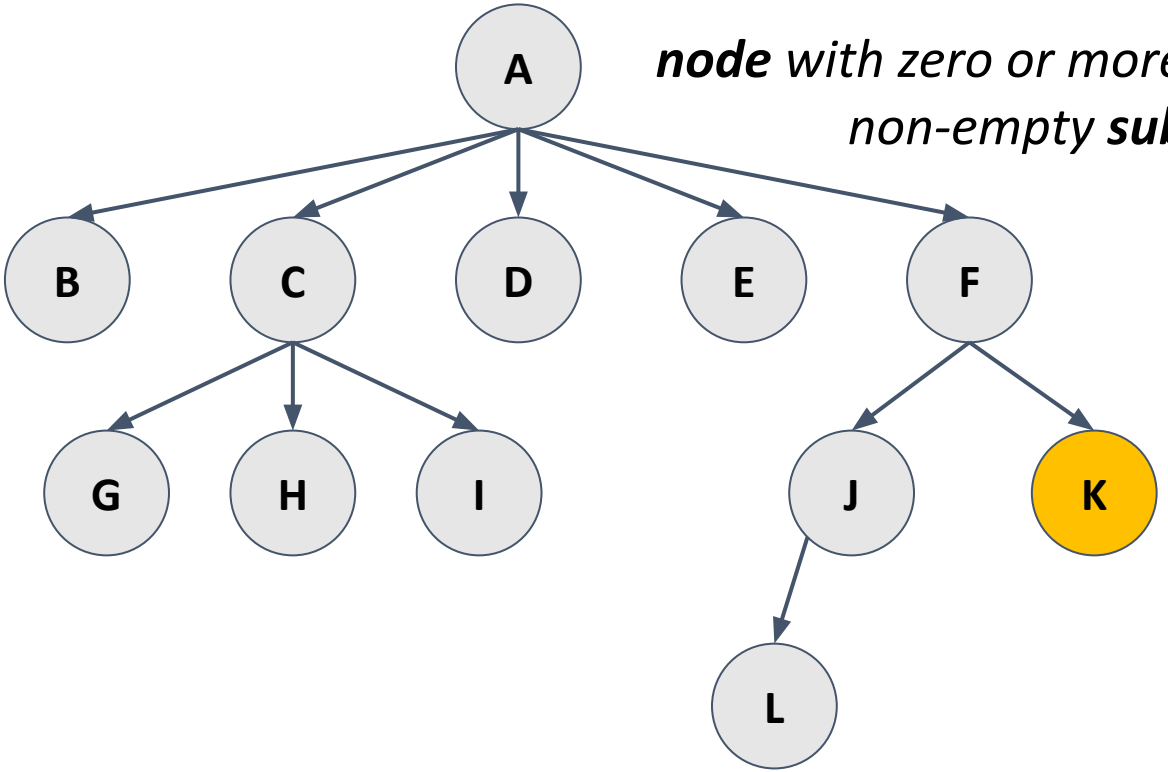


node *with zero or more non-empty* ***subtrees***

New Tree Terminology

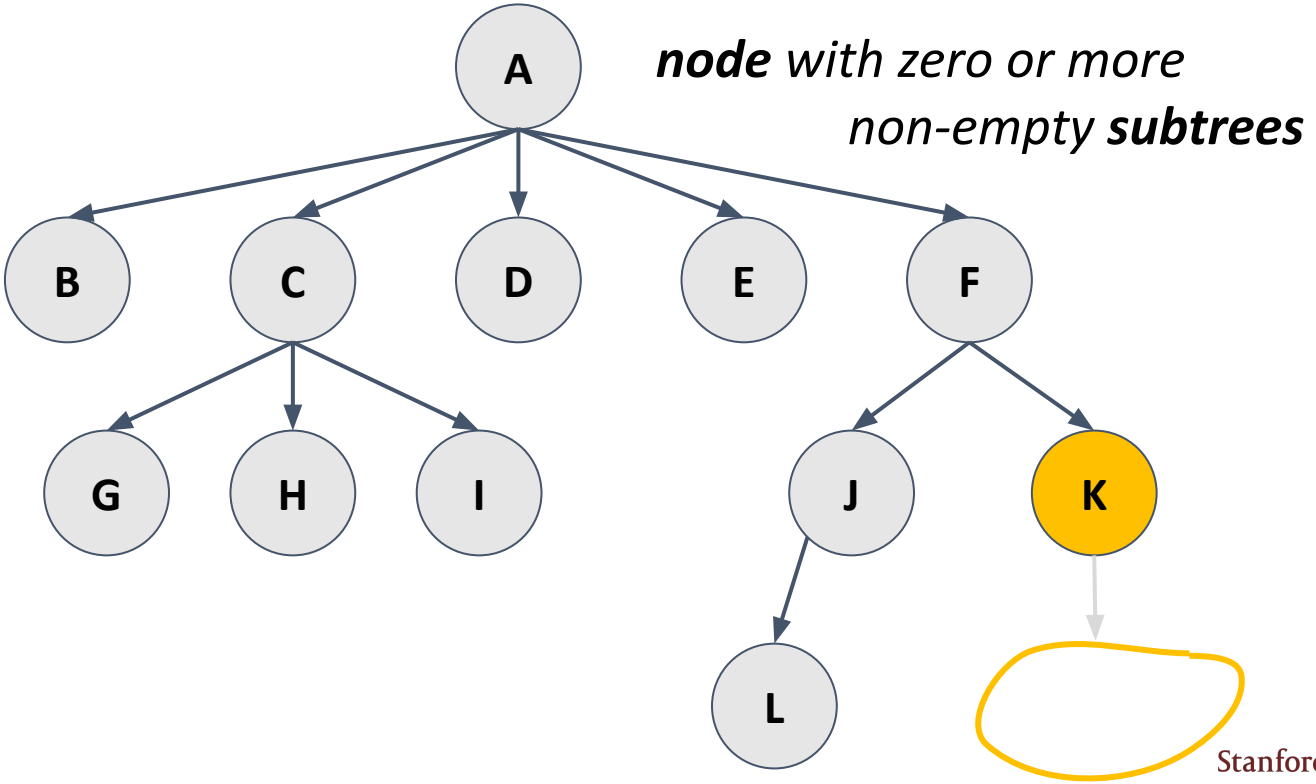


New Tree Terminology

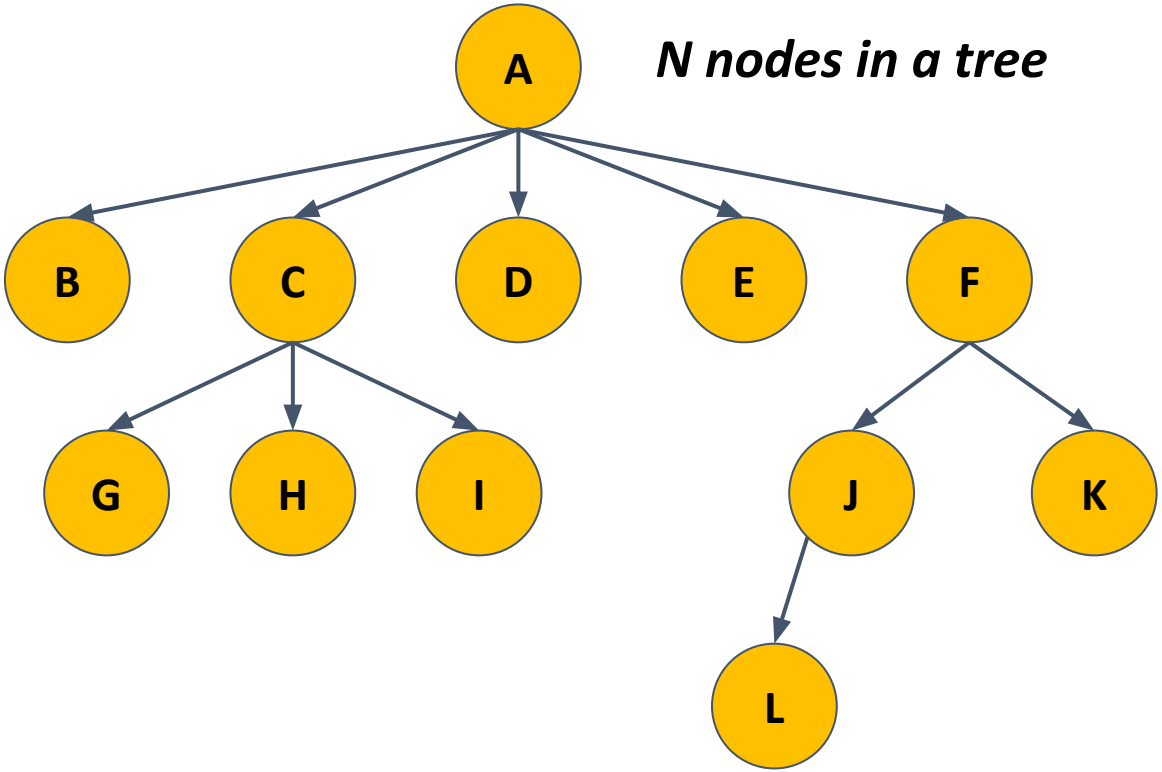


node *with zero or more non-empty* ***subtrees***

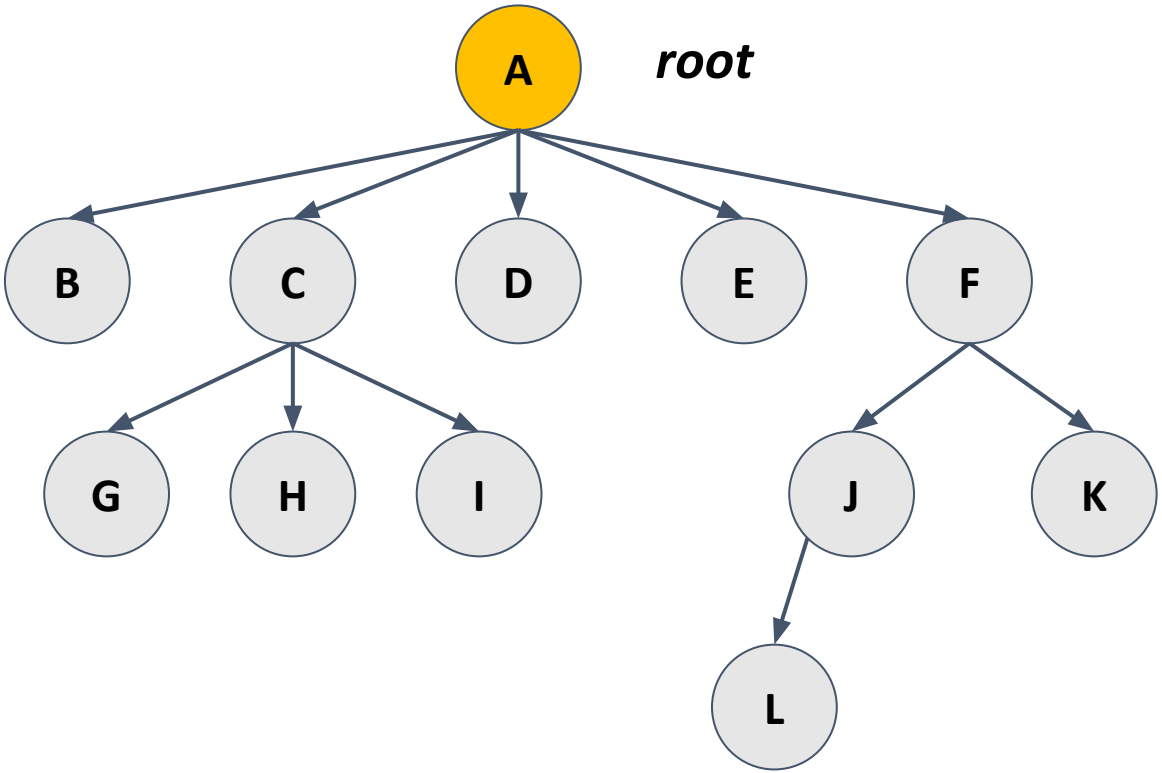
New Tree Terminology



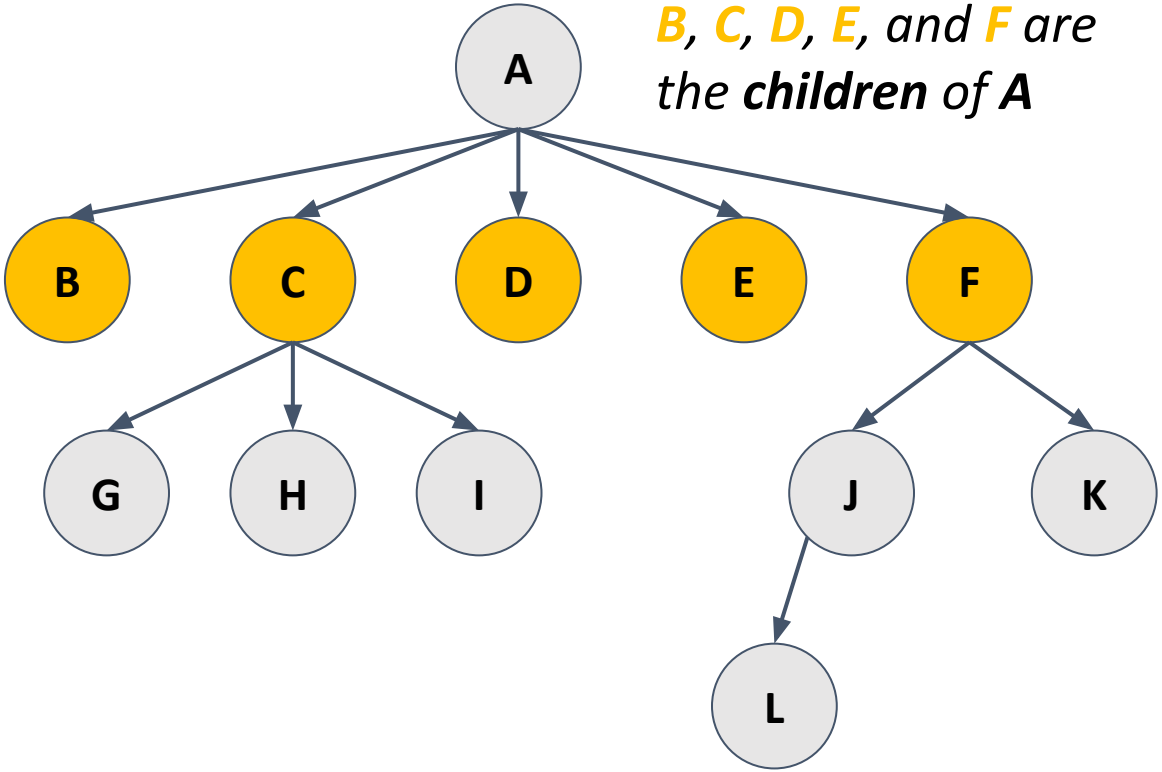
New Tree Terminology



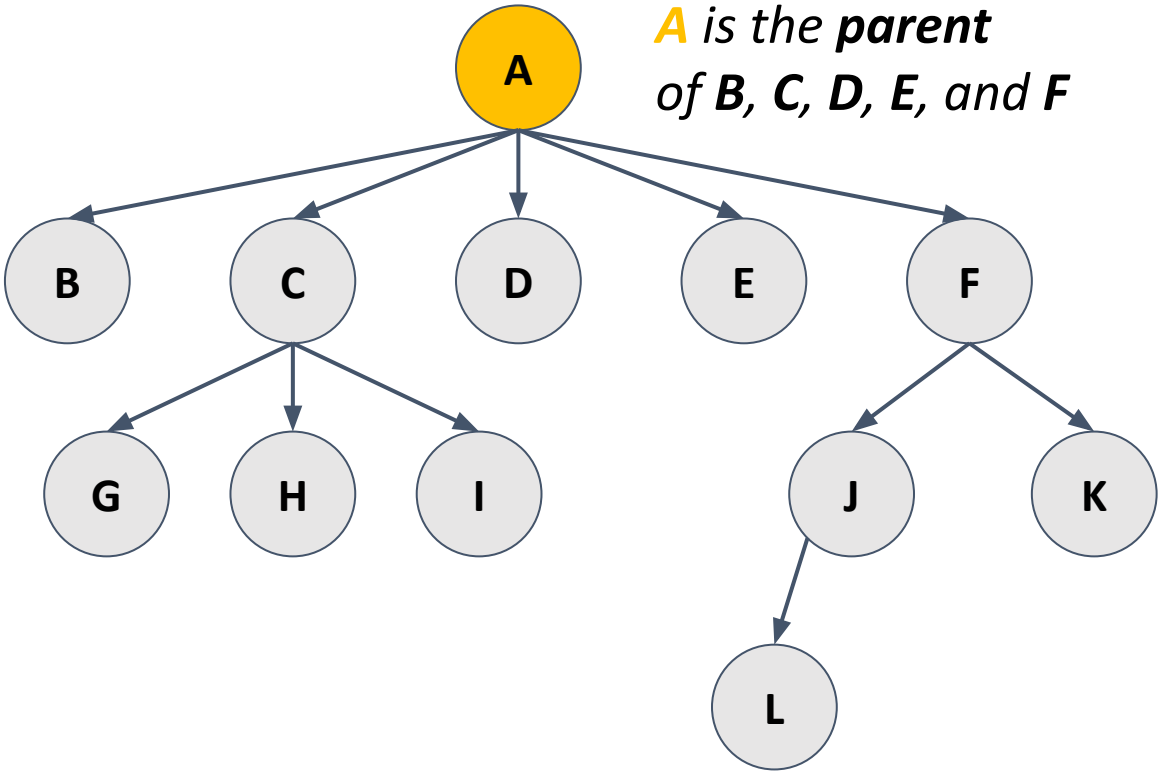
New Tree Terminology



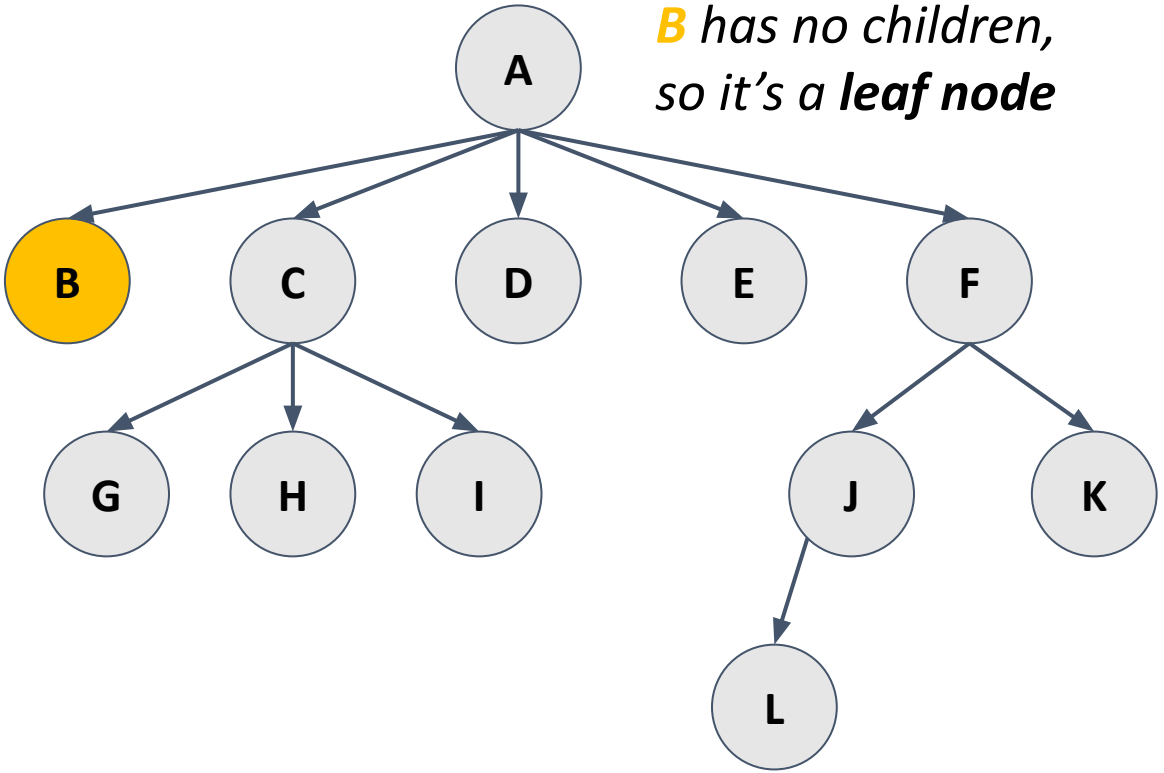
New Tree Terminology



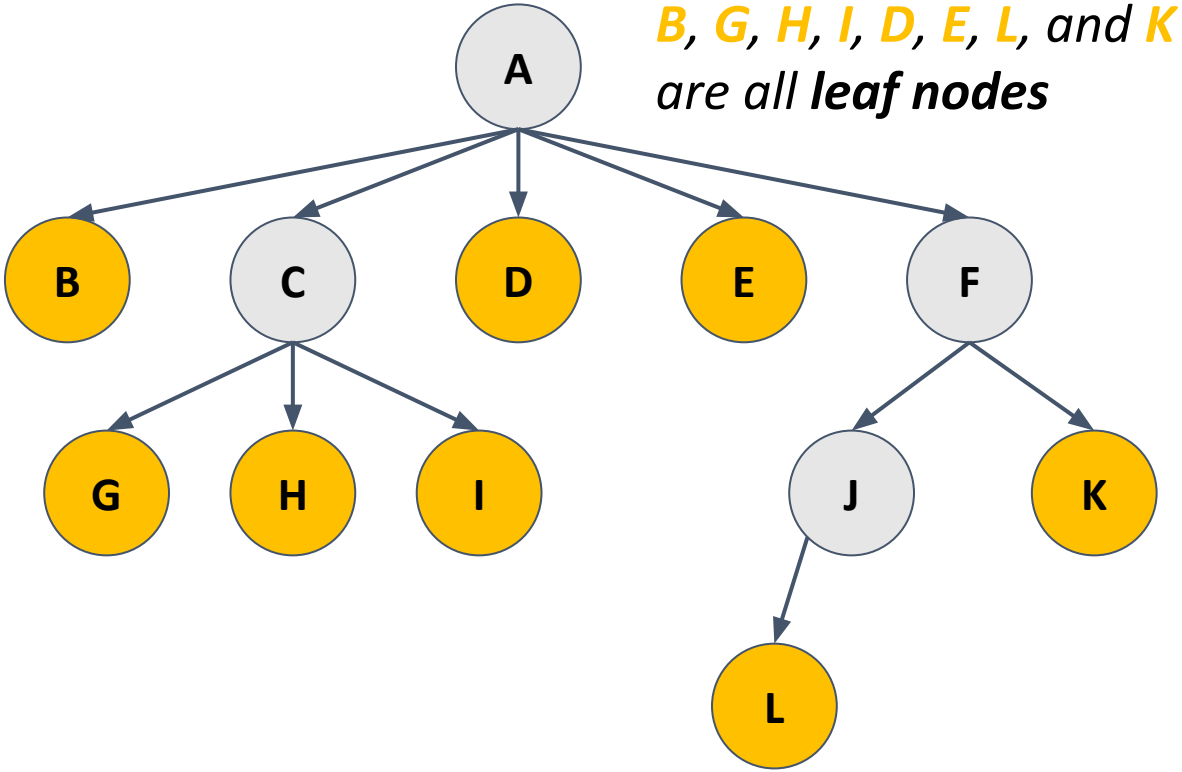
New Tree Terminology



New Tree Terminology

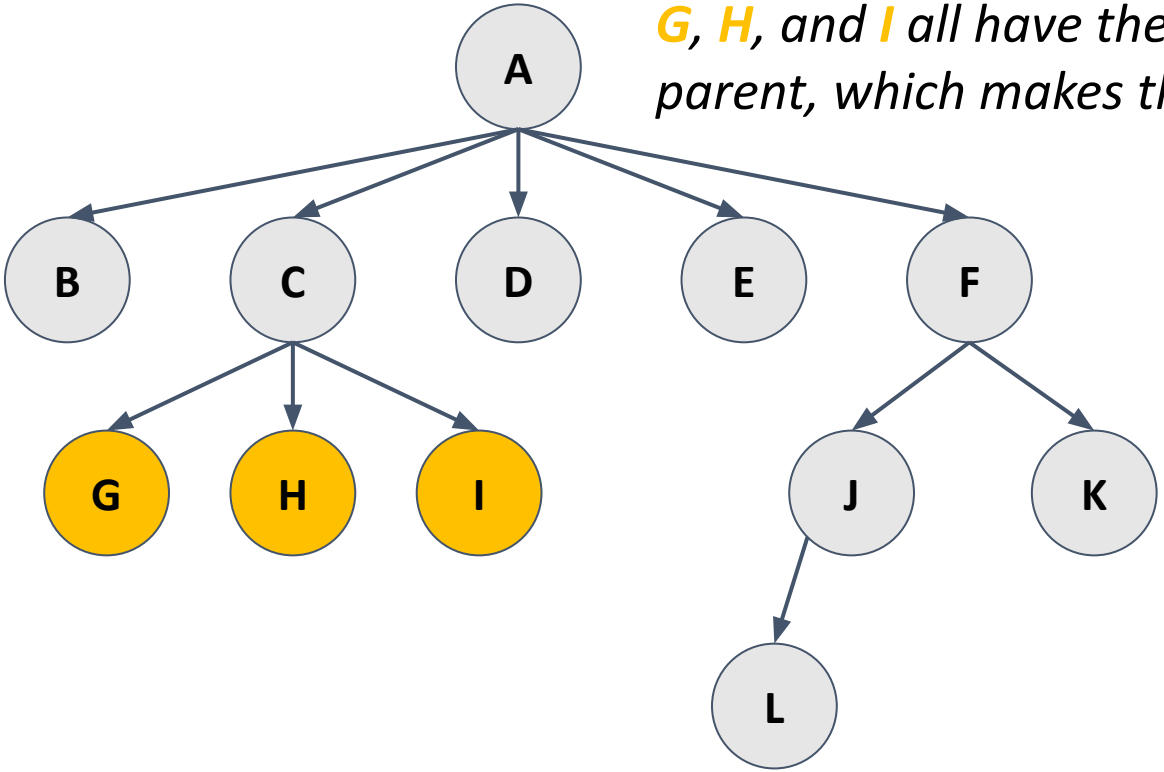


New Tree Terminology



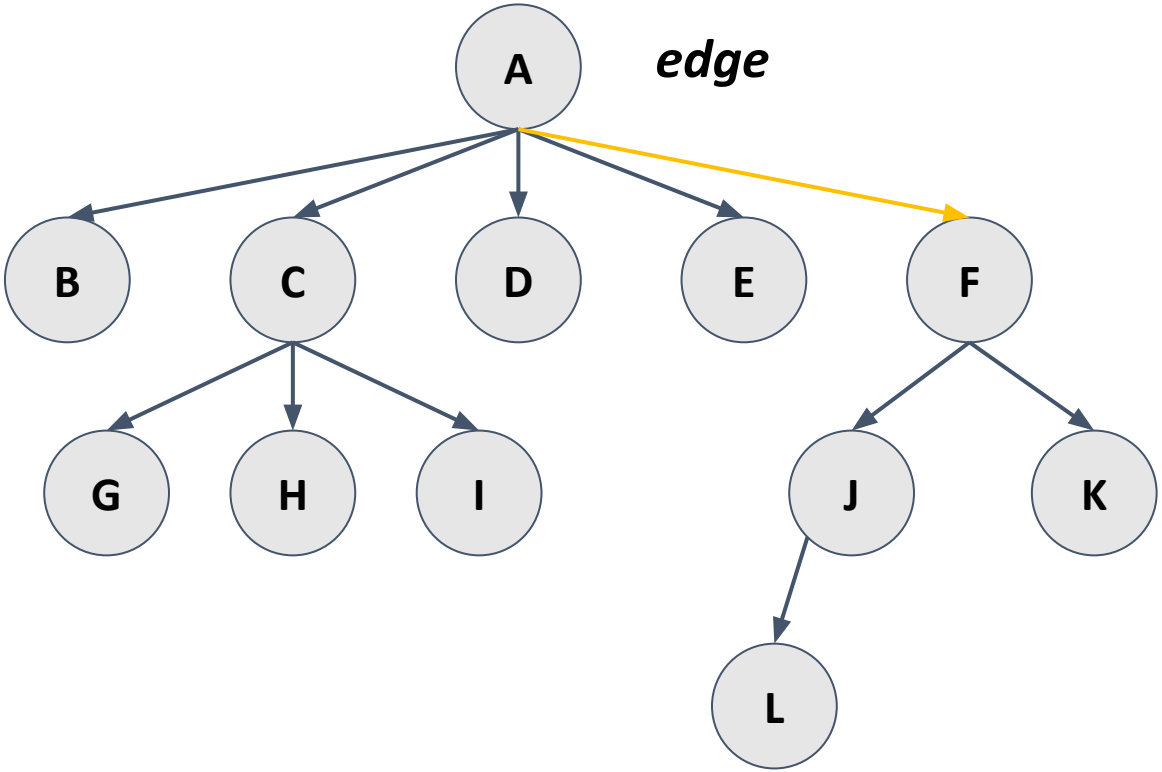
B, *G*, *H*, *I*, *D*, *E*, *L*, and *K* are all ***leaf nodes***

New Tree Terminology

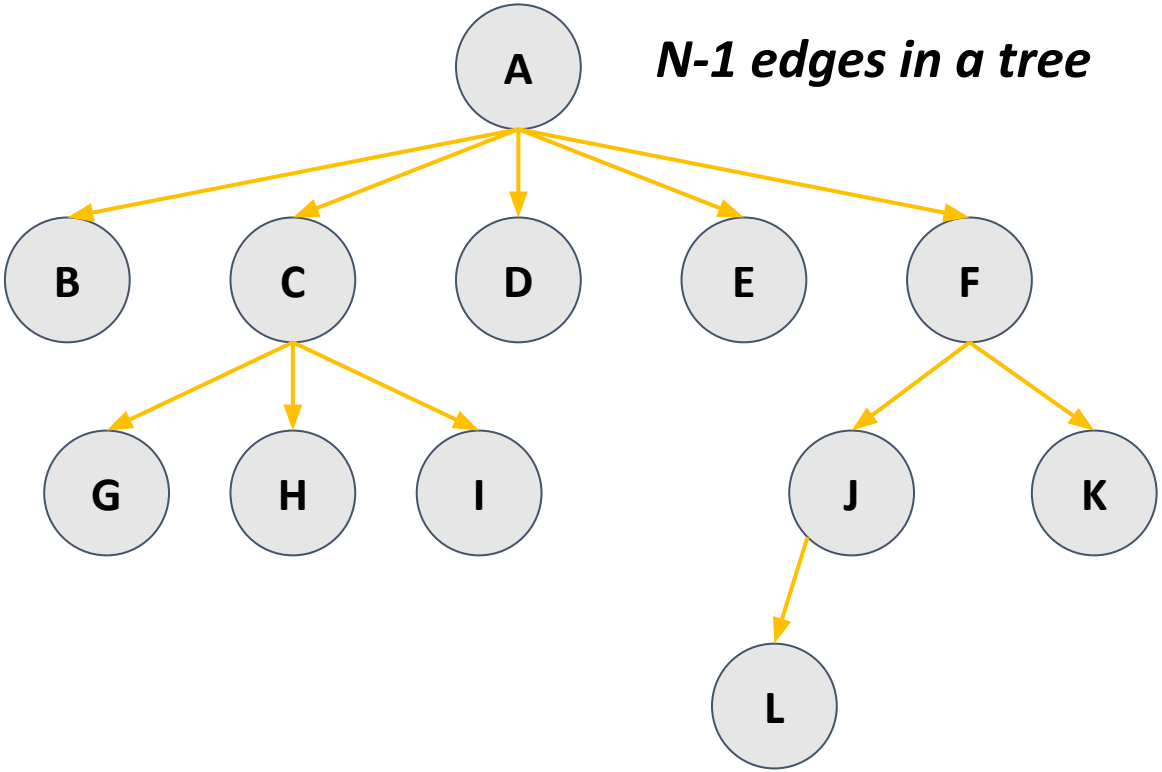


G, *H*, and *I* all have the same parent, which makes them ***siblings***

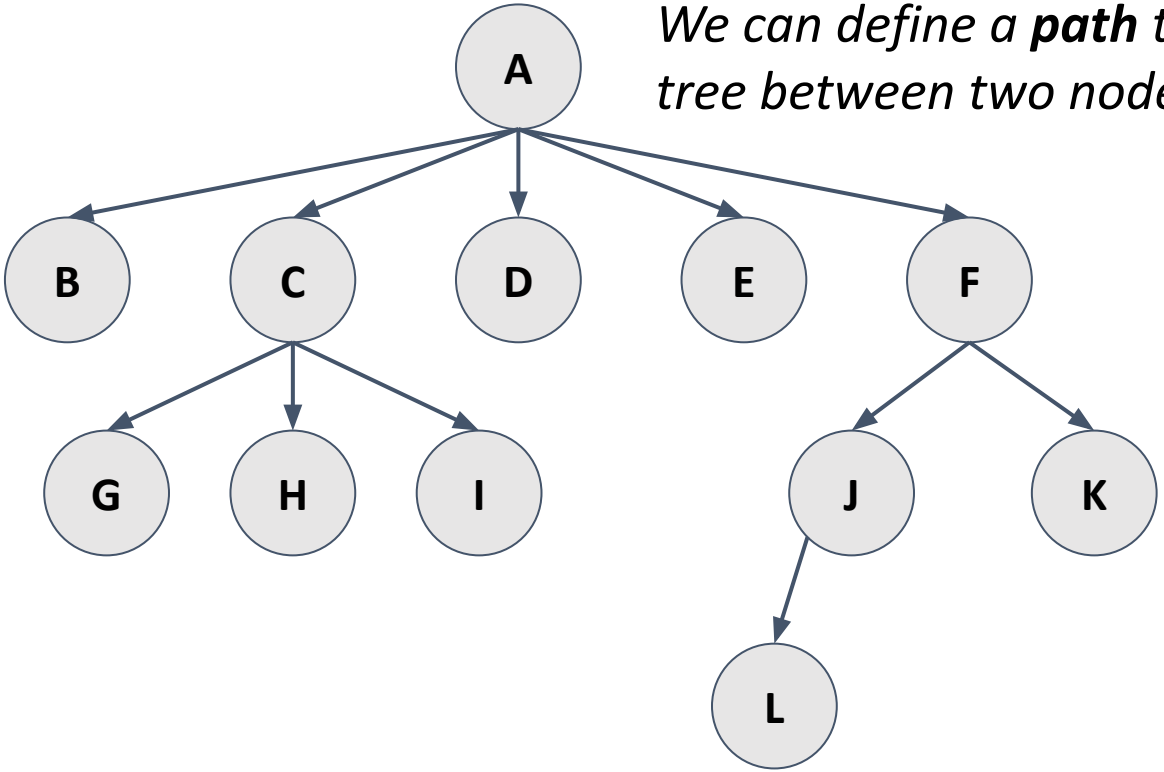
New Tree Terminology



New Tree Terminology

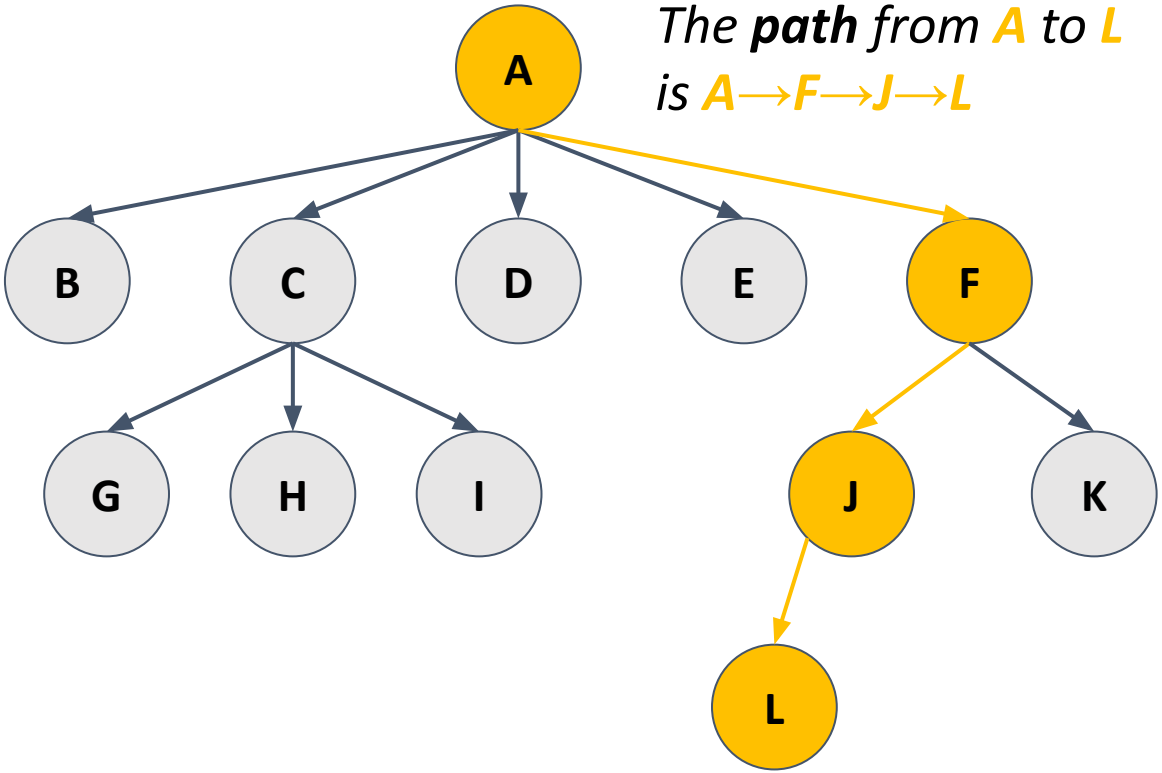


New Tree Terminology

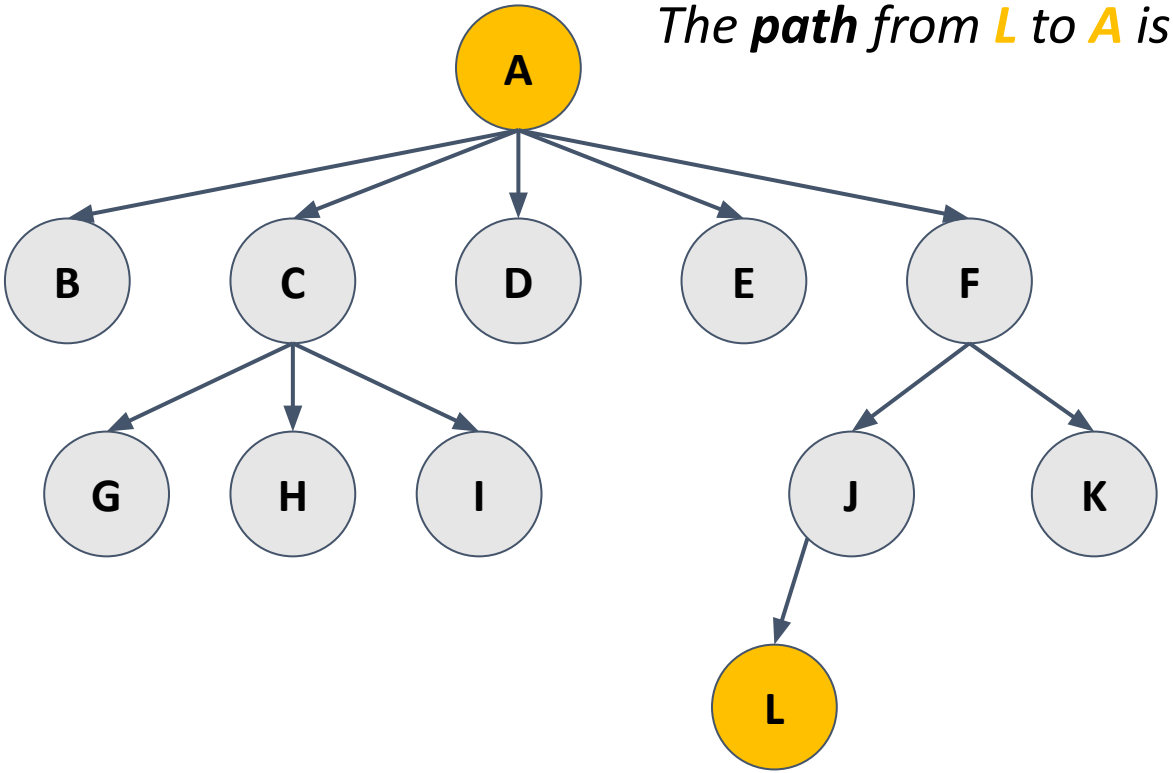


*We can define a **path** through the tree between two nodes*

New Tree Terminology

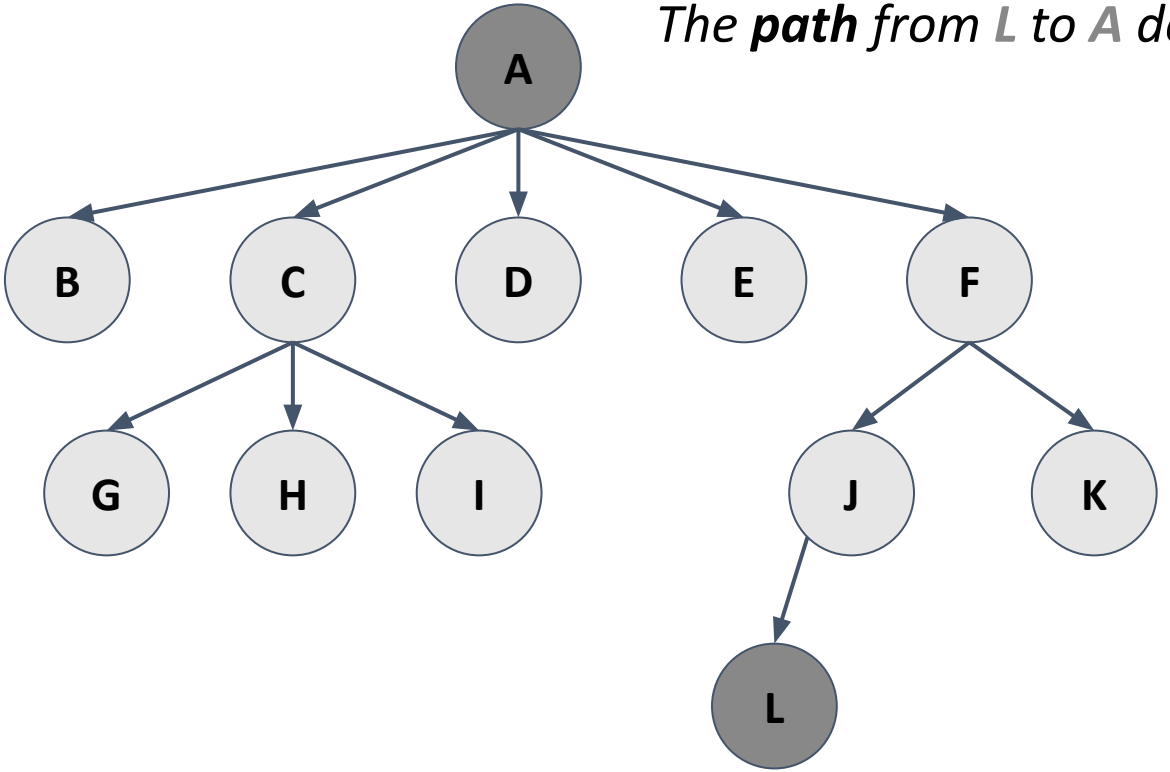


New Tree Terminology

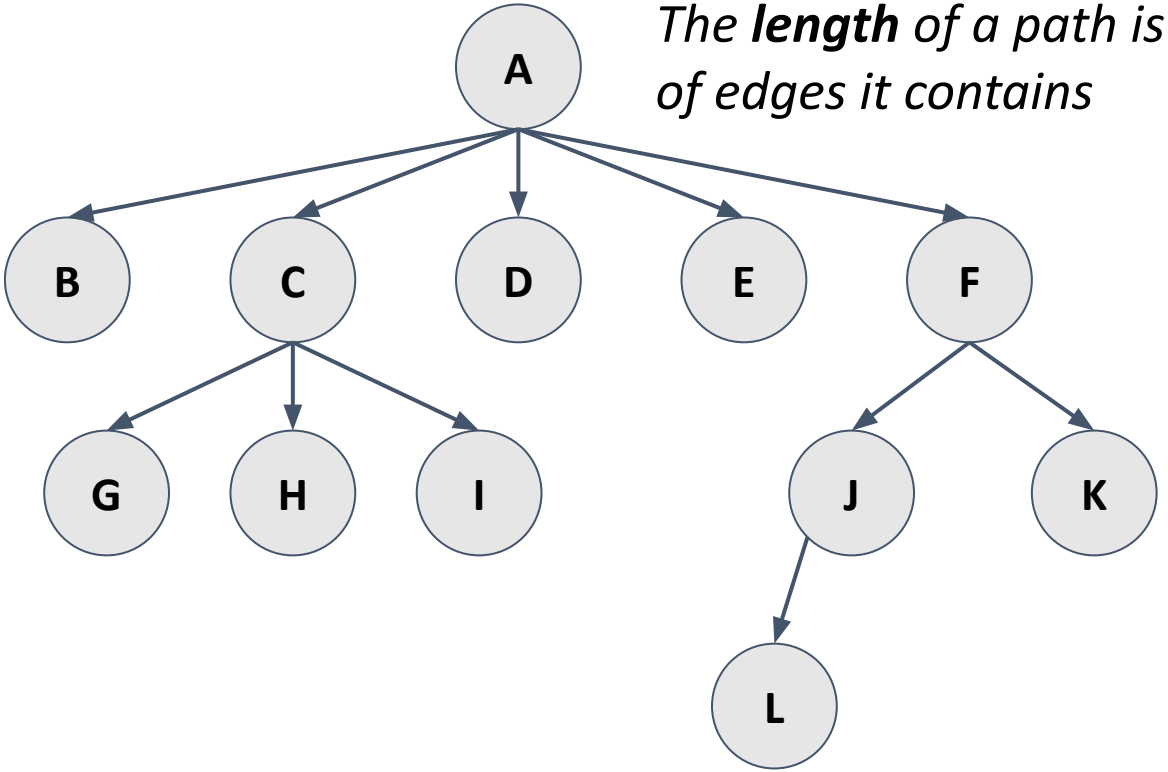


New Tree Terminology

*The **path** from L to A does not exist*

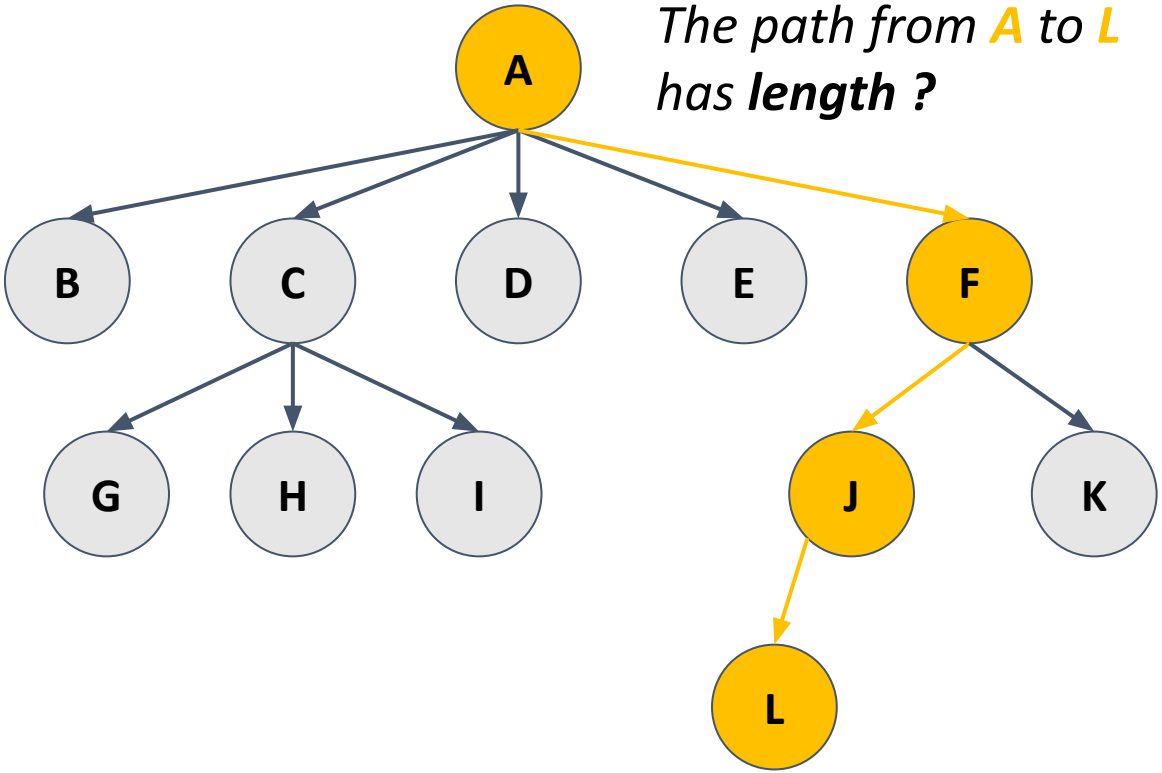


New Tree Terminology

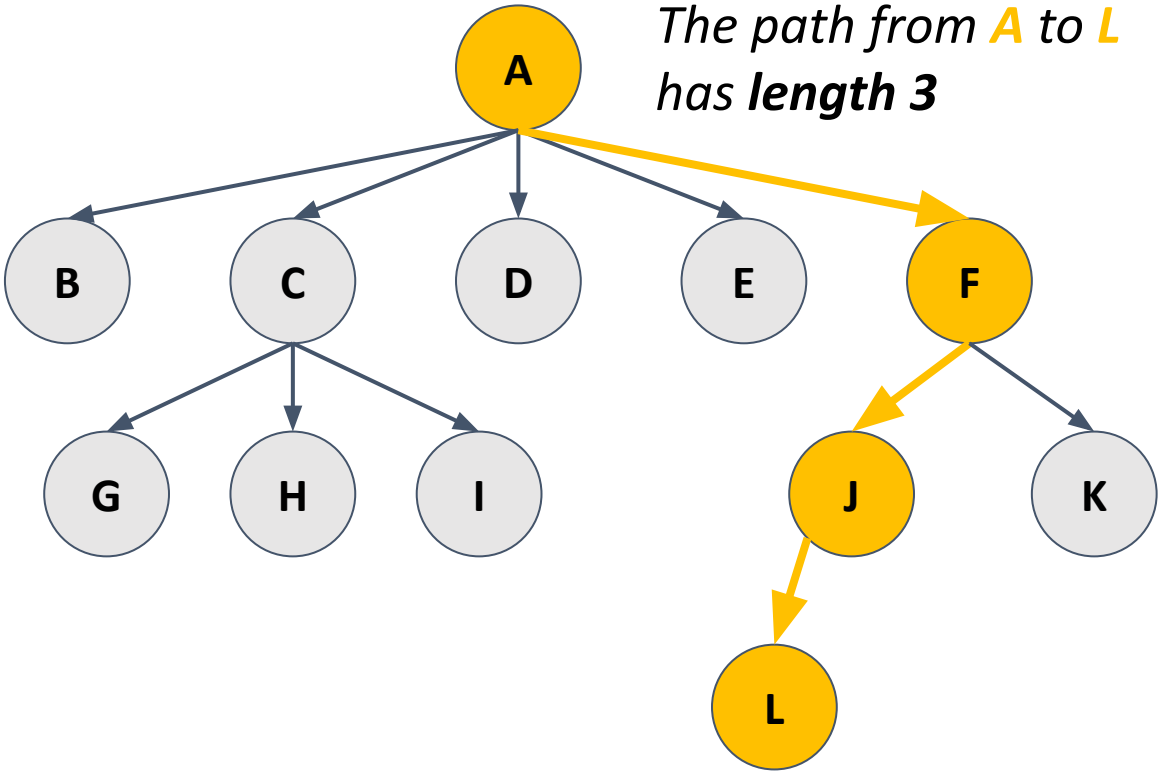


*The **length** of a path is the number of edges it contains*

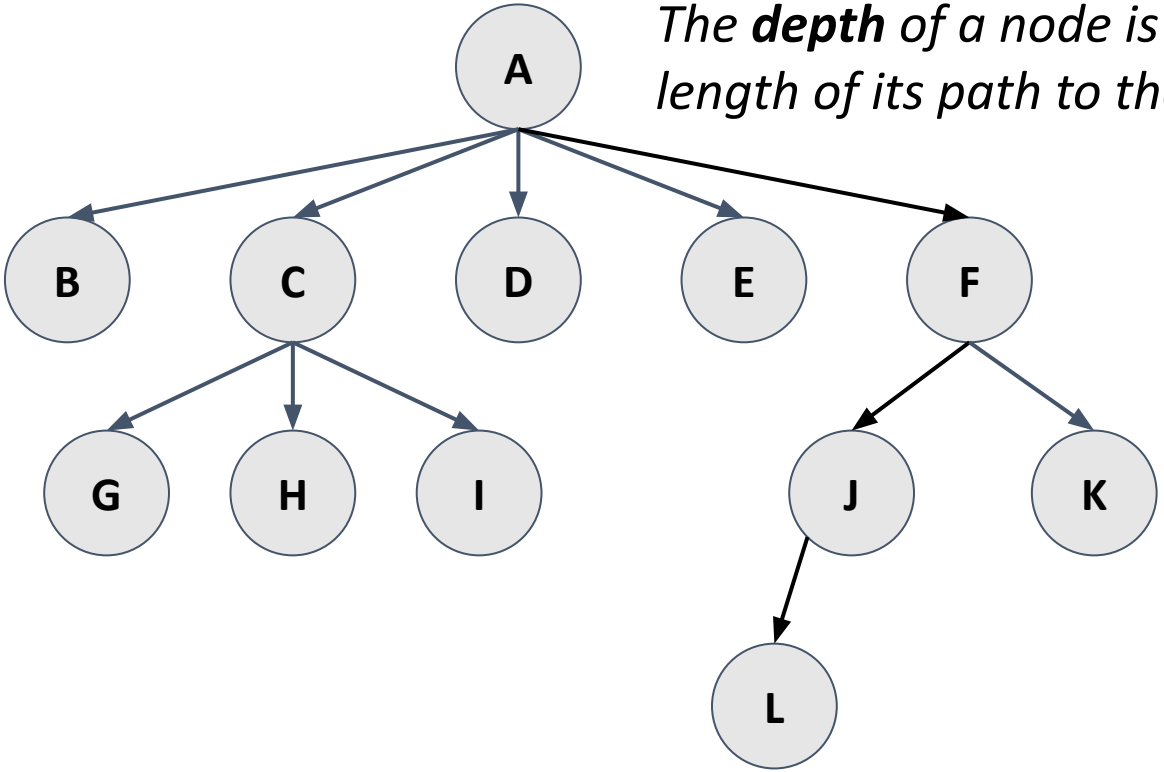
New Tree Terminology



New Tree Terminology

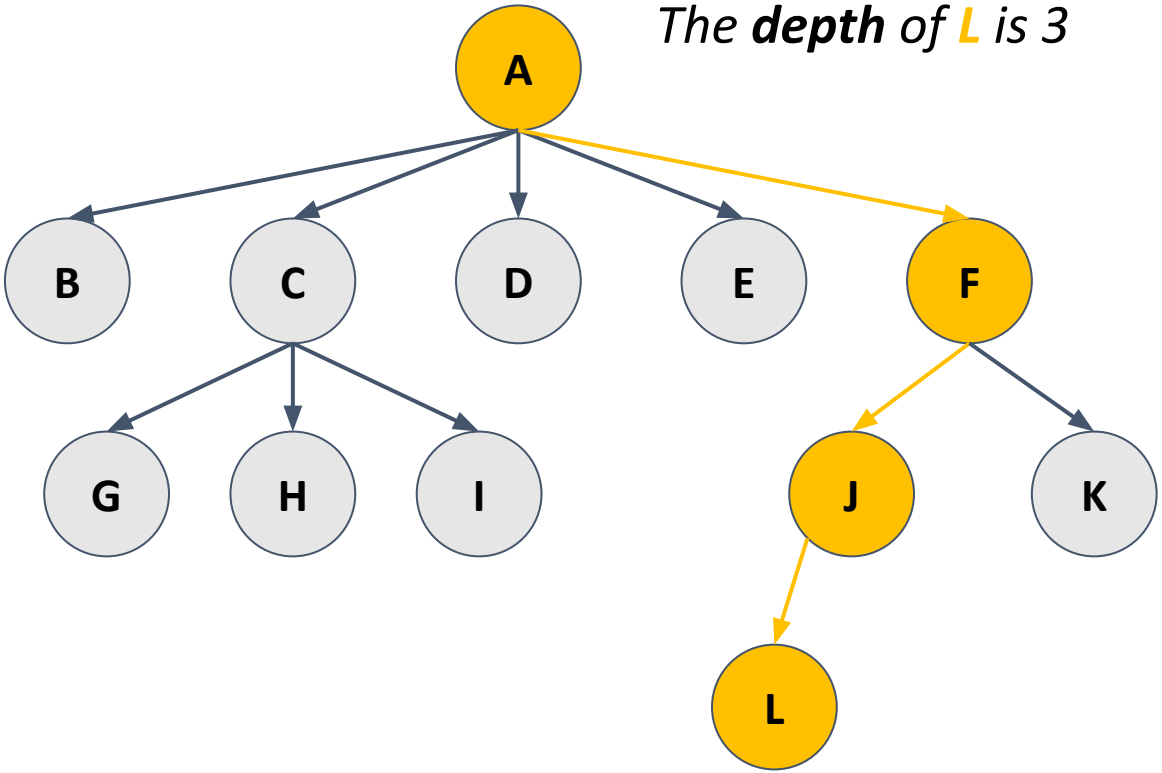


New Tree Terminology

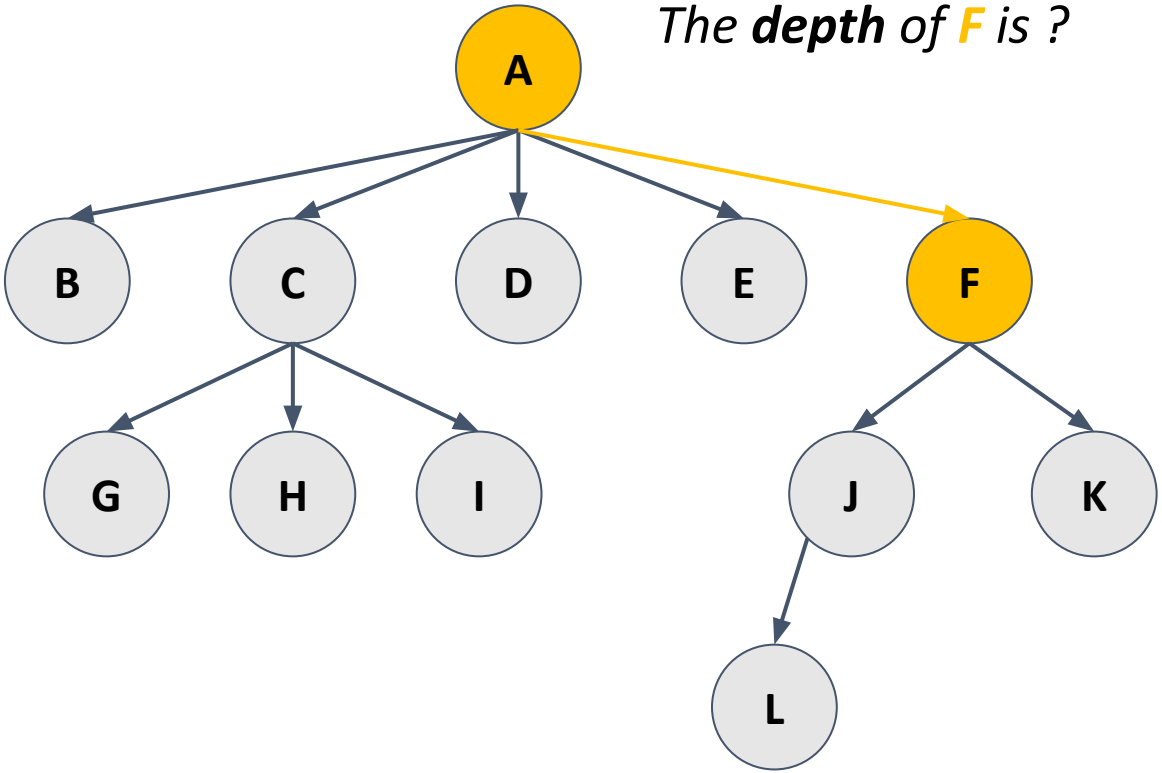


*The **depth** of a node is the length of its path to the root*

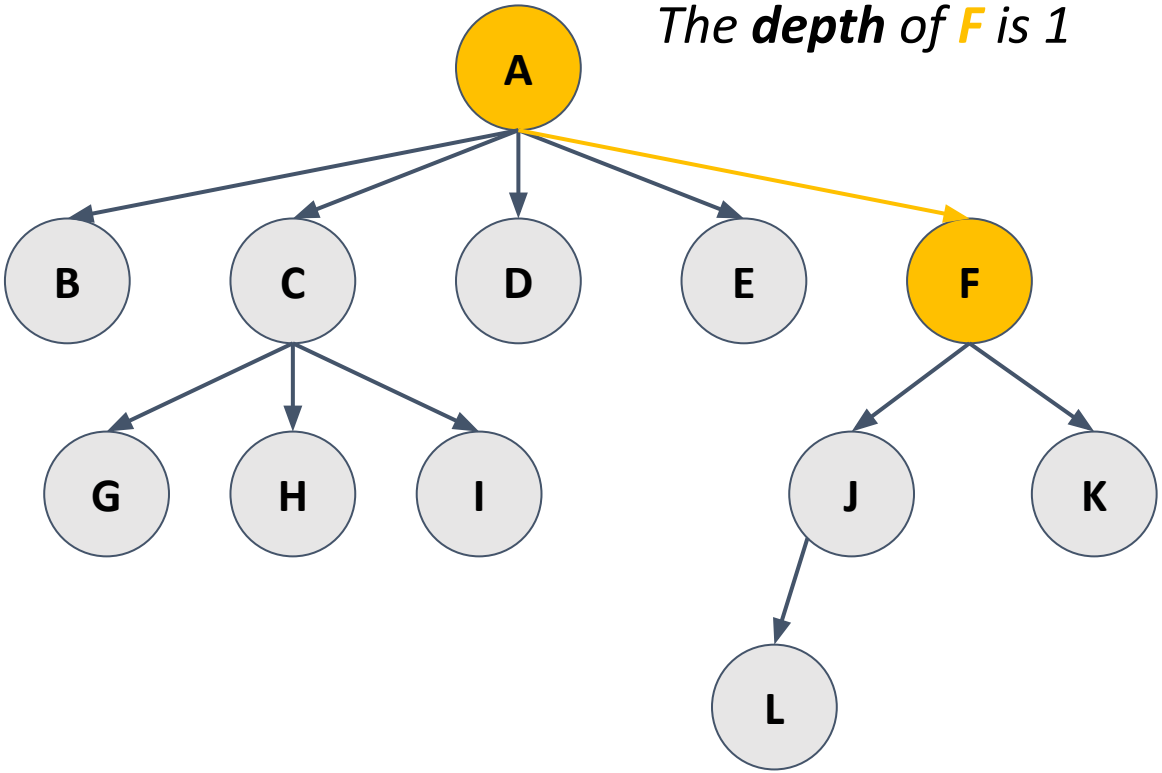
New Tree Terminology



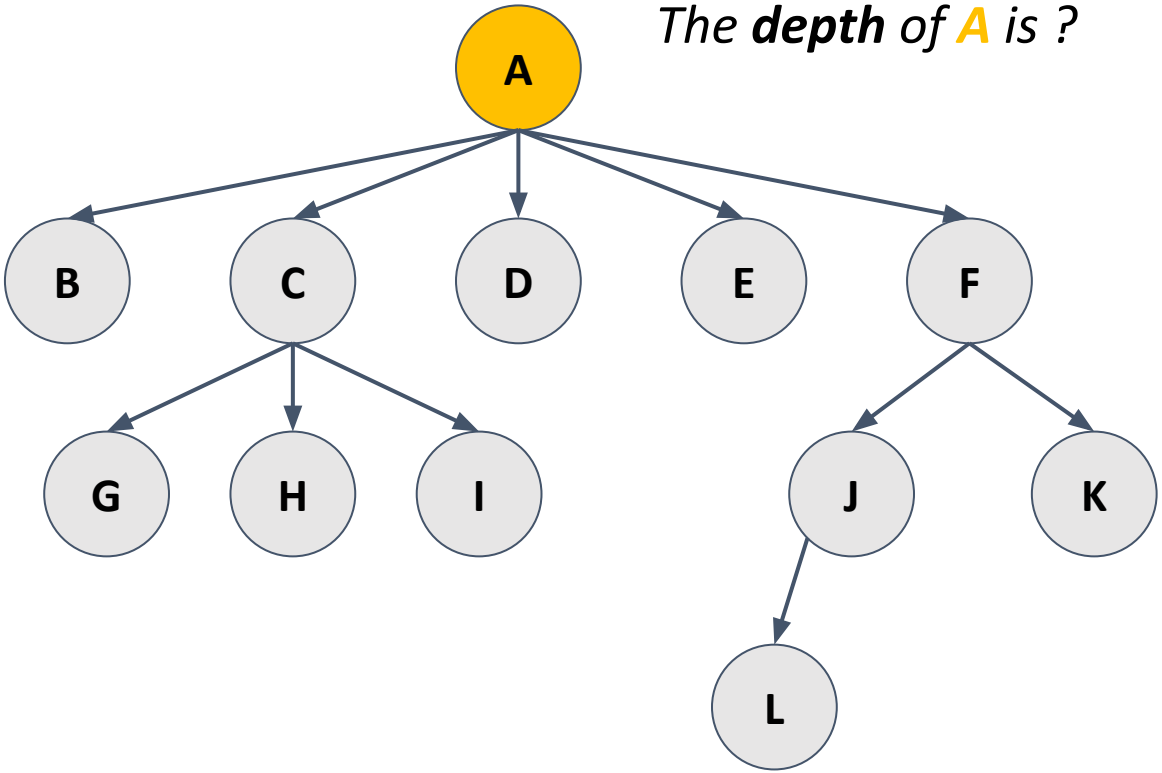
New Tree Terminology



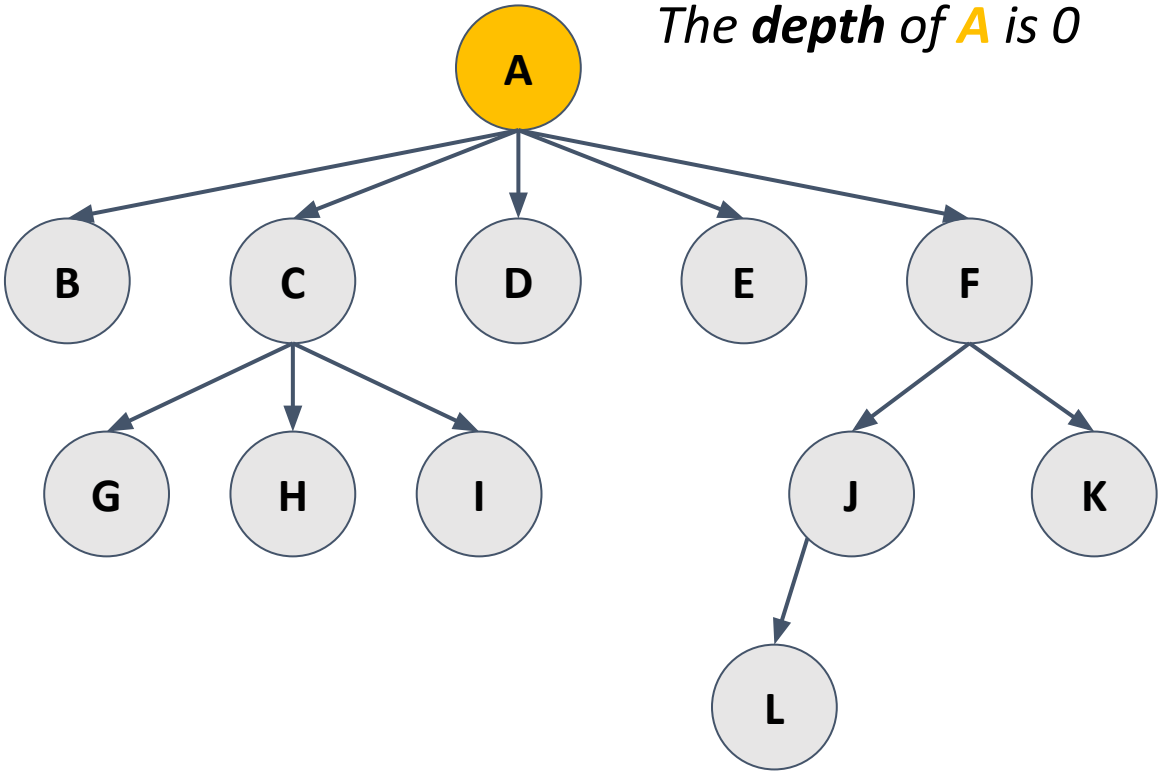
New Tree Terminology



New Tree Terminology

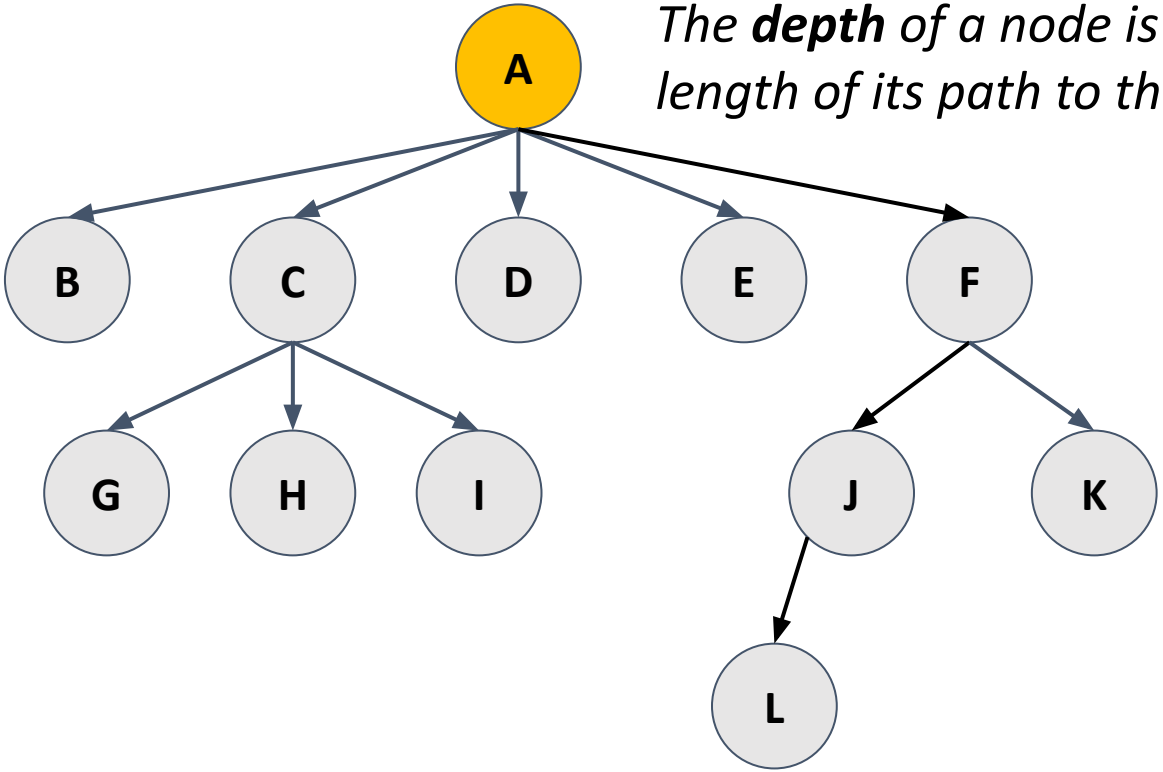


New Tree Terminology



New Tree Terminology

depth: 0

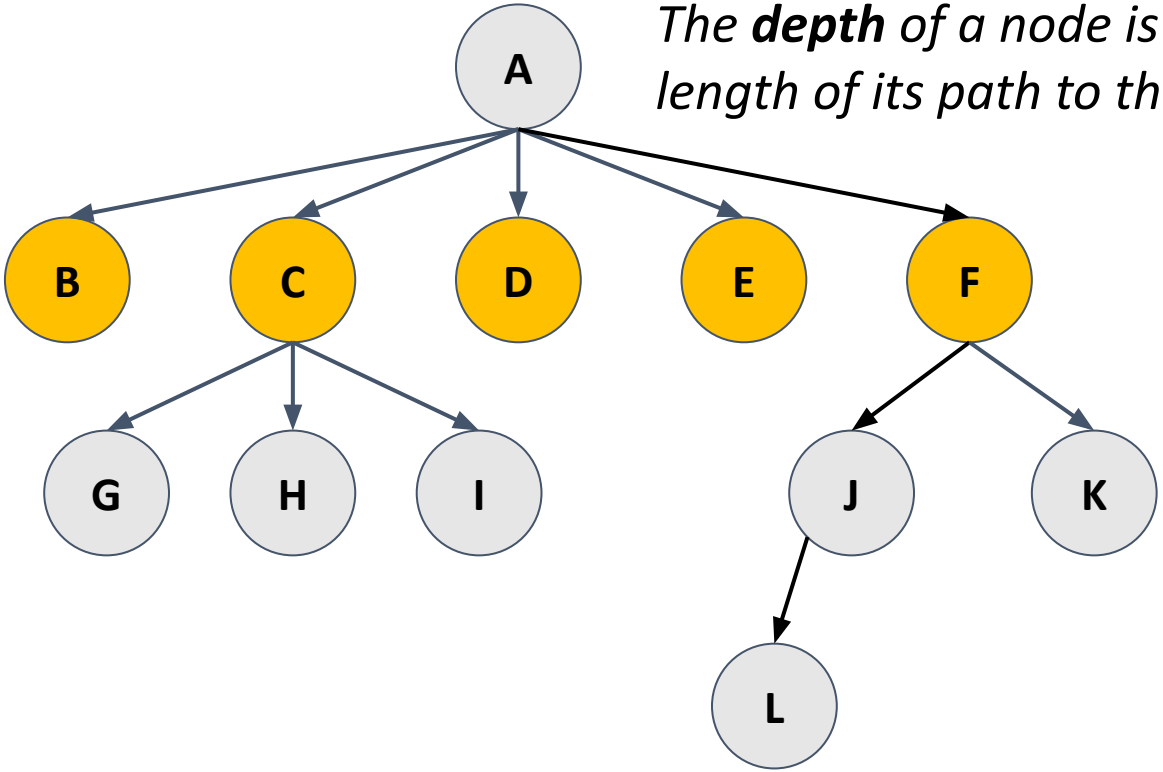


*The **depth** of a node is the length of its path to the root*

New Tree Terminology

depth: 0

depth: 1



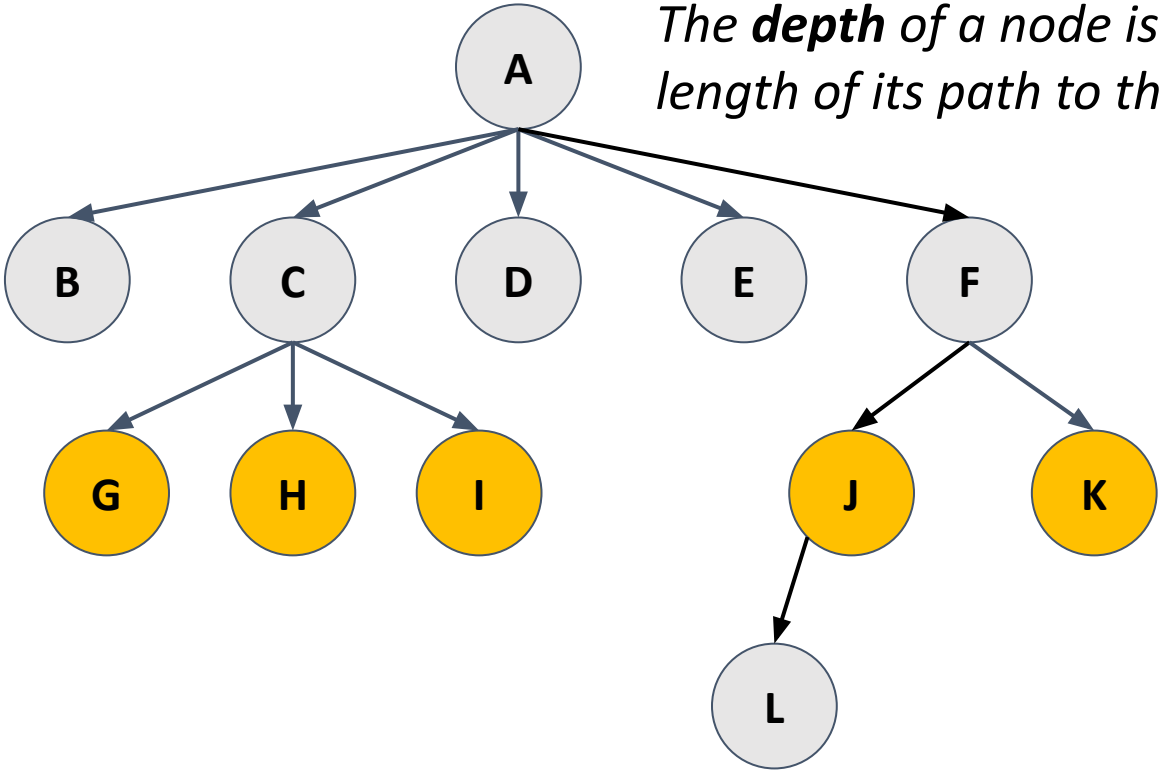
*The **depth** of a node is the length of its path to the root*

New Tree Terminology

depth: 0

depth: 1

depth: 2



*The **depth** of a node is the length of its path to the root*

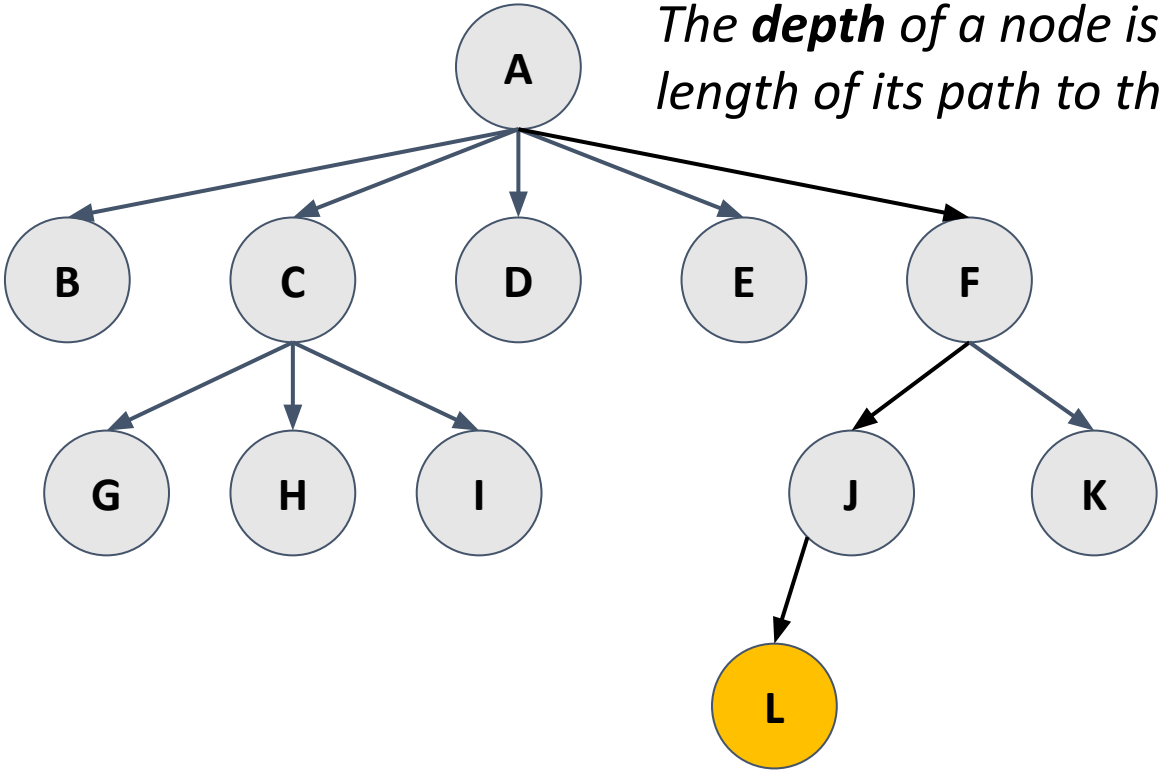
New Tree Terminology

depth: 0

depth: 1

depth: 2

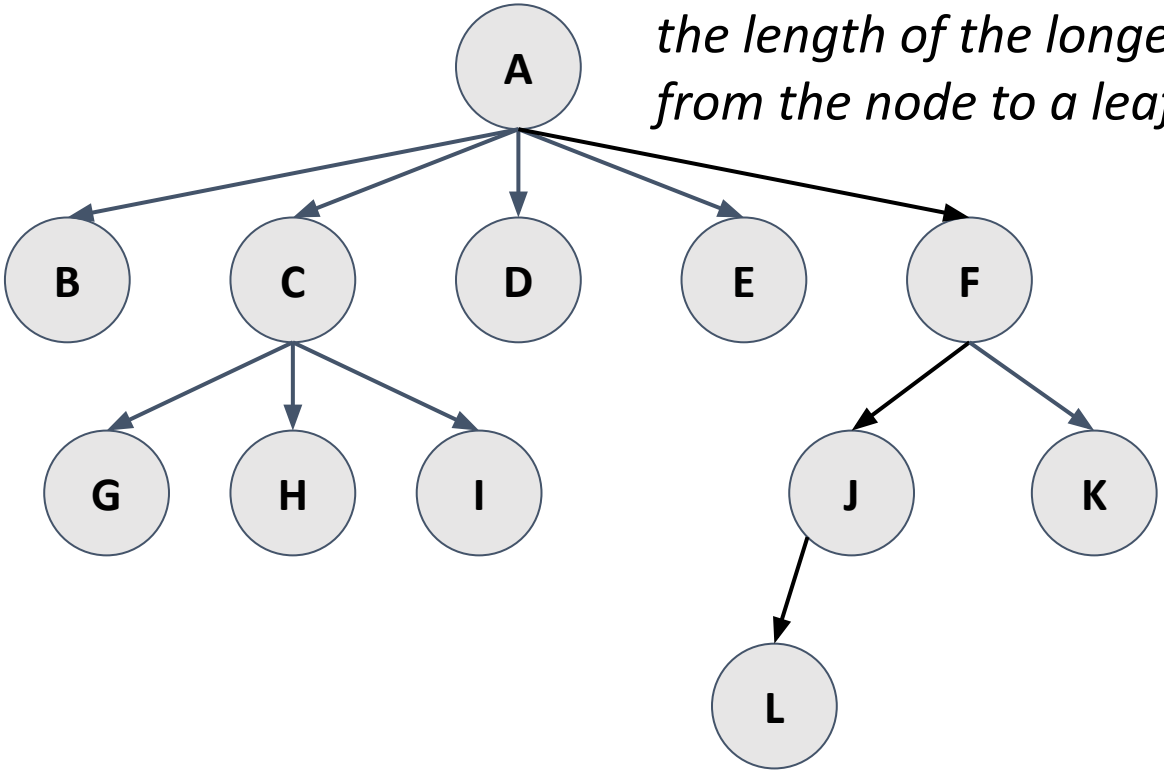
depth: 3



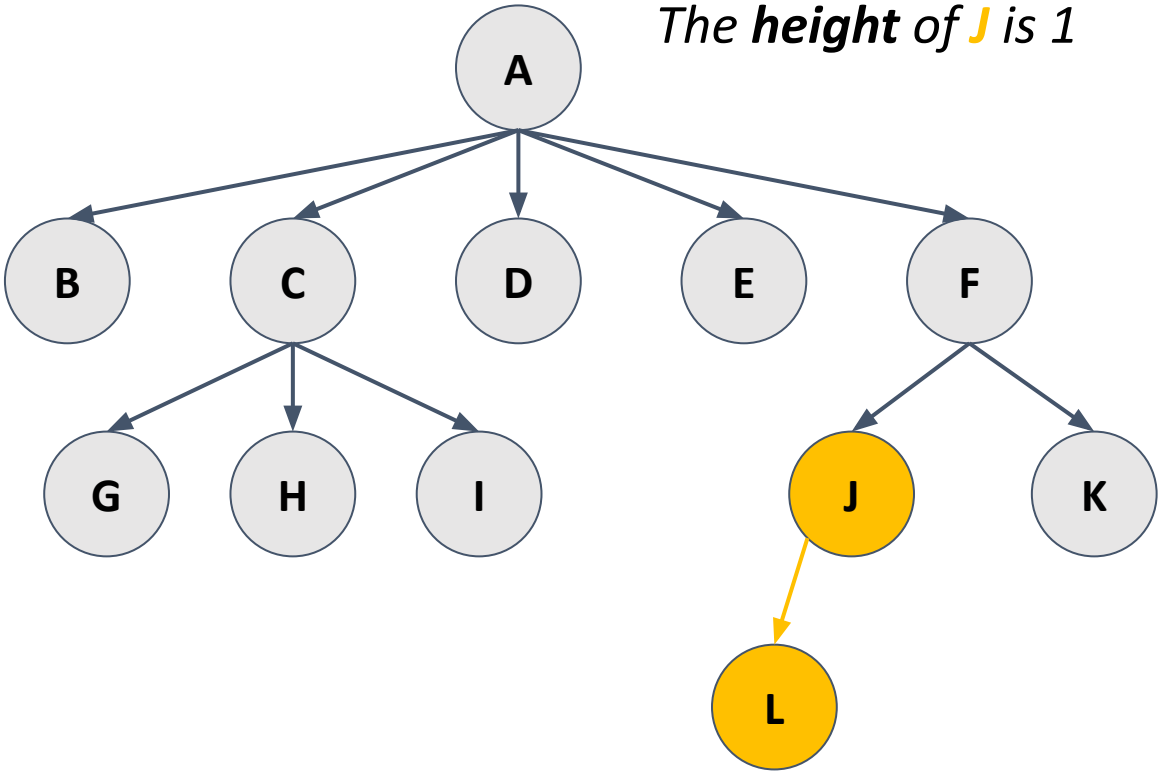
*The **depth** of a node is the length of its path to the root*

New Tree Terminology

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

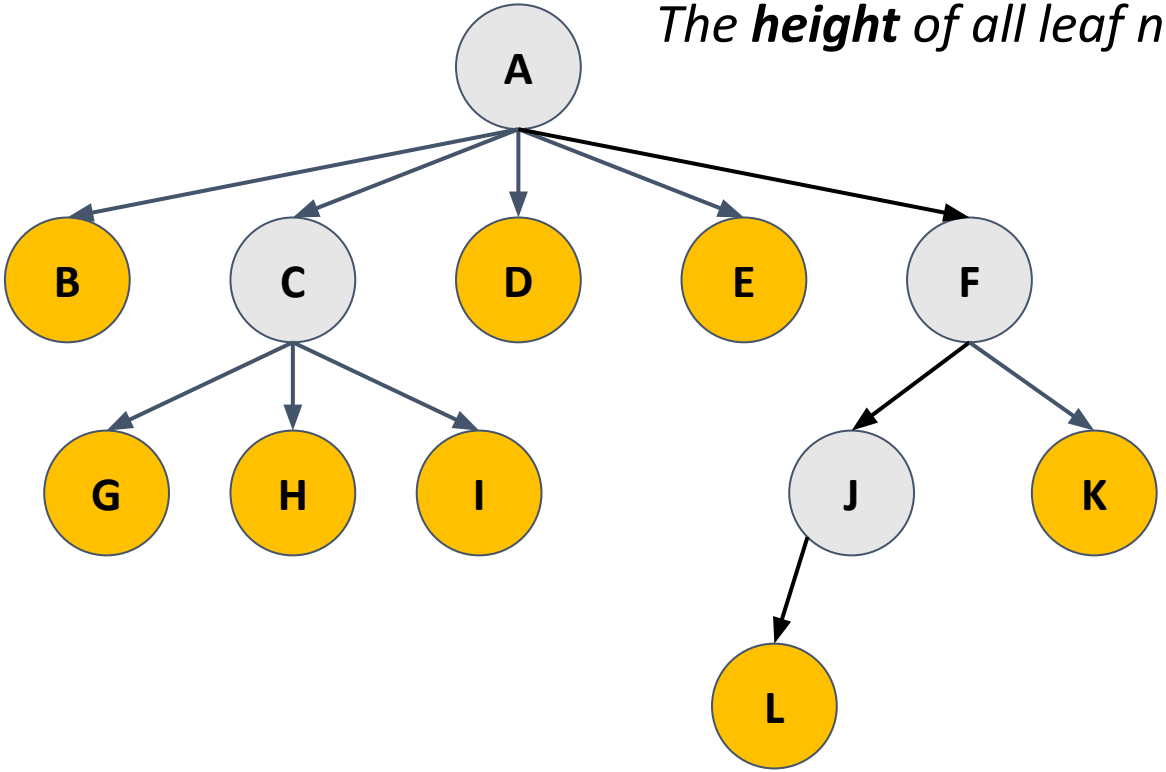


New Tree Terminology



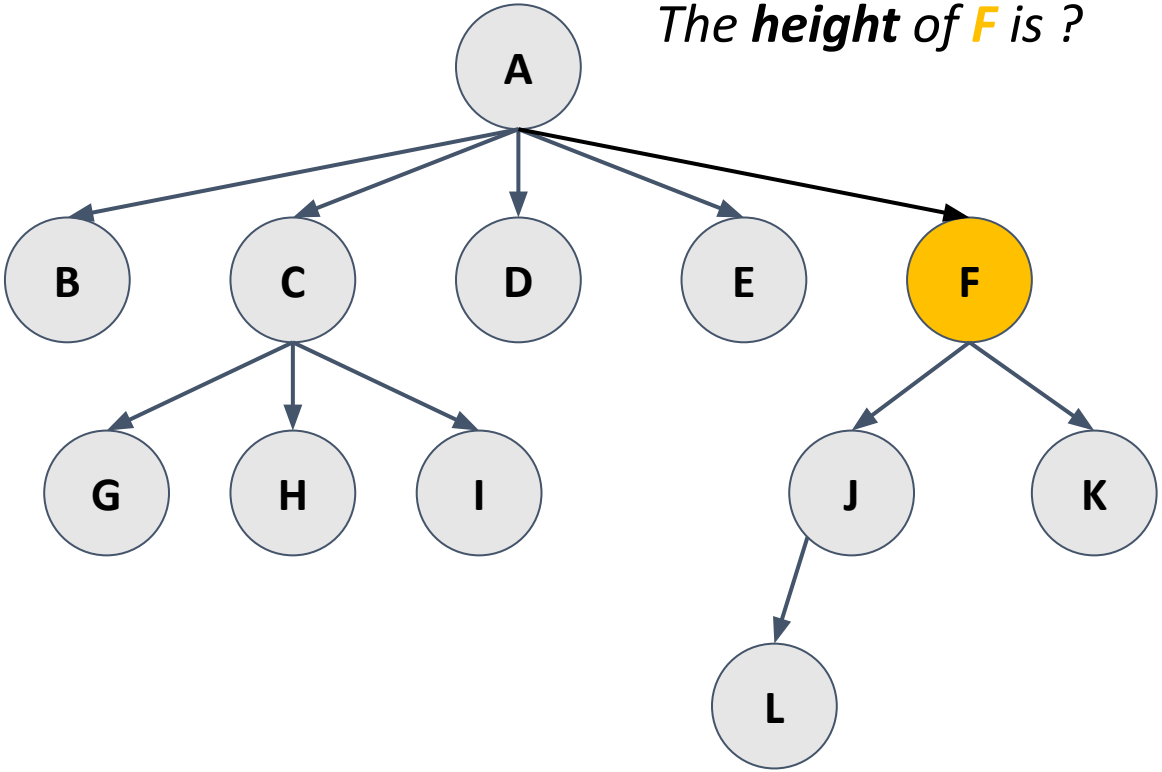
New Tree Terminology

*The **height** of all leaf nodes is 0*

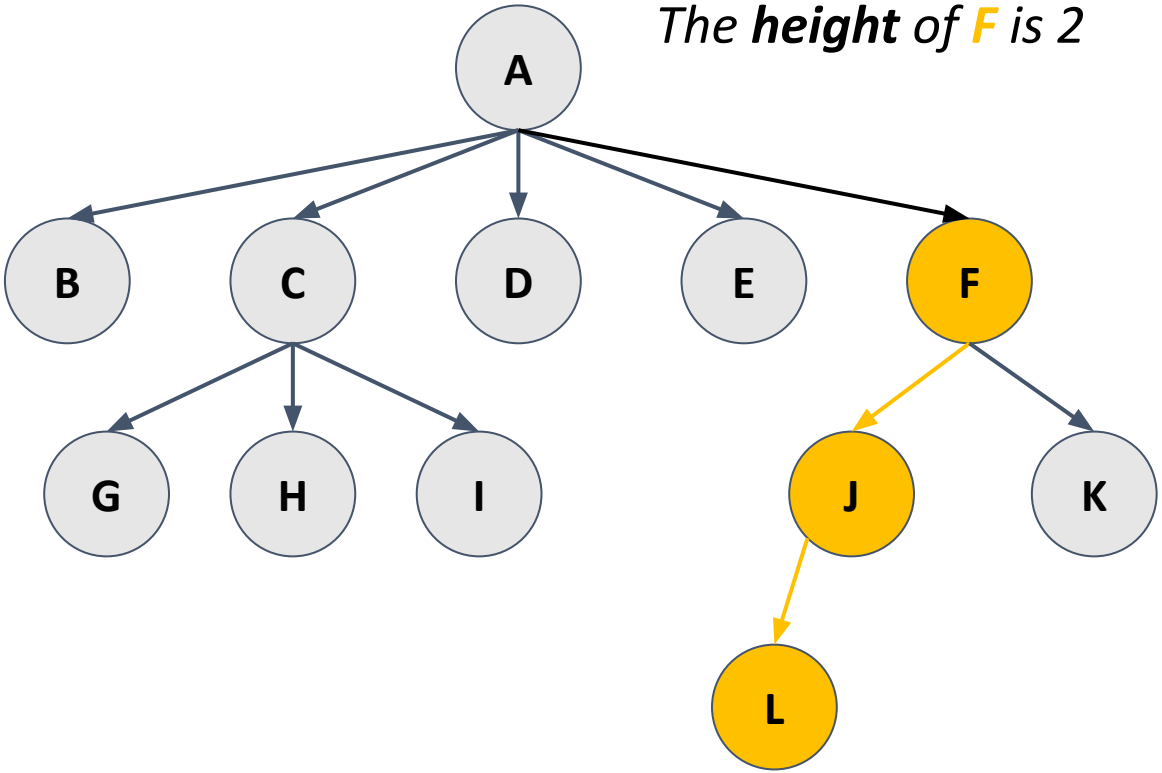


New Tree Terminology

*The **height** of **F** is ?*

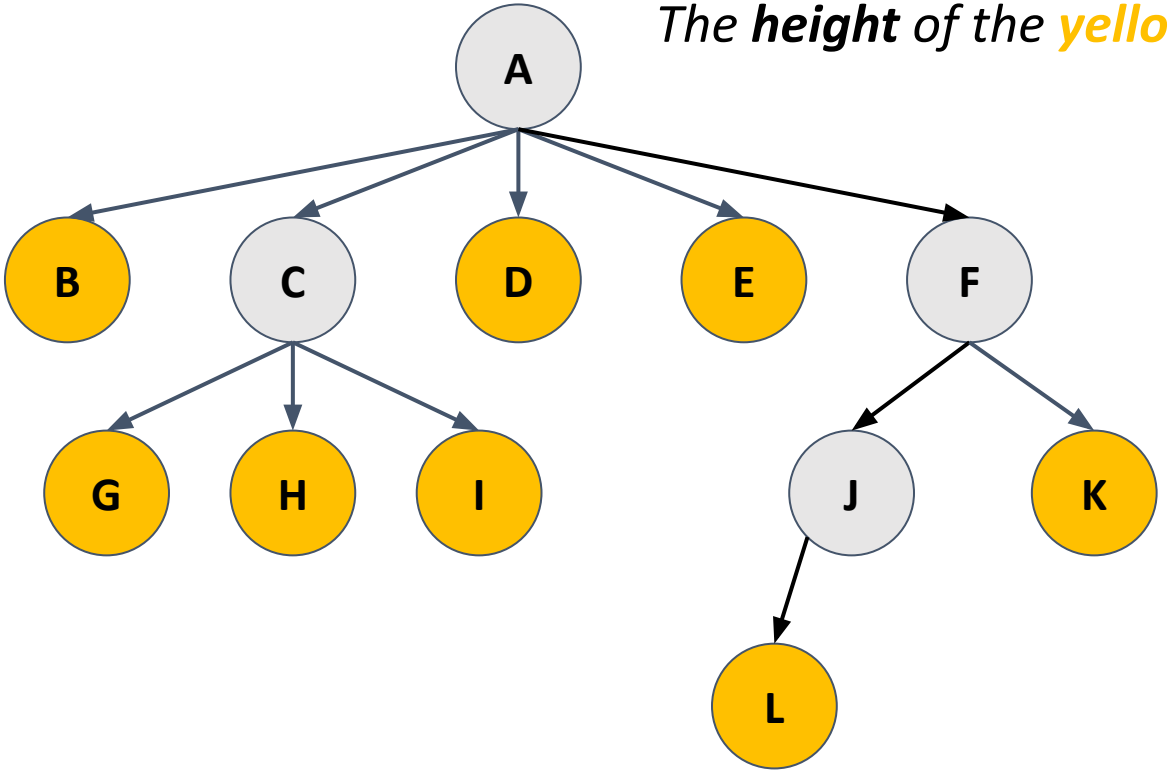


New Tree Terminology



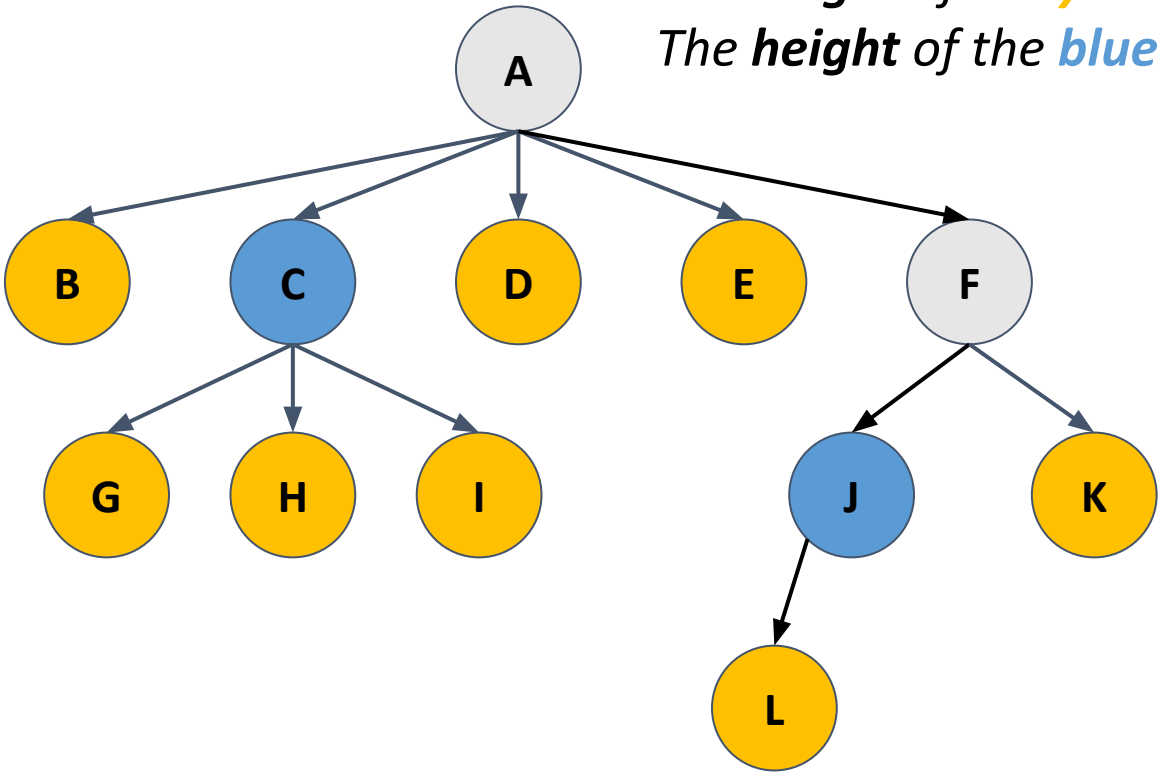
New Tree Terminology

*The **height** of the yellow nodes is 0*



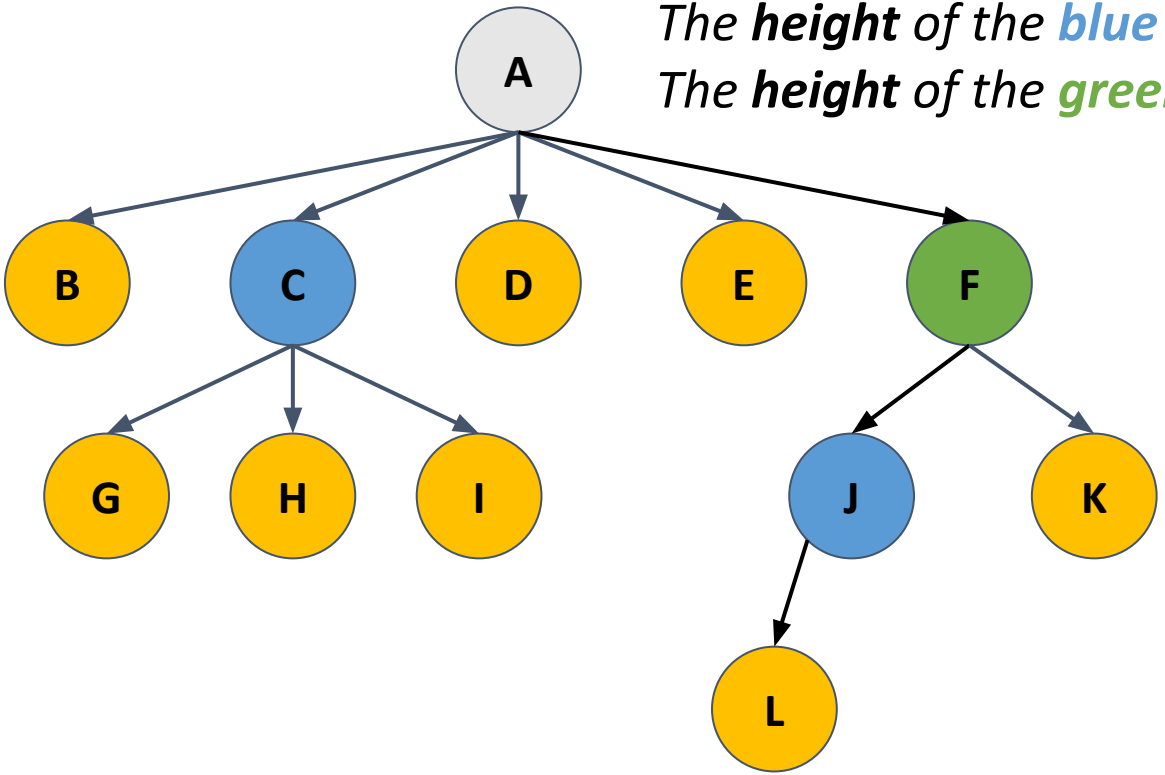
New Tree Terminology

*The **height** of the yellow nodes is 0*
*The **height** of the blue nodes is 1*



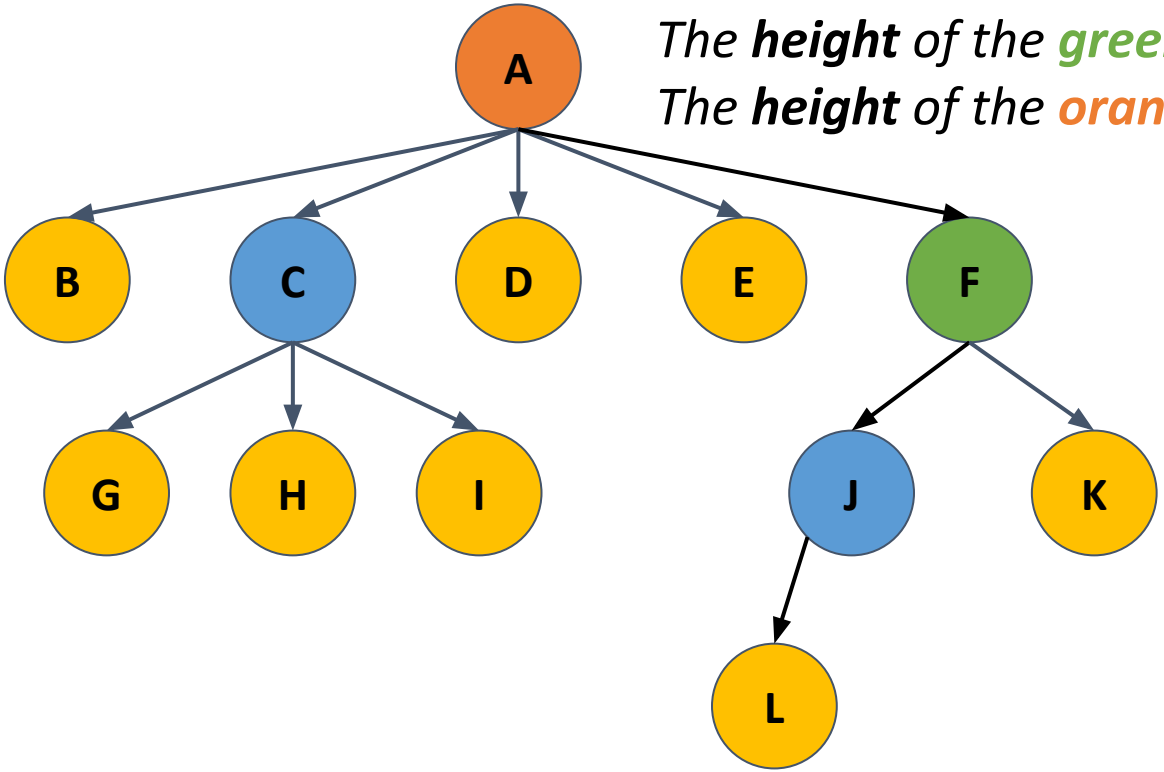
New Tree Terminology

*The **height** of the yellow nodes is 0*
*The **height** of the blue nodes is 1*
*The **height** of the green nodes is 2*



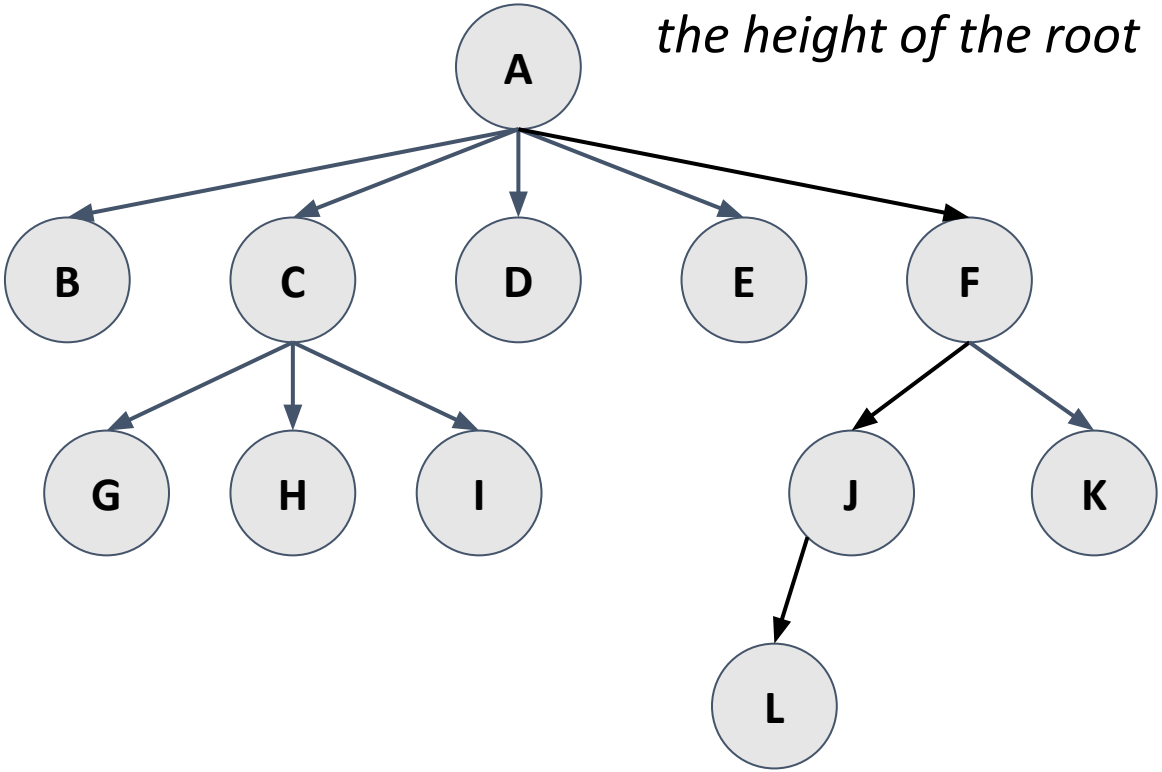
New Tree Terminology

*The **height** of the **yellow** nodes is 0*
*The **height** of the **blue** nodes is 1*
*The **height** of the **green** nodes is 2*
*The **height** of the **orange** nodes is 3*

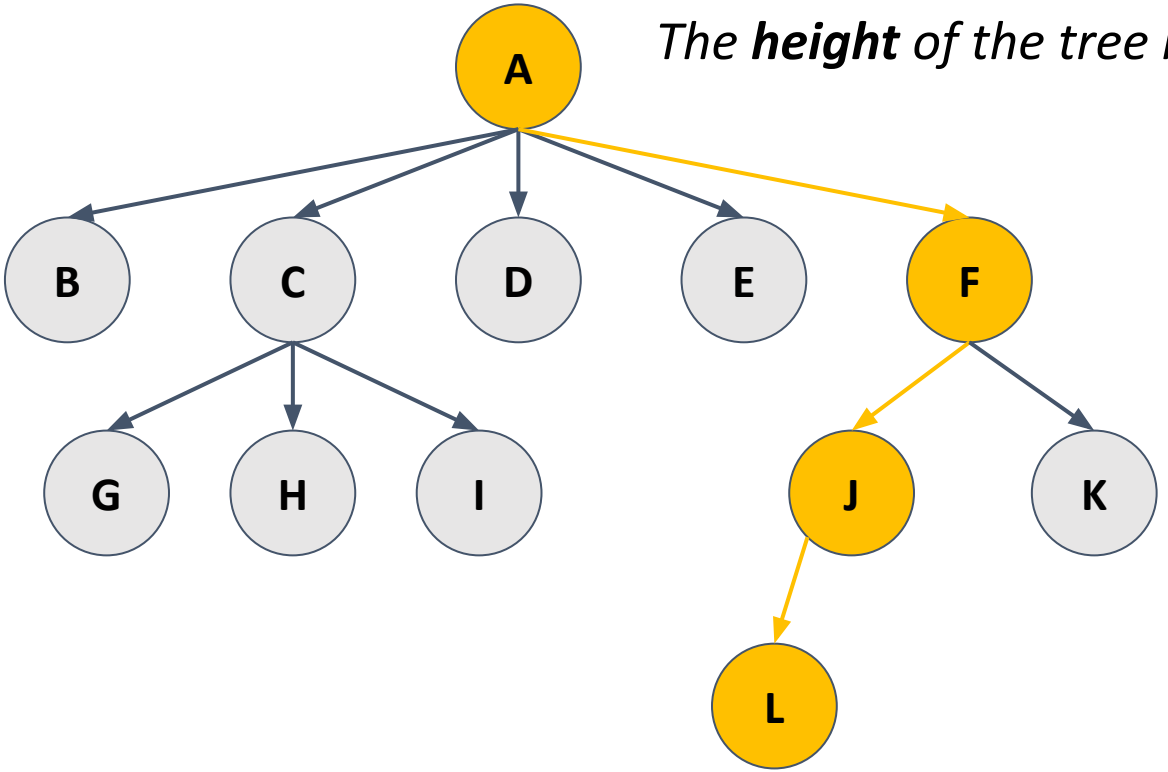


New Tree Terminology

*The **height** of a tree is the height of the root*



New Tree Terminology



*The **height** of the tree is 3*

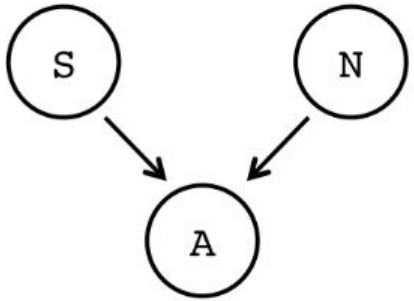
Tree Terminology Summary

- Can be **defined recursively** as either
  - An empty data structure
  - A single node with zero or more non-empty subtrees
- Every non-empty tree has a **root** node that defines the “top” of the tree
- Every node has zero or more **children** nodes
  - Nodes with no children are called **leaf** nodes
- Every node in the tree has exactly one **parent** node (except for the root)
- A **path** through the tree traverses edges between parents and their children
- The **depth** of a node is the length of the path between the root and that node
- The **height** of a tree is the number of nodes in the longest path through the tree

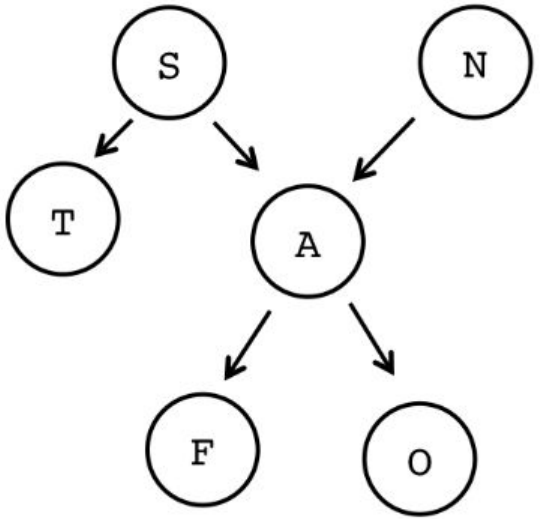
Tree Properties

Tree Properties

- Any node in a tree can only have one parent

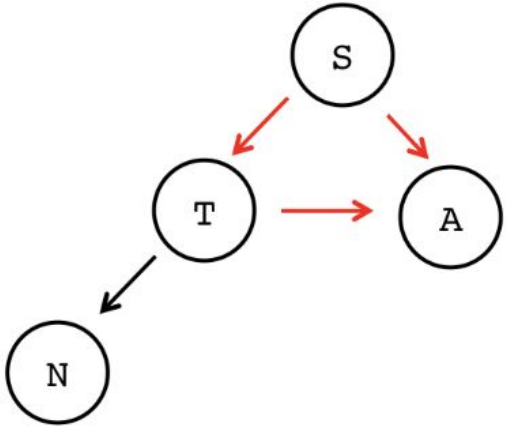


Not trees!



Tree Properties

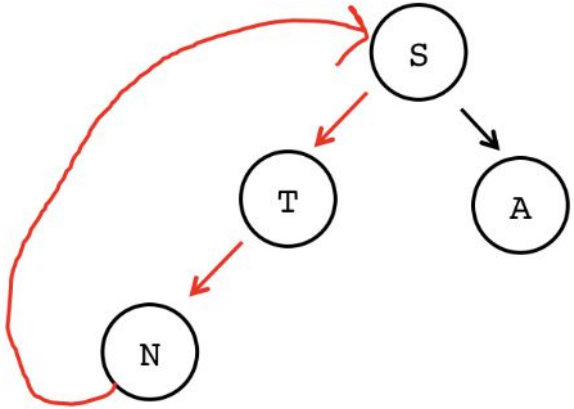
- Any node in a tree can only have one parent



Not a tree!

Tree Properties

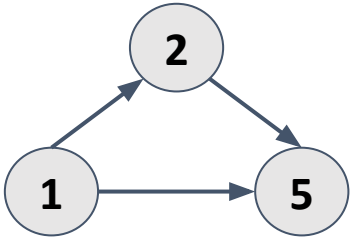
- Any node in a tree can only have one parent
- A tree cannot have cycles or loops



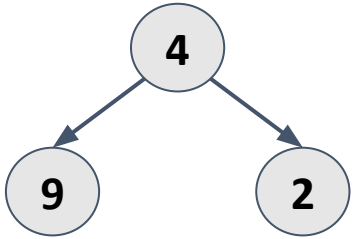
Not a tree!

Which of these are trees?

A.

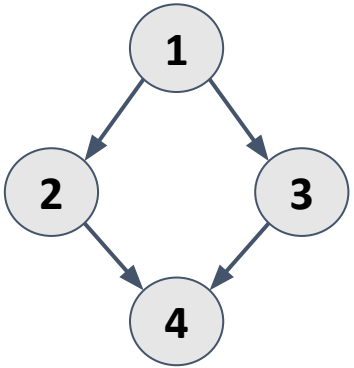


B.



C.

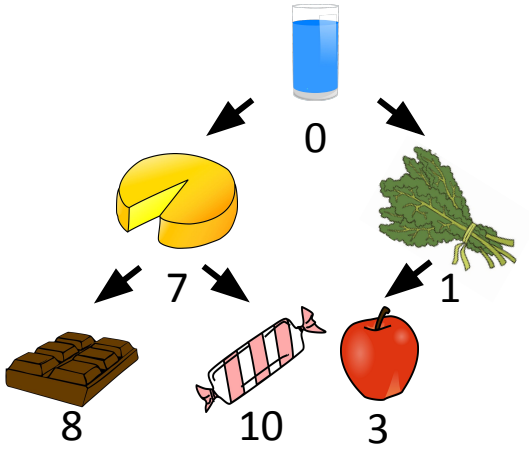
D.



E.

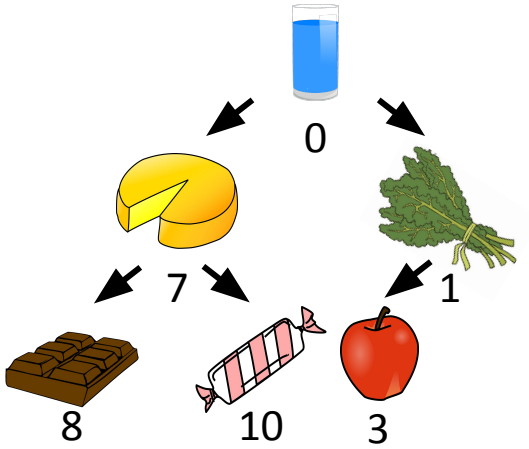
Binary Trees

- Today, we've seen that nodes in a tree can have a variable amount of children (subtrees)
- Previously, we've worked with binary trees

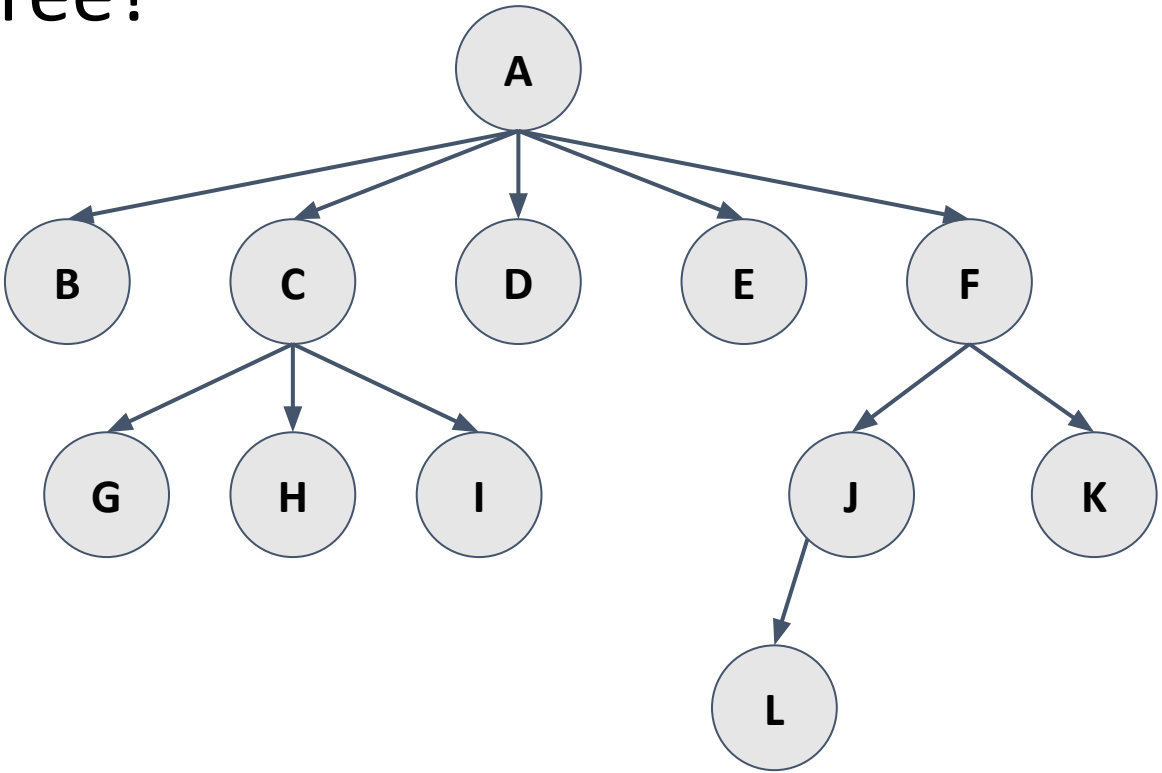


Binary Trees

- Today, we've seen that nodes in a tree can have a variable amount of children (subtrees)
- Previously, we've worked with binary trees
  - Most common trees in CS
  - Every node has either 0, 1, or 2 children
  - No node may have more than 2 children
  - Children are referred to as left child and right child



Binary Tree?

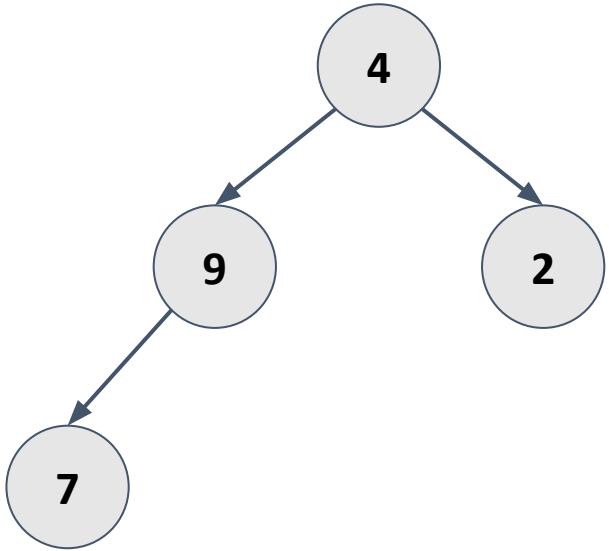


Not a binary tree!

Binary Tree?

Not a binary tree!

Binary Tree?

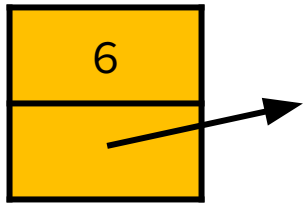


A binary tree!

Building Binary Trees

Building Linked Lists (Recap)

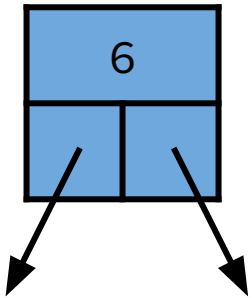
- A linked list is a chain of nodes
- Each node is a struct that contains:
  - A piece of data (like an int, or string)
  - A pointer to the next node




```
struct Node {
   int data;
   Node* next;
};
```

Building Binary Trees

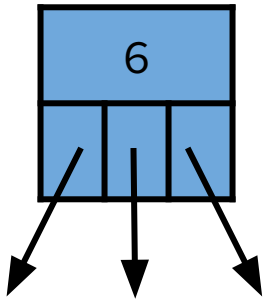
- A binary tree is composed of nodes
- Each node is a struct that contains:
  - A piece of data (like an int, or string)
  - A pointer to the left child
  - A pointer to the right child




```
struct TreeNode {
   int data;
   TreeNode* left;
   TreeNode* right;
};
```

Building Ternary Trees

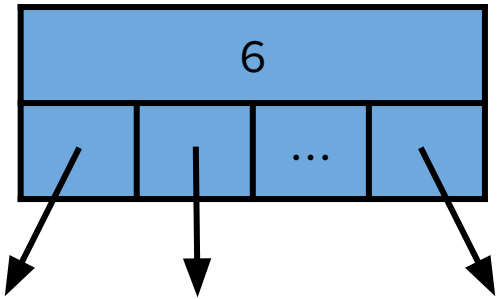
- A ternary tree is composed of nodes
- Each node is a struct that contains:
  - A piece of data (like an int, or string)
  - A pointer to the left child
  - A pointer to the middle child
  - A pointer to the right child




```
struct TernaryTreeNode {
    int data;
    TernaryTreeNode* left;
    TernaryTreeNode* middle;
    TernaryTreeNode* right;
};
```

Building N-ary Trees

- An N-ary tree is composed of nodes
- Each node is a struct that contains:
  - A piece of data (like an int, or string)
  - A vector of pointers to the children



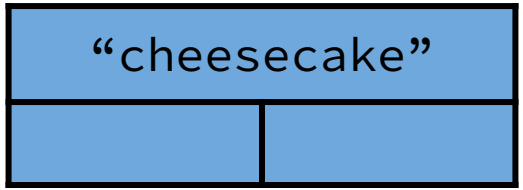

```
struct NAryTreeNode {
    int data;
    Vector<NAryTreeNode*> children;
};
```

Building Binary Tree

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
};
```

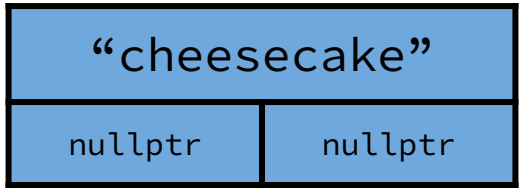
Building Binary Tree

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
};
```



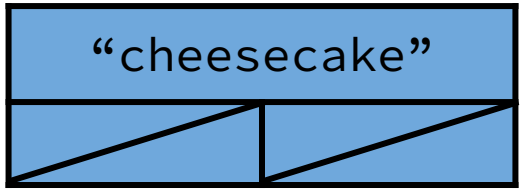
Building Binary Tree

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
};
```



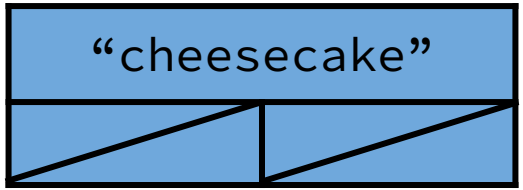
Building Binary Tree

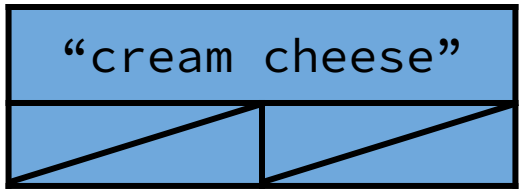
```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
};
```



Building Binary Tree

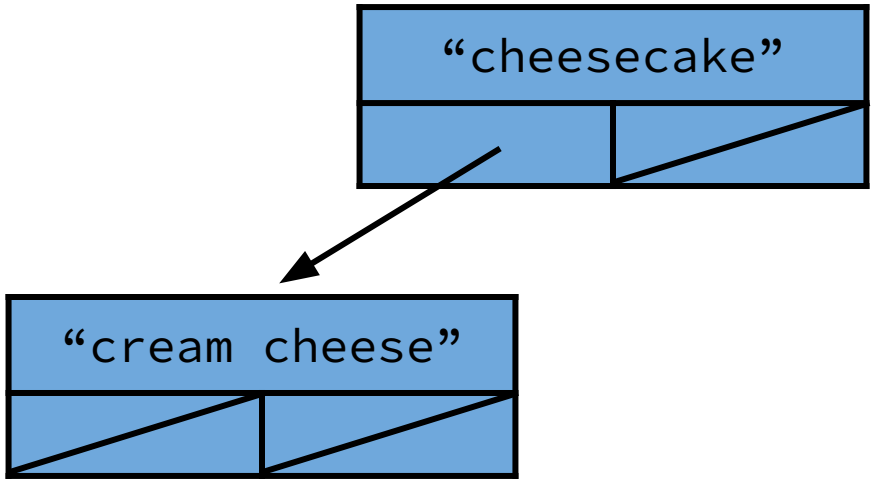
```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
};
```





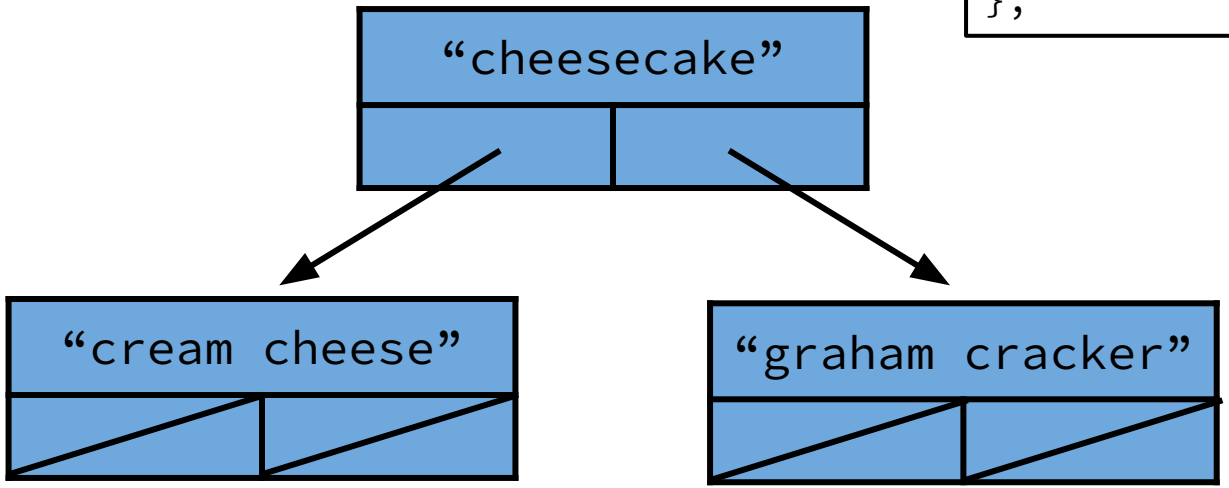
Building Binary Tree

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
};
```



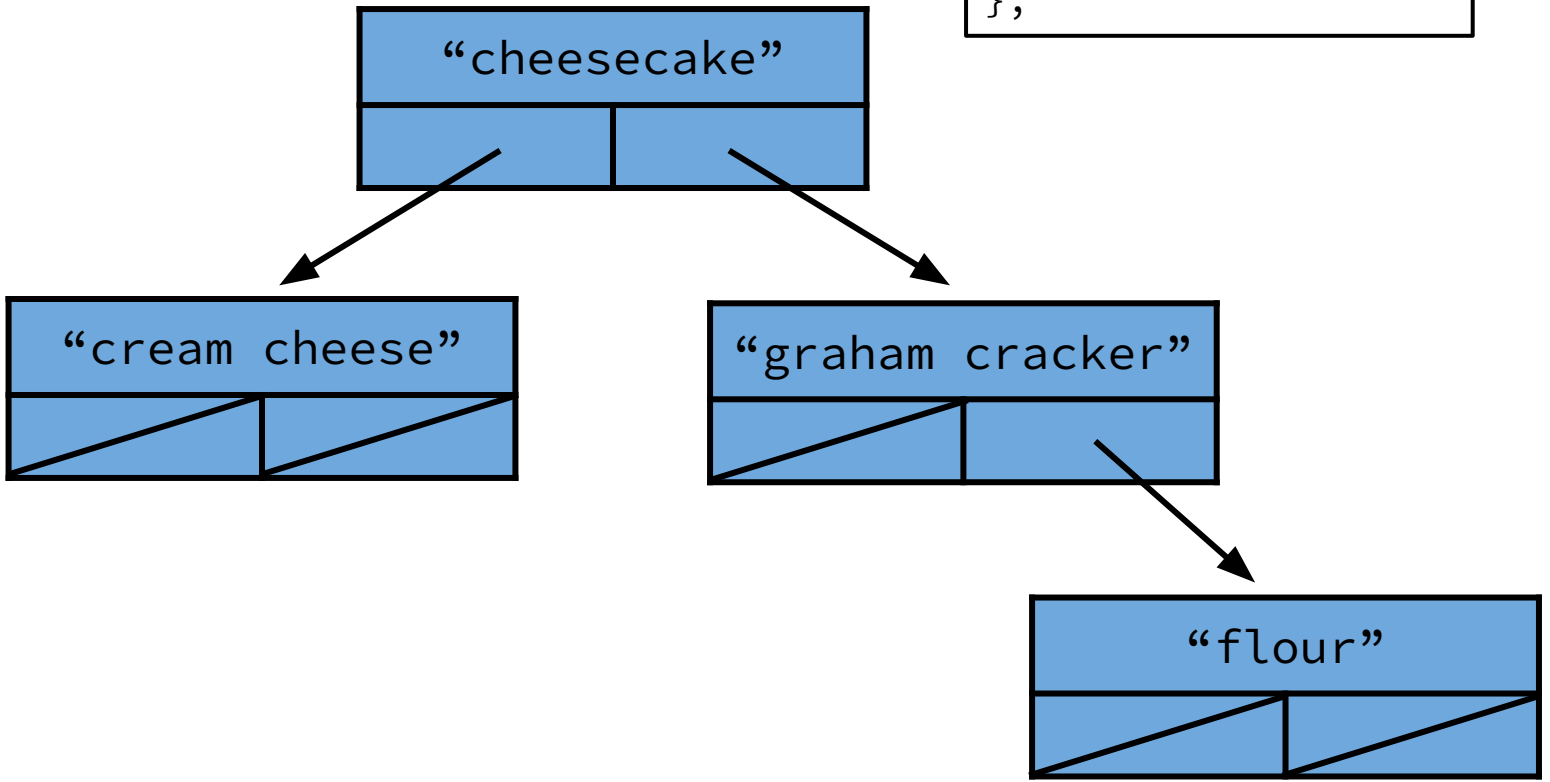
Building Binary Tree

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
};
```



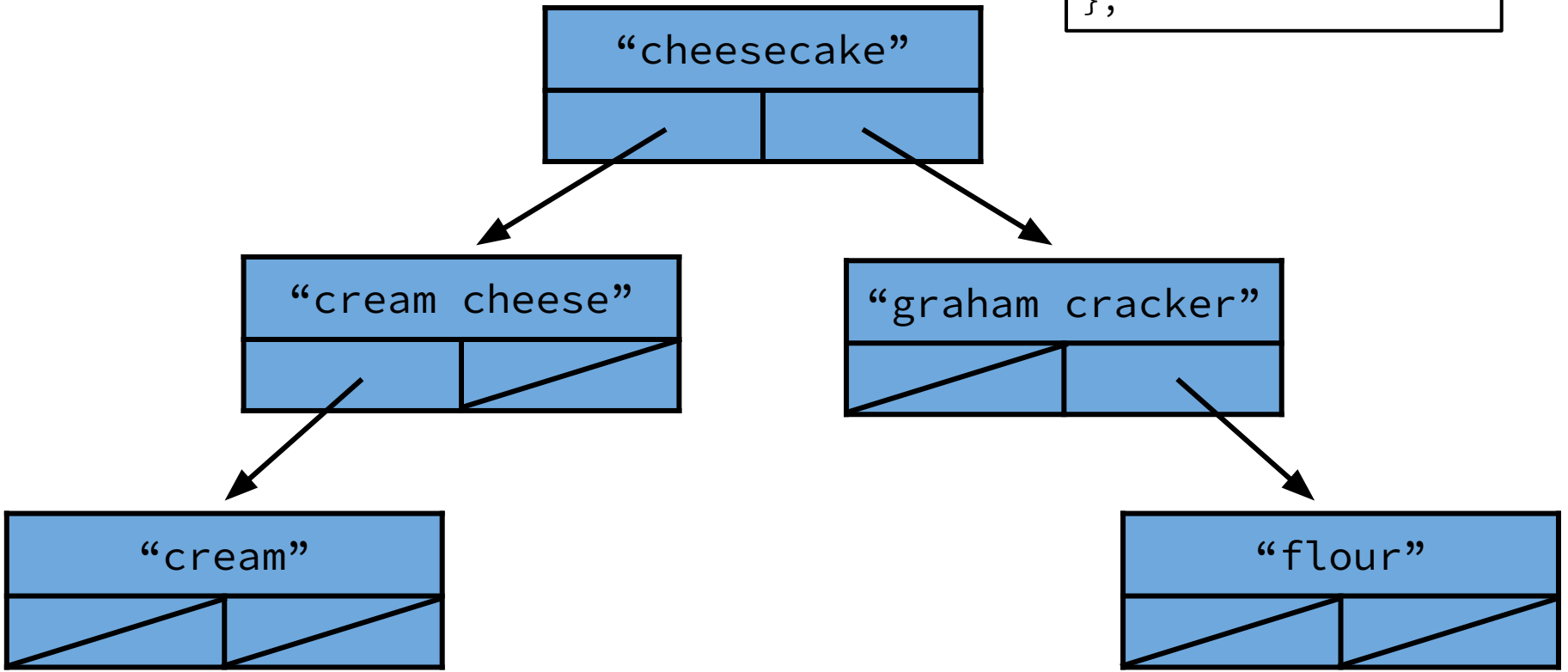
Building Binary Tree

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
};
```



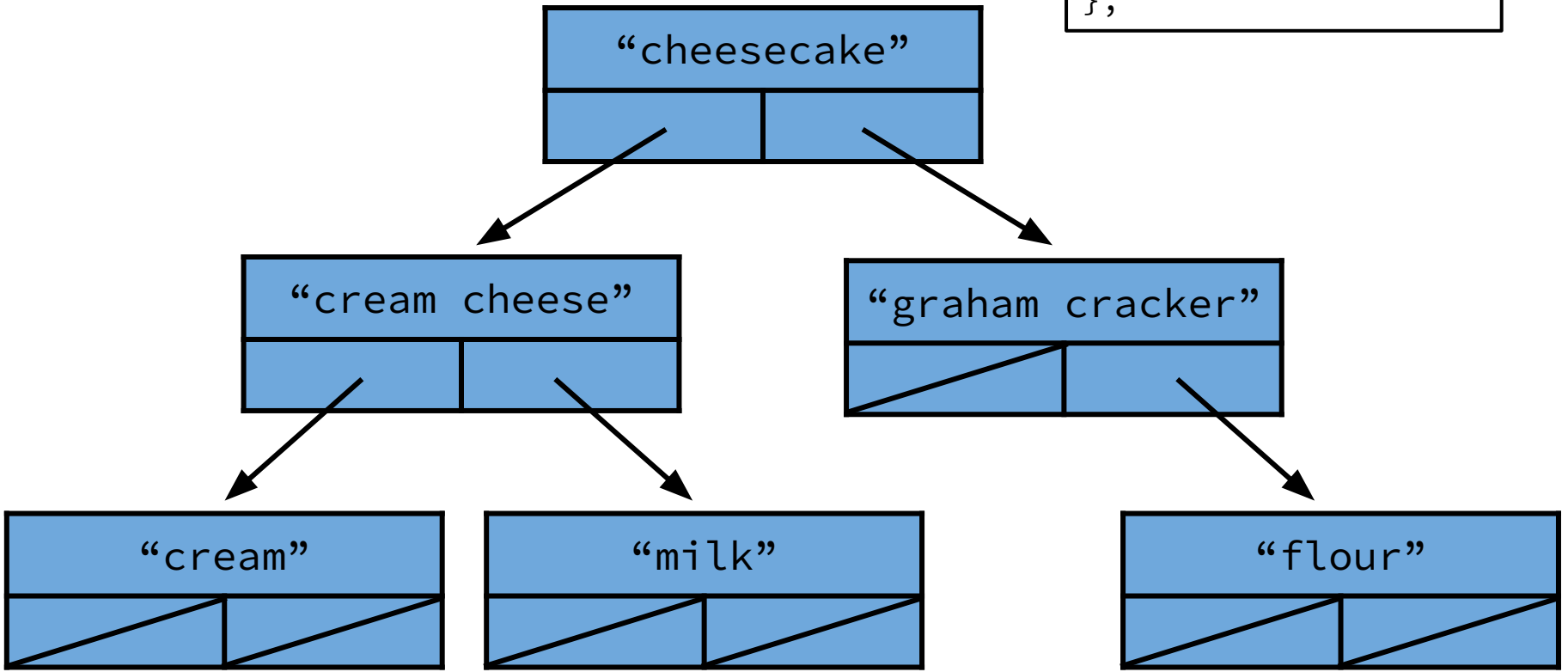
Building Binary Tree

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
};
```



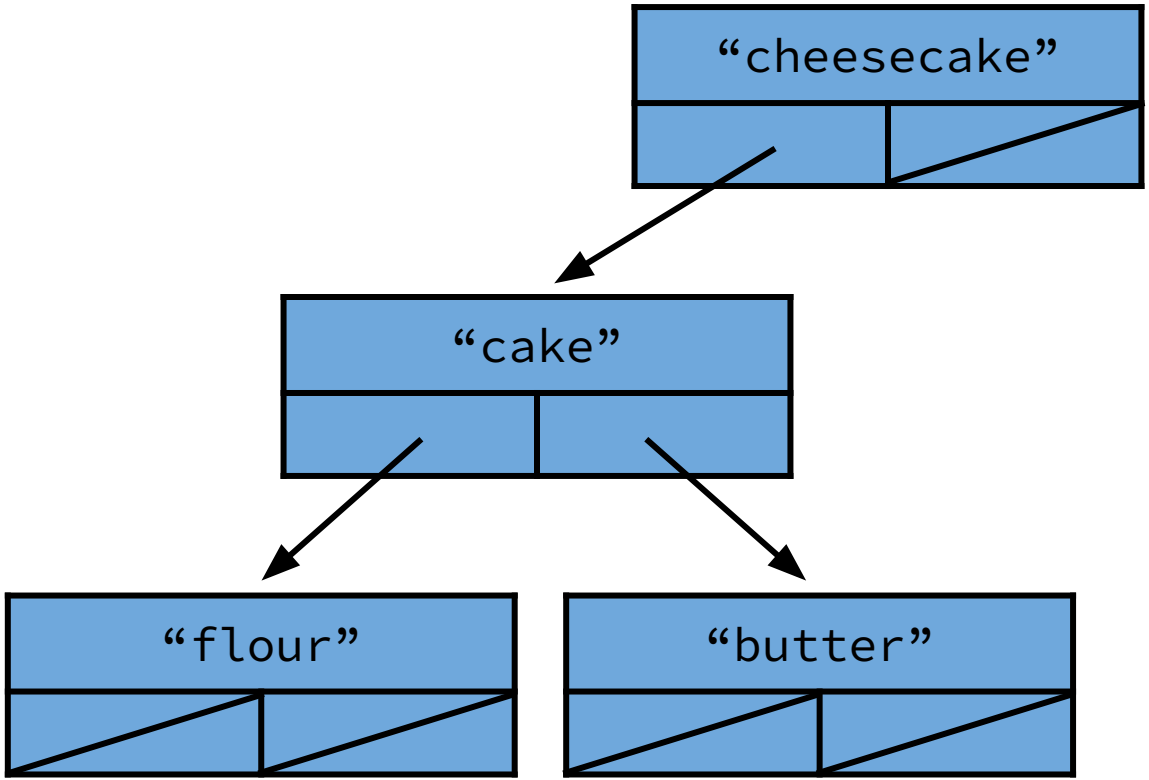
Building Binary Tree

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
};
```



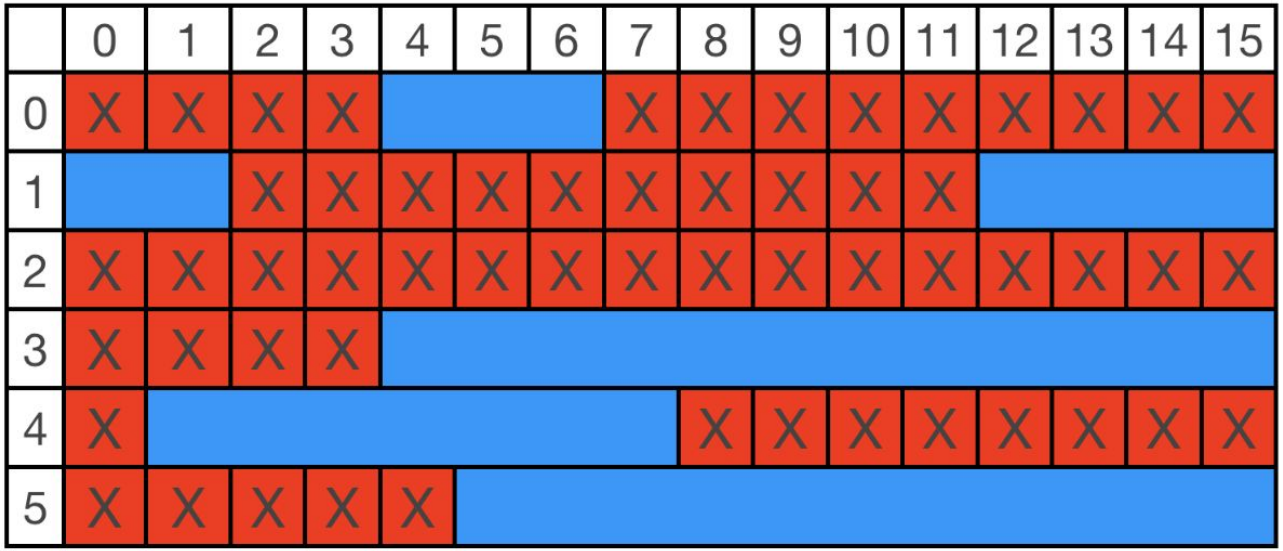
Building Binary Tree

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
};
```



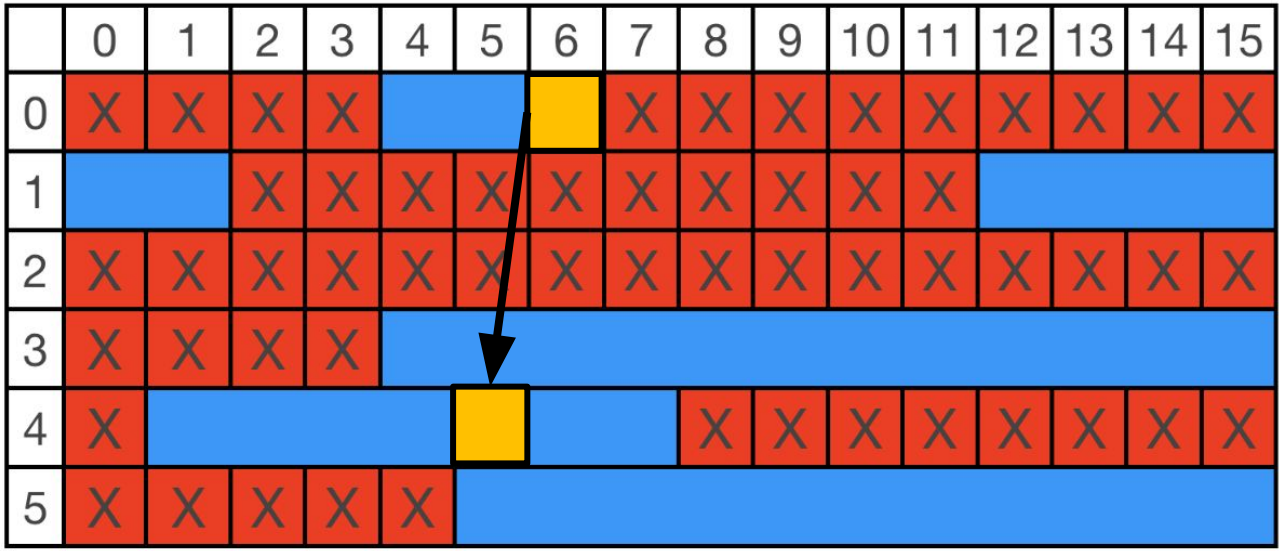
What are Trees?

- A way we can use pointers to organize non-contiguous memory on the heap



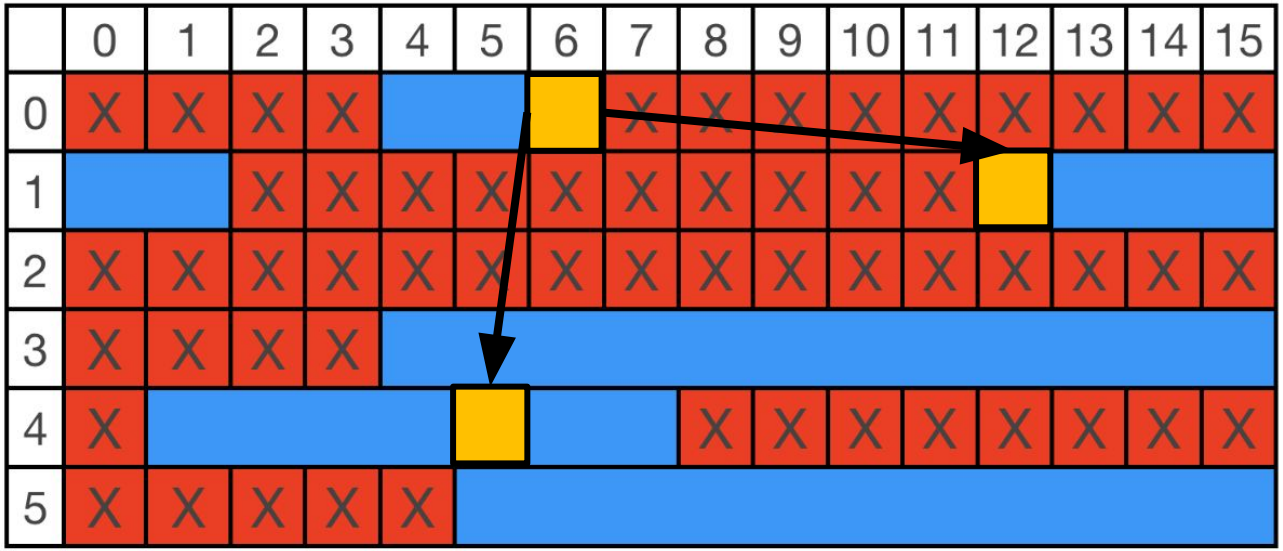
What are Trees?

- A way we can use pointers to organize non-contiguous memory on the heap



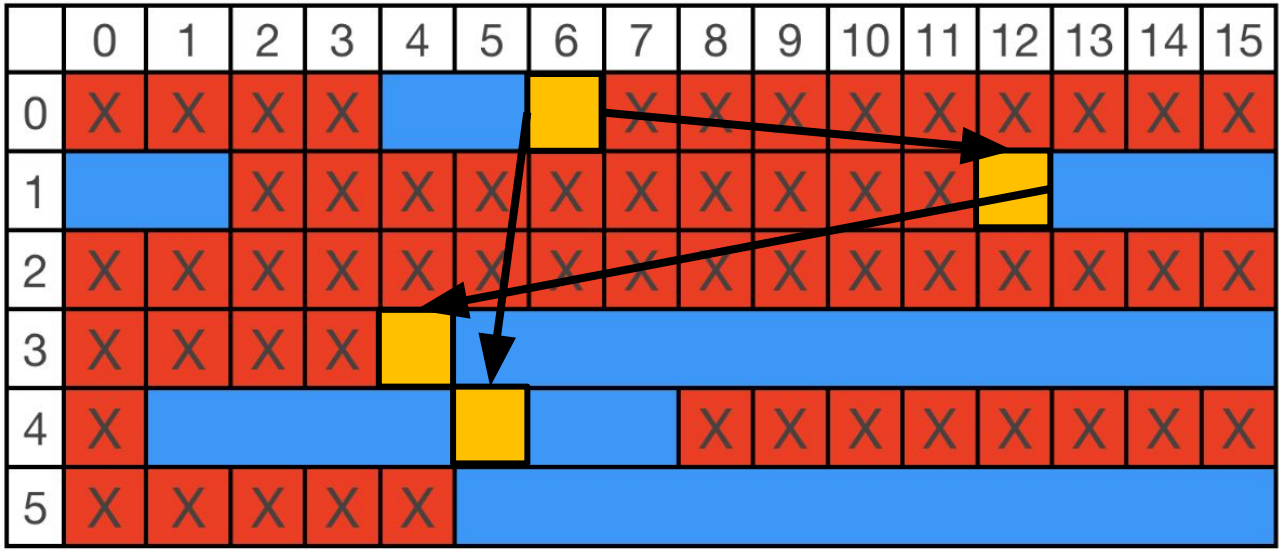
What are Trees?

- A way we can use pointers to organize non-contiguous memory on the heap



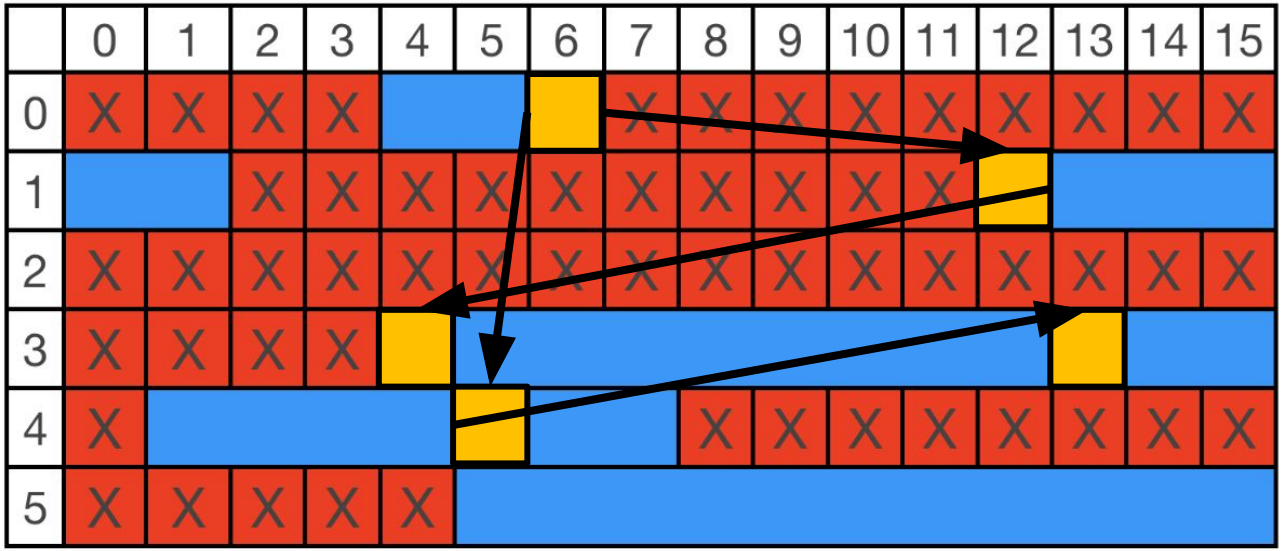
What are Trees?

- A way we can use pointers to organize non-contiguous memory on the heap



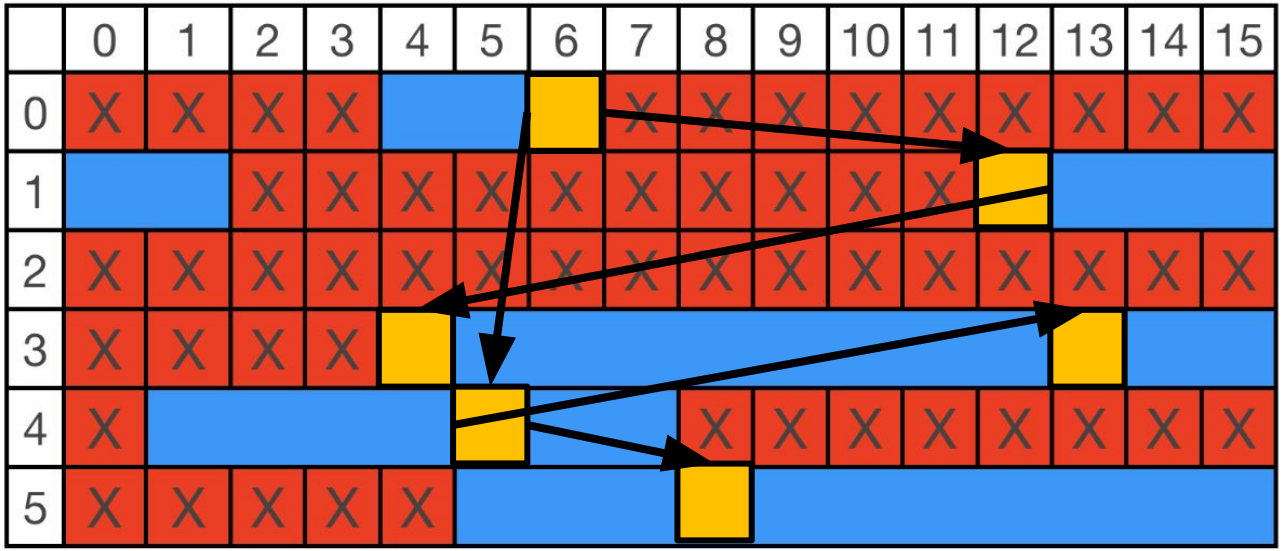
What are Trees?

- A way we can use pointers to organize non-contiguous memory on the heap



What are Trees?

- A way we can use pointers to organize non-contiguous memory on the heap



Let’s Code It Up!

Building a Tree Takeaways

- Building a tree is very similar to building a linked list
- We create new nodes by dynamically allocating memory
- We integrate new nodes into the tree by rewiring the pointers of the existing nodes in the tree

Tree Traversals

Tree Traversals

- If we want to “do something” with each node in the tree, we need to do so by traversing the tree
  - More involved than traversing a linked list because of the branching
- Three main ways to traverse a tree:
  - Pre-order traversal
  - In-order traversal
  - Post-order traversal
- Due to the recursive nature of trees, these algorithms are most easily defined recursively

Pre-Order Traversal

- The algorithm for a pre-order traversal is as follows:
  1. “Do something” with the current node
  2. Traverse the left subtree
  3. Traverse the right subtree

- For our example, let's make the "do something" part print the data at a particular node, which will allow us to print the whole tree

Pre-Order Traversal

```
void preOrderTraversal(TreeNode* tree) {
    if(tree == nullptr) {
        return;
    }
    cout<< tree->data <<" ";
    preOrderTraversal(tree->left);
    preOrderTraversal(tree->right);
}
```

In-Order Traversal

- The algorithm for a in-order traversal is as follows:
  1. Traverse the left subtree
  2. “Do something” with the current node
  3. Traverse the right subtree

In-Order Traversal

```
void inOrderTraversal(TreeNode* tree) {
    if(tree == nullptr) {
        return;
    }
    inOrderTraversal(tree->left);
    cout<< tree->data <<" ";
    inOrderTraversal(tree->right);
}
```

Post-Order Traversal

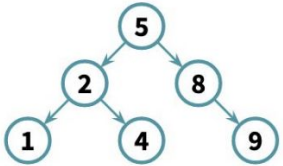
- The algorithm for a post-order traversal is as follows:
  1. Traverse the left subtree
  2. Traverse the right subtree
  3. “Do something” with the current node

Post-Order Traversal

```
void postOrderTraversal(TreeNode* tree) {
    if(tree == nullptr) {
        return;
    }
    postOrderTraversal(tree->left);
    postOrderTraversal(tree->right);
    cout<< tree->data <<" ";
}
```

Tree Traversal Recap

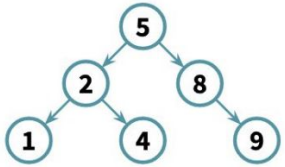
Pre-order



do something (aka cout)
traverse left subtree
traverse right subtree

5 2 1 4 8 9

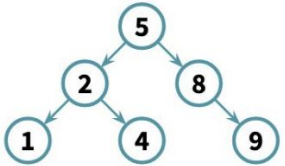
In-order



traverse left subtree
do something (aka cout)
traverse right subtree

1 2 4 5 8 9

Post-order



traverse left subtree
traverse right subtree
do something (aka cout)

1 4 2 9 8 5

Trees Recap

- Allow us to organize information in a linked data structure such that the distance to any element is short, even if there are many elements
  - Added branching, which is so powerful!
- Organize nodes hierarchically, where each element contains connections to child nodes that exist “lower” in the tree
- Three main ways to traverse a tree, and each way visits the nodes in a distinctly different order