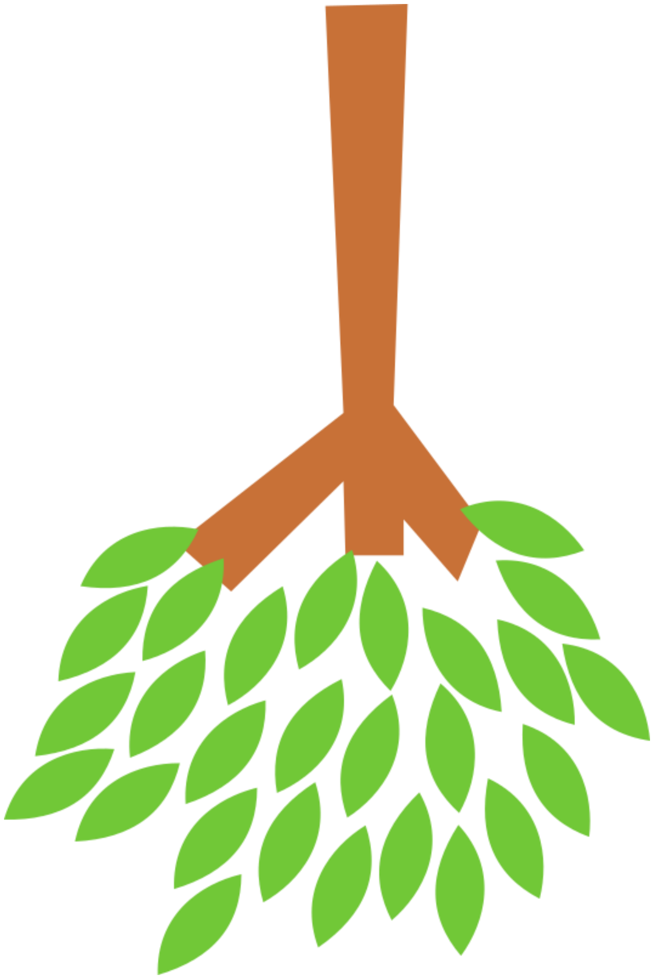
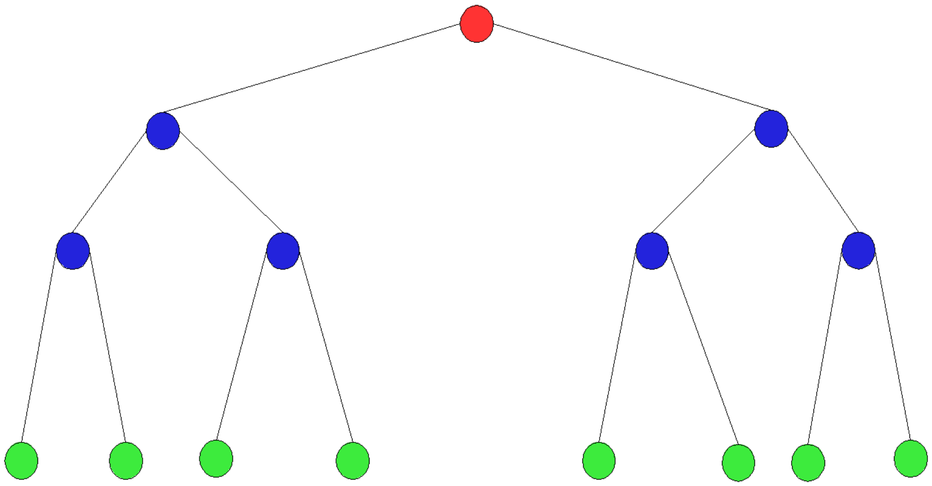




Trees and Binary Trees

Computer Science CS112
Boston University

Christine Papadakis

Motivation: Implementing a Dictionary

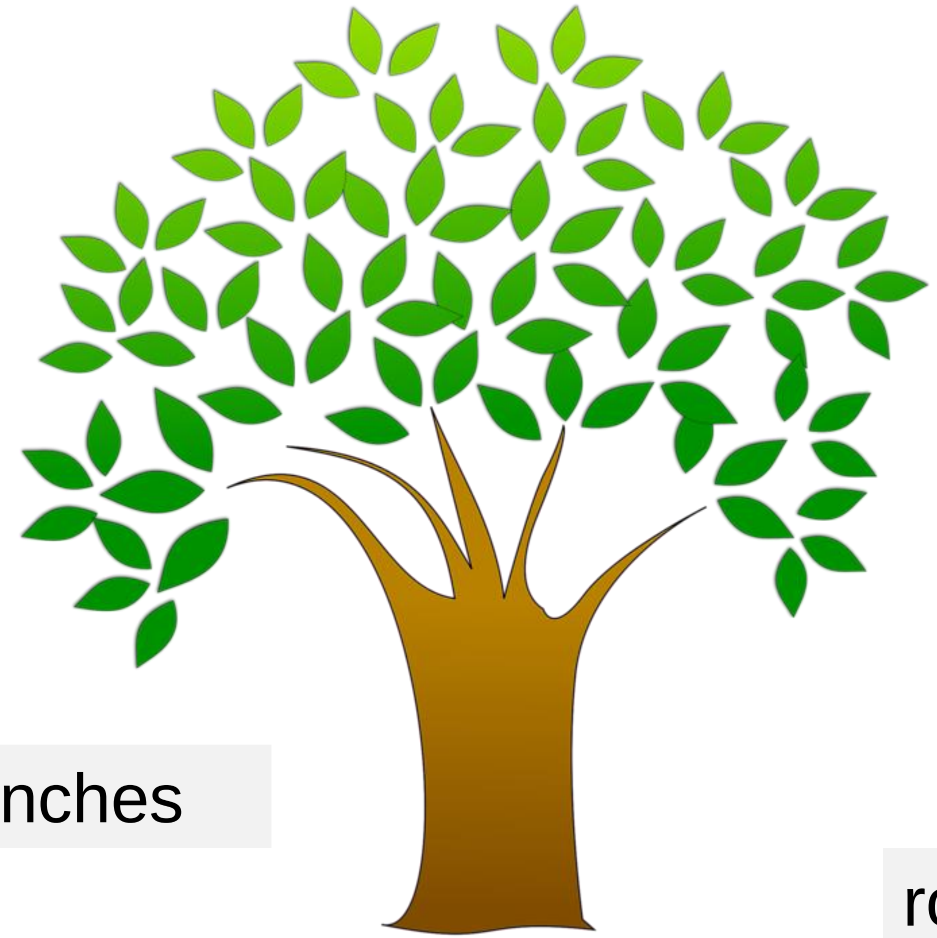
- A *data dictionary* is a collection of data with two main operations:
  - *search* for an item (and possibly delete it)
  - *insert* a new item

- If we use a *sorted* list to implement it, efficiency = O(n).

data structure	***searching for an item***	***inserting an item***
a list implemented using an array	O(n) using sequential search O(log n) using binary search	O(n) because we need to shift items over
a list implemented using a linked list	O(n) using sequential search binary search? O(nlogn)	O(n) (O(1) to do the actual insertion, but O(n) to find where it belongs)

- Can we do better?

Characteristics of a Tree *in Nature*

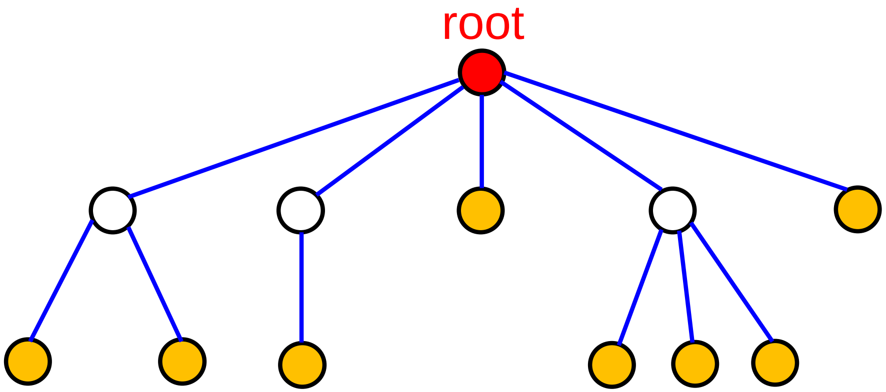


leaves

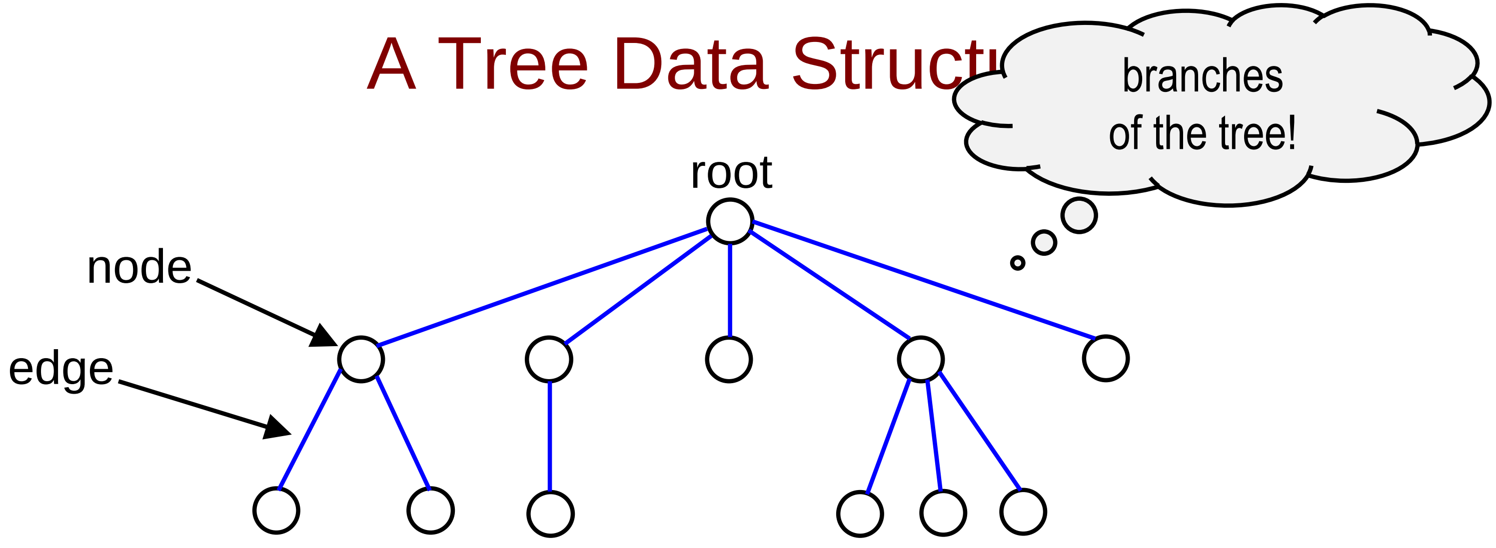
branches

root

A Tree Data Structure

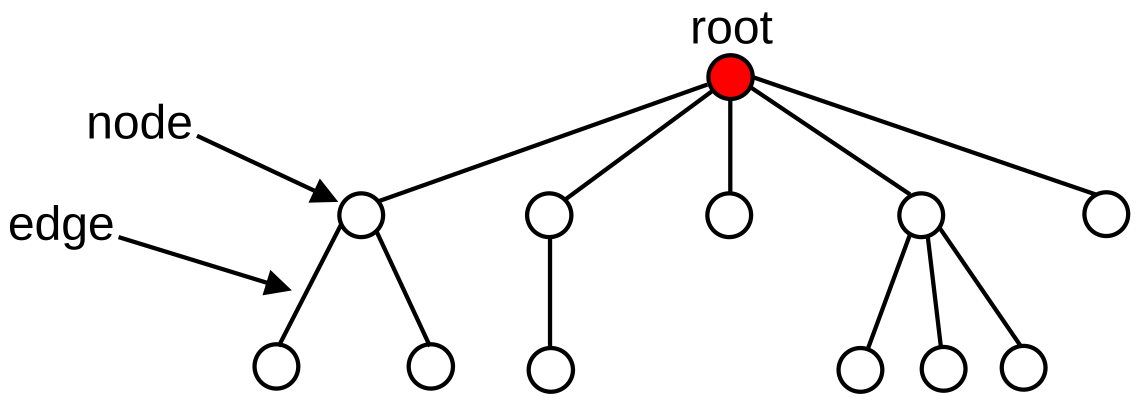


A Tree Data Structure



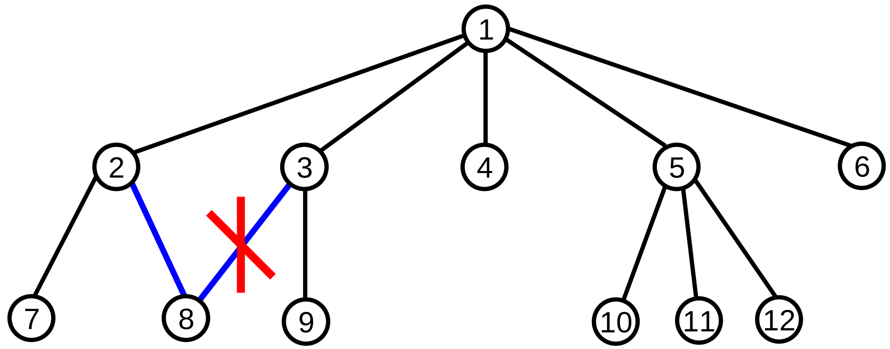
- A tree consists of:
  - a set of *nodes*
  - a set of *edges*, each of which connects a pair of nodes

A Tree Data Structure



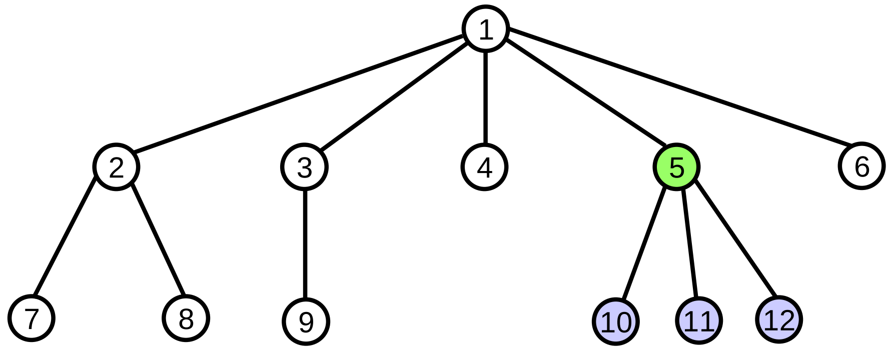
- A tree consists of:
  - a set of *nodes*
  - a set of *edges*, each of which connects a pair of nodes
- Each node may have one or more *data items.*
  - each data item consists of one or more fields
  - *key field* = the field used when searching for a data item
  - multiple data items with the same key are referred to as *duplicates*
- The node at the "top" of the tree is called the *root* of the tree.

Relationships Between Nodes



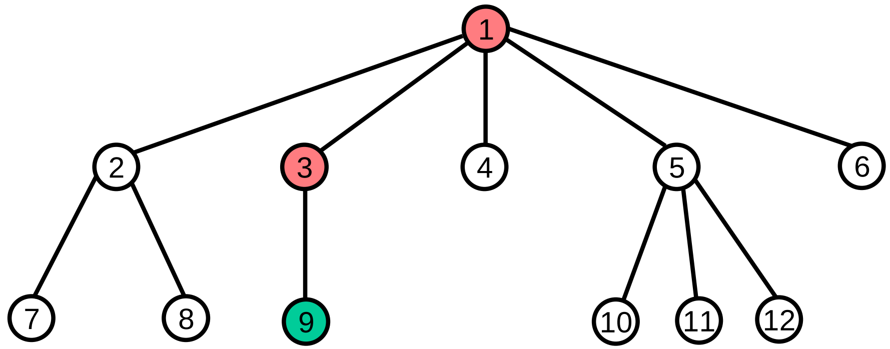
- If a node N is connected to nodes directly below it in the tree:
  - N is referred to as their *parent*
  - they are referred to as its *children*.
    - example: node 5 is the parent of nodes 10, 11, and 12
- Each node is the child of *at most one* parent.

Relationships Between Nodes



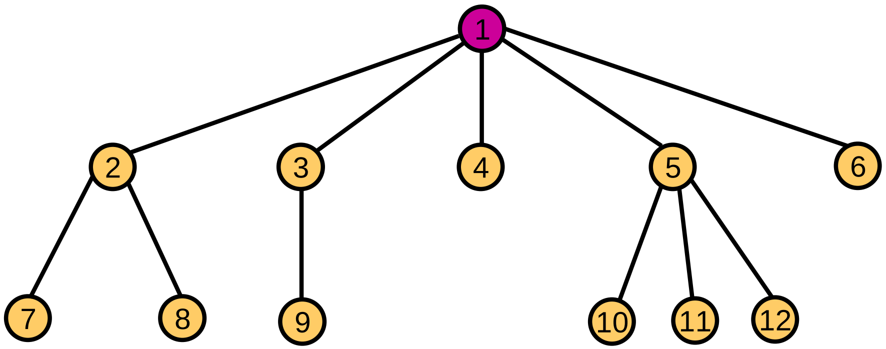
- If a node N is connected to nodes directly below it in the tree:
  - N is referred to as their *parent*
  - they are referred to as its *children*.
    - example: node 5 is the parent of nodes 10, 11, and 12
- Each node is the child of *at most one* parent.
- Other family-related terms are also used:
  - nodes with the same parent are *siblings*

Relationships Between Nodes



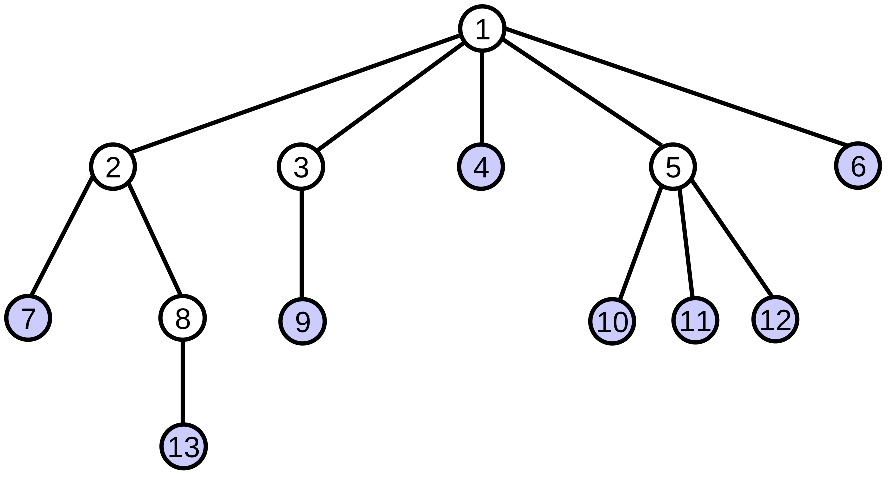
- If a node N is connected to nodes directly below it in the tree:
  - N is referred to as their *parent*
  - they are referred to as its *children*.
    - example: node 5 is the parent of nodes 10, 11, and 12
- Each node is the child of *at most one* parent.
- Other family-related terms are also used:
  - nodes with the same parent are *siblings*
  - a node's *ancestors* are its parent, its parent's parent, etc.
    - example: node 9's ancestors are 3 and 1

Relationships Between Nodes



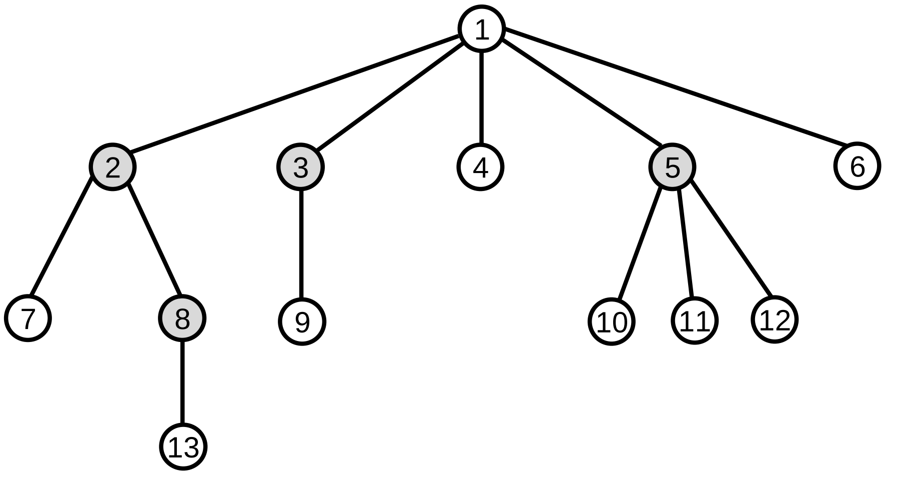
- If a node N is connected to nodes directly below it in the tree:
  - N is referred to as their *parent*
  - they are referred to as its *children*.
    - example: node 5 is the parent of nodes 10, 11, and 12
- Each node is the child of *at most one* parent.
- Other family-related terms are also used:
  - nodes with the same parent are *siblings*
  - a node's *ancestors* are its parent, its parent's parent, etc.
    - example: node 9's ancestors are 3 and 1
  - a node's *descendants* are its children, their children, etc.
    - example: node 1's descendants are *all* of the other nodes

Types of Nodes



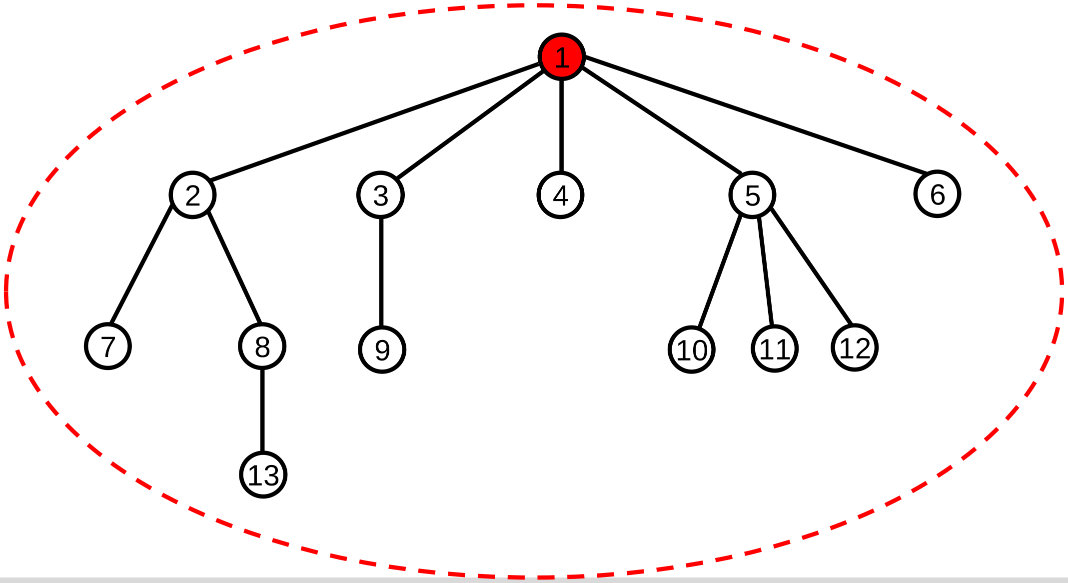
- A *leaf node* is a node without children.
- An *interior node* is a node with one or more children.

Types of Nodes



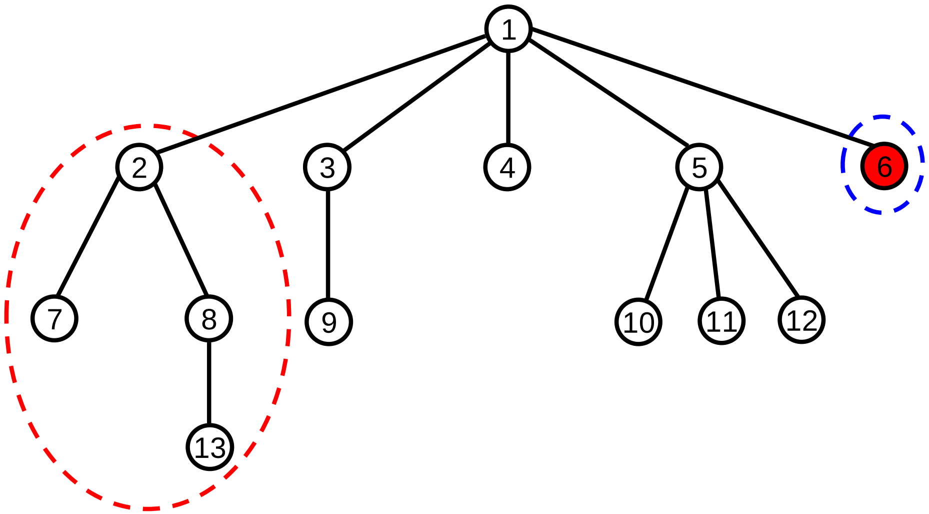
- A *leaf node* is a node without children.
- An *interior node* is a node *from root to leaf* with one or more children.

A Tree is a Recursive Data Structure



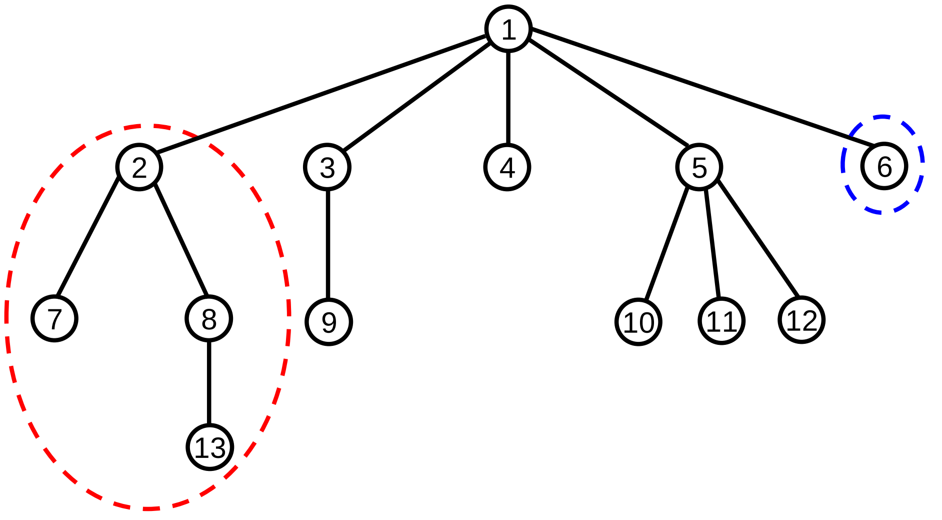
- Each node in the tree is the root of a smaller tree!
  - refer to such trees as *subtrees* to distinguish them from the tree as a whole

A Tree is a Recursive Data Structure



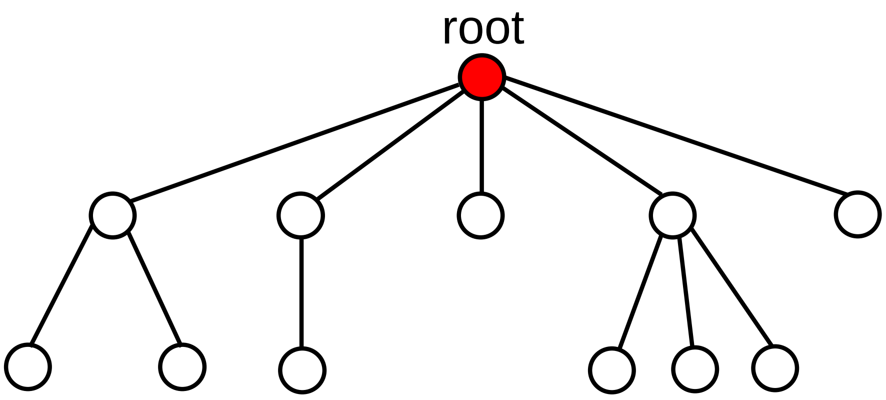
- Each node in the tree is the root of a smaller tree!
  - refer to such trees as *subtrees* to distinguish them from the tree as a whole
  - example: node 2 is the root of the subtree circled above
  - example: node 6 is the root of a subtree with only one node

A Tree is a Recursive Data Structure



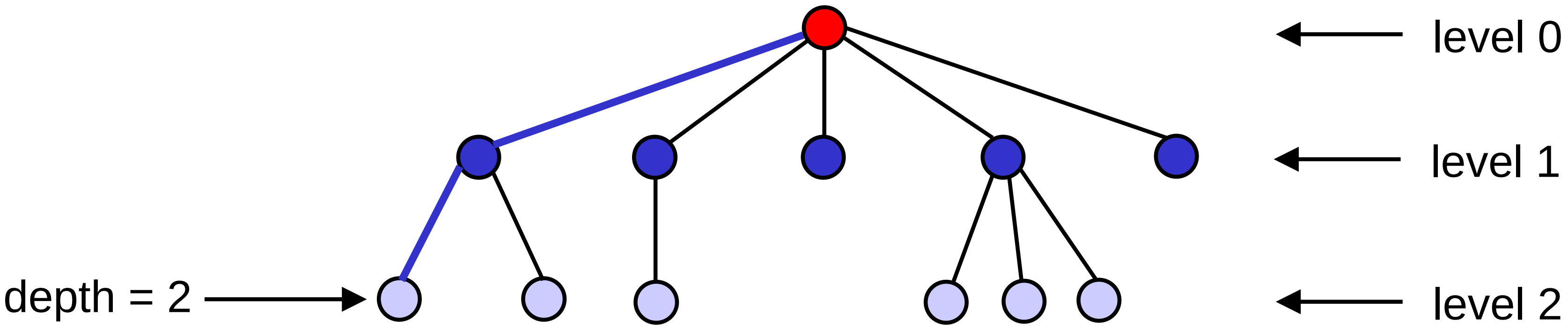
- Each node in the tree is the root of a smaller tree!
  - refer to such trees as *subtrees* to distinguish them from the tree as a whole
  - example: node 2 is the root of the subtree circled above
  - example: node 6 is the root of a subtree with only one node
- We'll see that tree algorithms often lend themselves to recursive implementations.

Path, Depth, Level, and Height



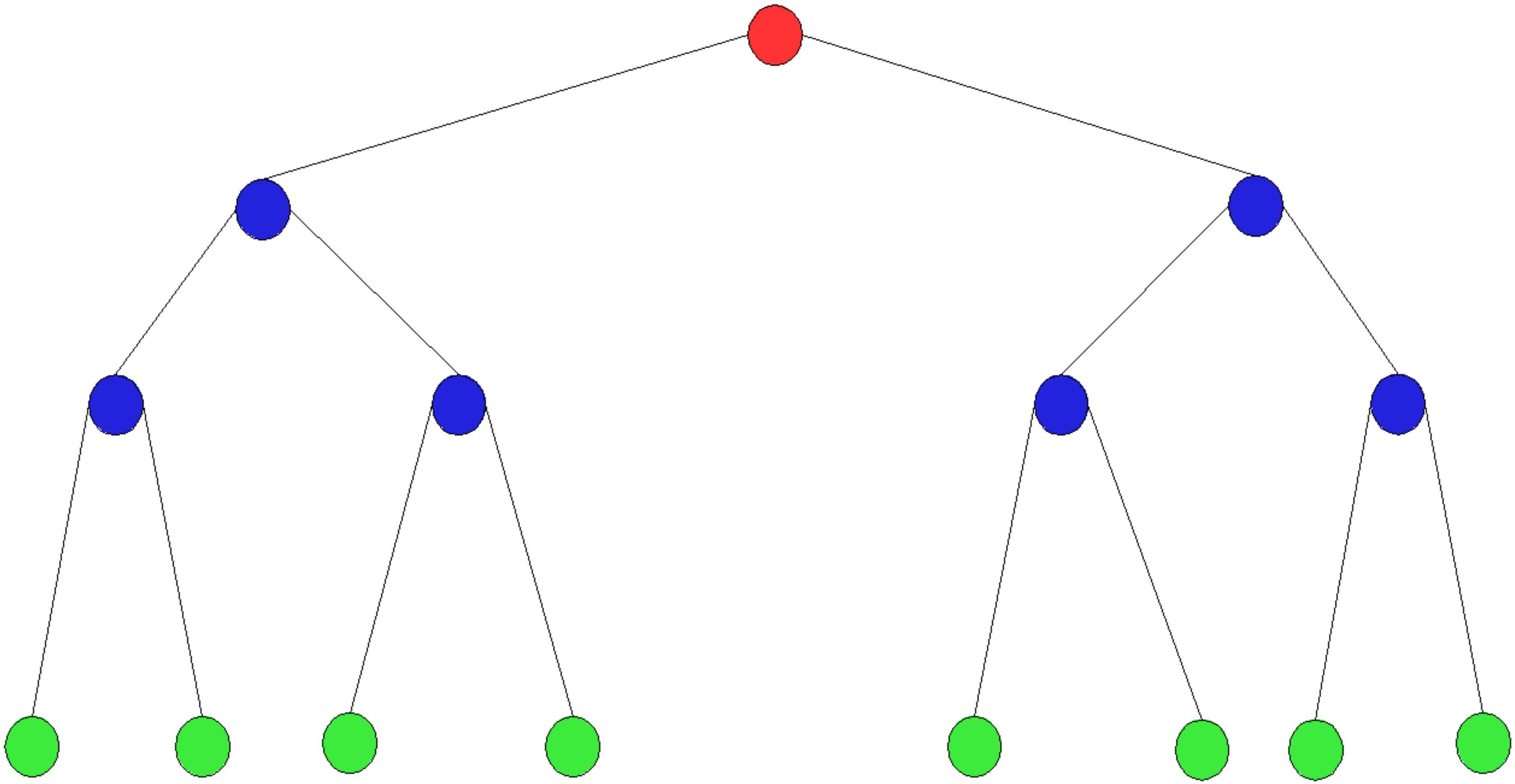
- The *path* is the sequence of edges connecting each node to the root.

Path, Depth, Level, and Height



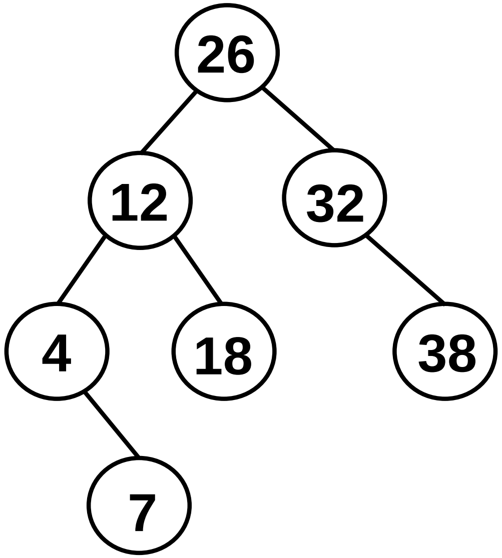
- There is exactly one *path* (one sequence of edges) connecting each node to the root.
- *depth* of a node = # of edges on the path from it to the root
- Nodes with the same depth form a *level* of the tree.
- The *height* of a tree is the maximum depth of its nodes.
  - example: the tree above has a height of 2

Binary Tree



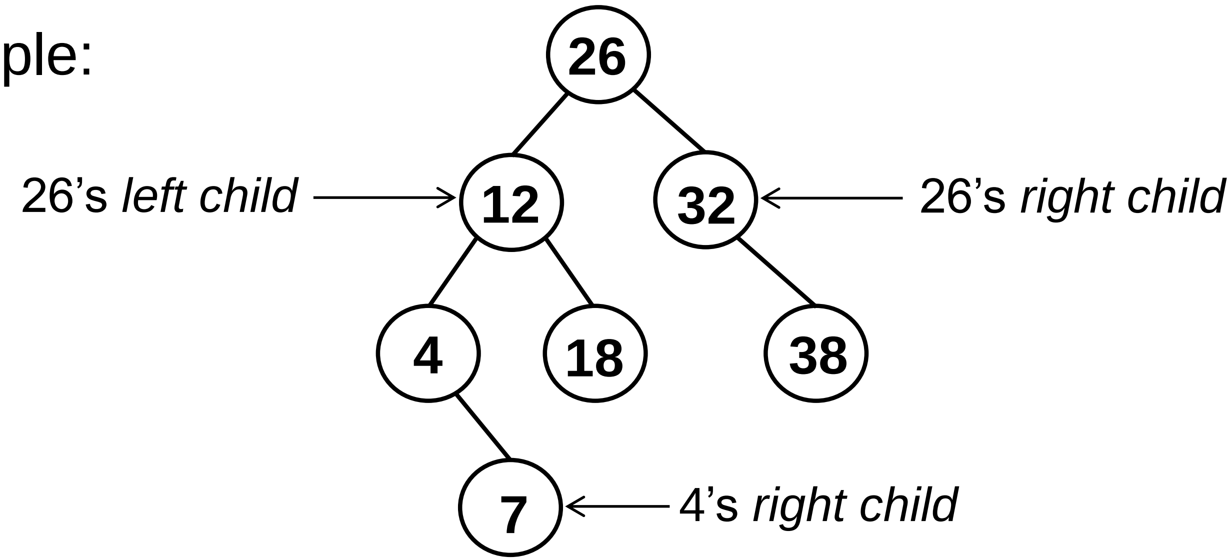
Binary Trees

- In a *binary tree*, nodes have *at most two* children.
  - distinguish between them using the direction *left* or *right*
- Example:



Binary Trees

- In a *binary tree*, nodes have *at most two* children.
  - distinguish between them using the direction *left* or *right*
- Example:

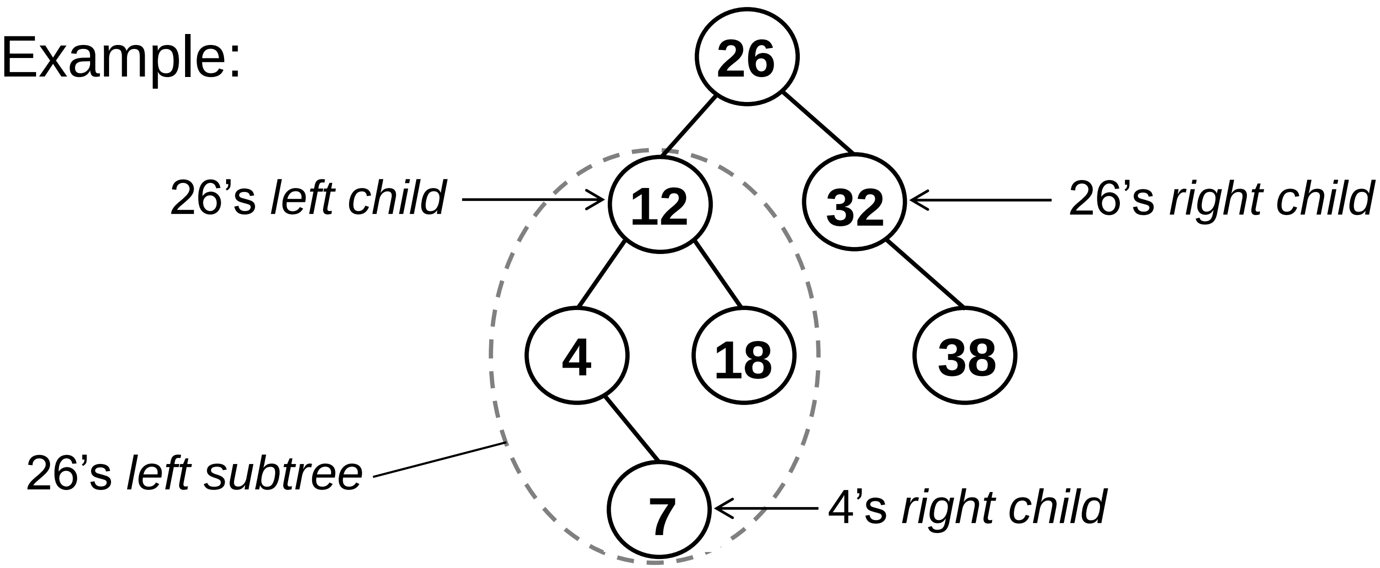


- Recursive definition: a binary tree is either:
  1) empty, or
  2) a node (the root of the tree) that has:
     - one or more pieces of data (the key, and possibly others)
     - a *left subtree*, which is itself a binary tree
     - a *right subtree*, which is itself a binary tree

Binary Trees

- In a *binary tree*, nodes have *at most two* children.
  - distinguish between them using the direction *left* or *right*

- Example:



- Recursive definition: a binary tree is either:

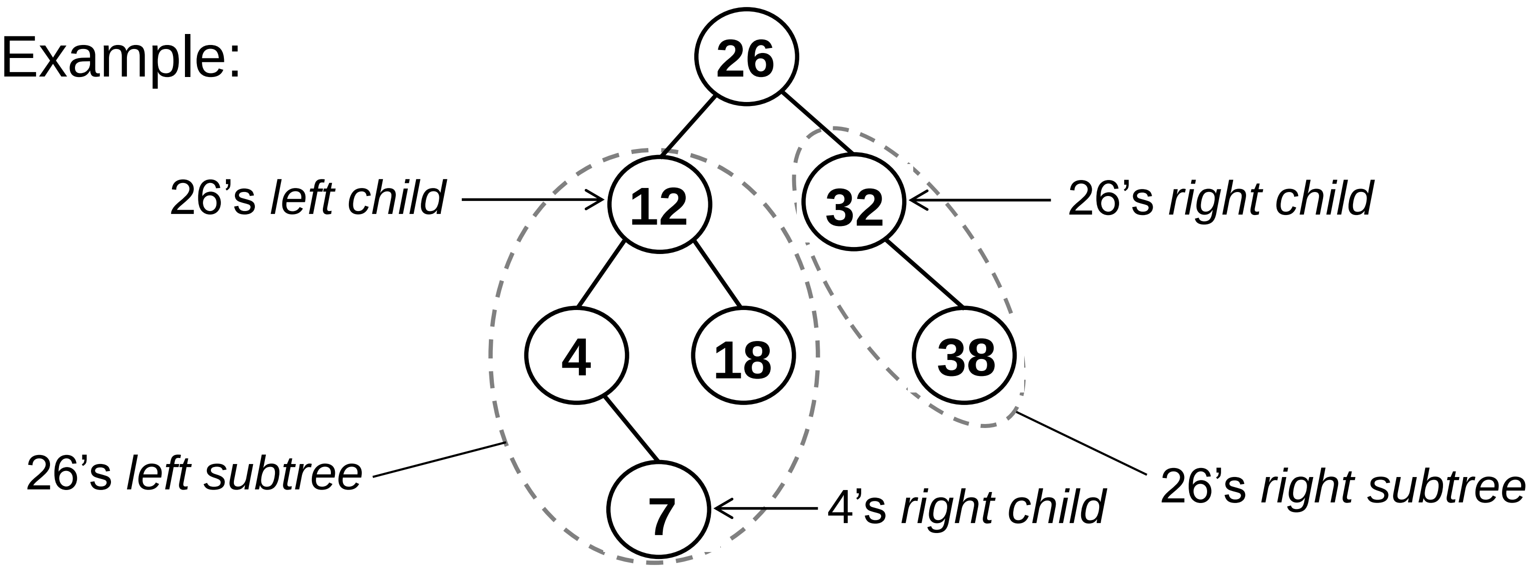
  1) empty, or

  2) a node (the root of the tree) that has:
     - one or more pieces of data (the key, and possibly others)
     - a *left subtree*, which is itself a binary tree
     - a *right subtree*, which is itself a binary tree

Binary Trees

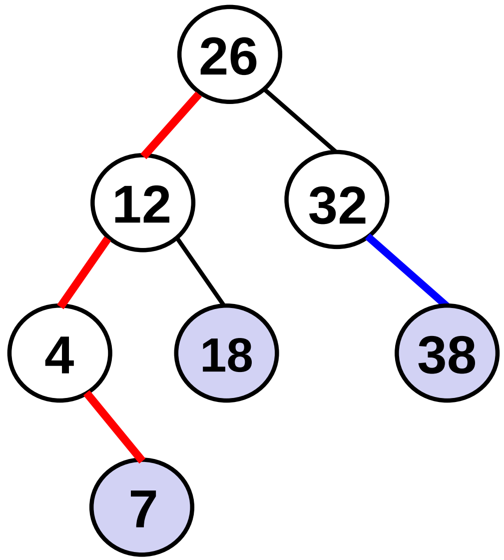
- In a *binary tree*, nodes have *at most two* children.
  - distinguish between them using the direction *left* or *right*

- Example:



- Recursive definition: a binary tree is either:
  1) empty, or
  2) a node (the root of the tree) that has:
     - one or more pieces of data (the key, and possibly others)
     - a *left subtree*, which is itself a binary tree
     - a *right subtree*, which is itself a binary tree

Which of the following is/are <u>not</u> true?



A. ~~This tree has a height of 4.~~ *It has a height of 3.*

B. There are 3 leaf nodes.

C. The 38 node is the right child of the 32 node.

D. ~~The 12 node has 3 children.~~ 12 has two children (4 and 18). It has three descendants (4, 18, and 7).

E. more than one of the above are <u>not</u> true (A and D)

Representing a Binary Tree Using Linked Nodes:

for a data dictionary

```
public class LinkedTree {
    private class Node {




    }


    …
}
```

Assumptions:

1) keys are integers
2) a key can have more than one value associated with it.

Representing a Binary Tree Using Linked Nodes:

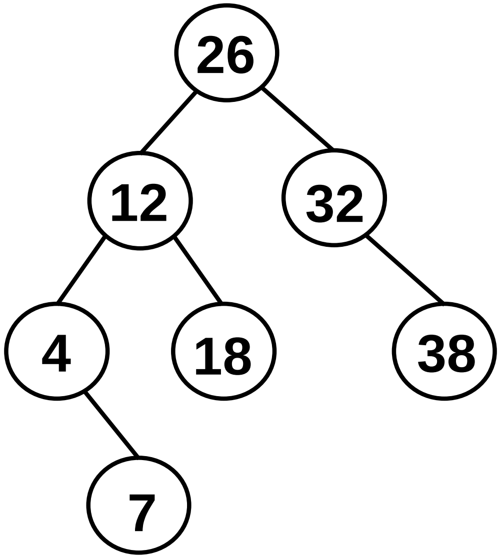
for a data dictionary

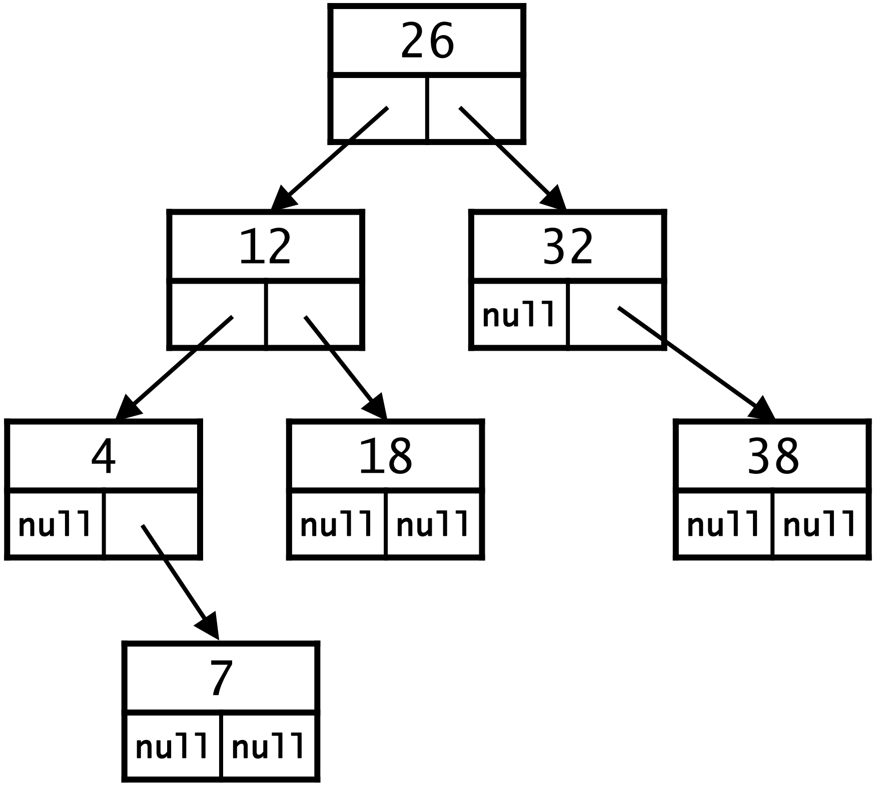
```
public class LinkedTree {
    private class Node {
        // key;
        // data value(s) associated with the key;
        // reference to left child;
        // reference to right child;
        …
    }

    // reference to first node in the tree;
    …
}
```

Representing a Binary Tree Using Linked Nodes:

for a data dictionary

```
public class LinkedTree {
    private class Node {
        private int key;          // limit ourselves to int keys
        private LLList data;      // list of data for that key
        private Node left;        // reference to left child
        private Node right;       // reference to right child
        …
    }

    private Node root;
    …
}
```

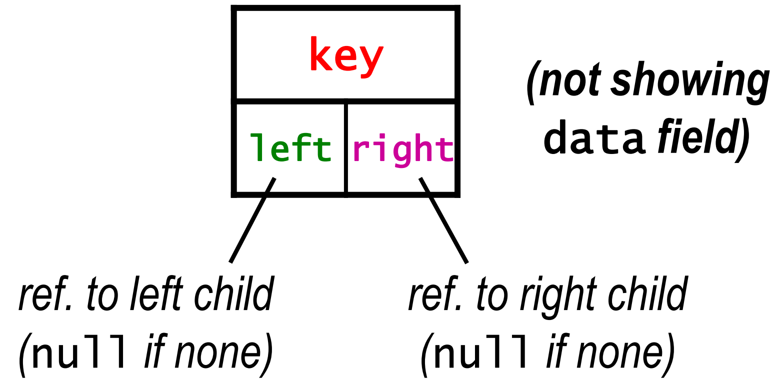


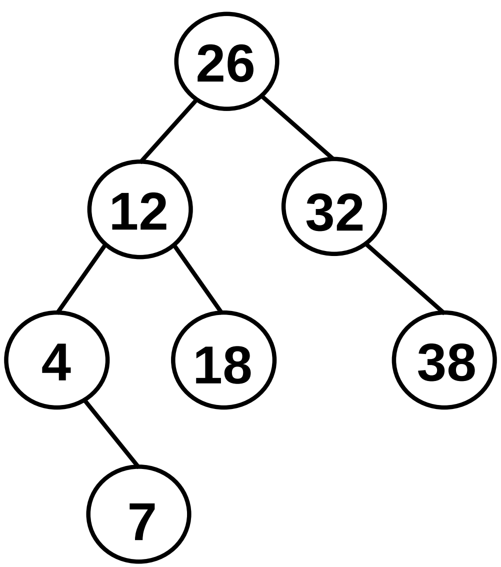


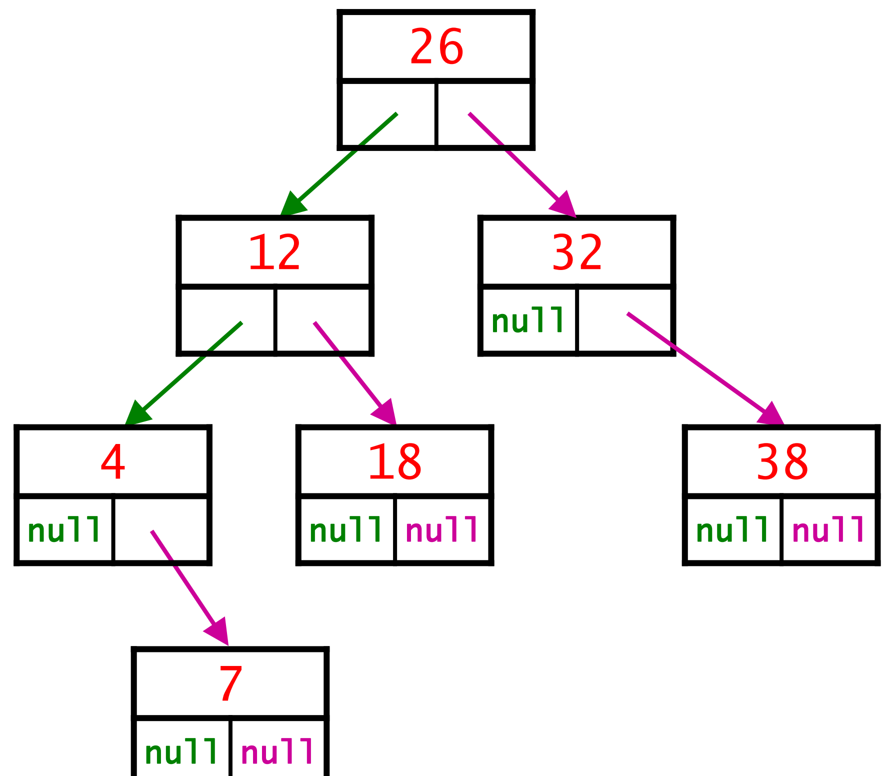
Representing a Binary Tree Using Linked Nodes:

for a data dictionary

```
public class LinkedTree {
    private class Node {
        private int key;
        private LLList data;
        private Node left;
        private Node right;
        …
    }

    private Node root;
    …
}
```

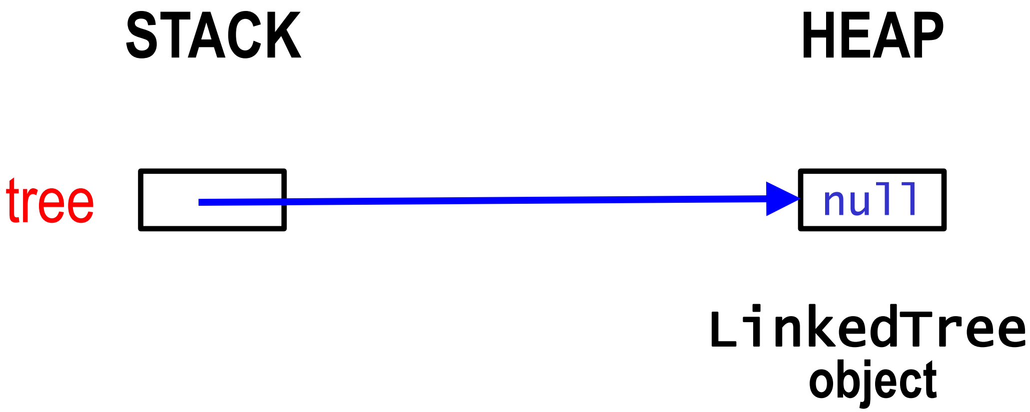






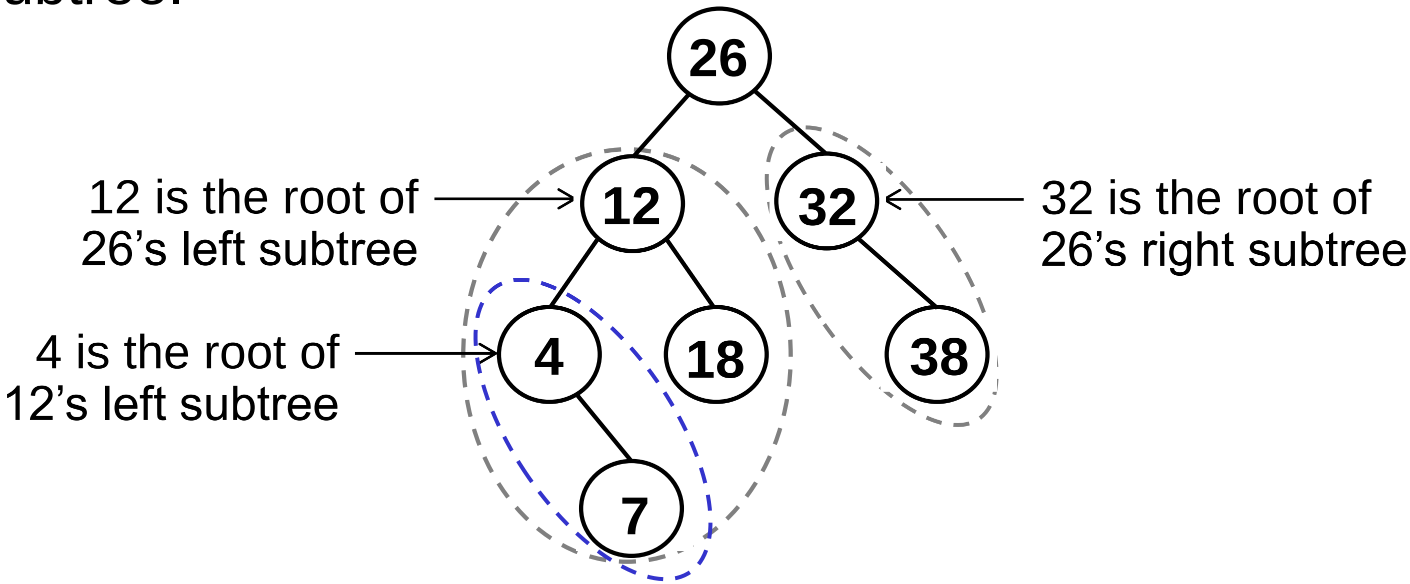
An instance of LinkedTree

- `LinkedTree tree = new LinkedTree();`



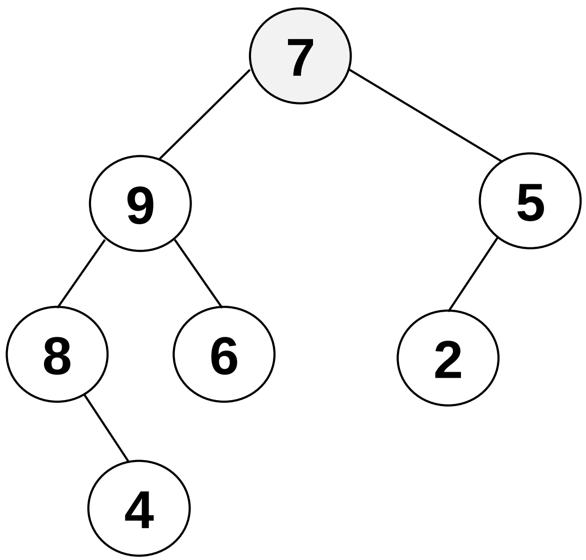
Traversing a Binary Tree

- Traversing a tree involves *visiting* all of the nodes in the tree.
  - visiting a node = processing its data in some way
    - example: print the key, perform a comparison on the key, etc.
- We will look at four types of traversals. Each of them visits the nodes in a different order.
- To understand traversals, it helps to remember the recursive definition of a binary tree, in which every node is the root of a subtree.



Preorder Traversal

- preorder traversal of the tree whose root is N:
  1) visit the root, N
  2) recursively perform a preorder traversal of N's left subtree
  3) recursively perform a preorder traversal of N's right subtree



- Preorder traversal of the tree above:

  7 9 8 4 6 5 2

Implementing Preorder Traversal

```
public class LinkedTree {
    ...
    private Node root;

    public void preorderPrint() {
        if (root != null) {
            preorderPrintTree(root);
        }
    }
    private static void preorderPrintTree(Node root) {
        System.out.print(root.key + " ");
        if (root.left != null) {
            preorderPrintTree(root.left);
        }
        if (root.right != null) {
            preorderPrintTree(root.right);
        }
    }
```

- `preorderPrintTree()` is a static, recursive method that takes as a parameter the root of the tree/subtree that you want to print.

Implementing Preorder Traversal

```
public class LinkedTree {
    ...
    private Node root;

    public void preorderPrint() {
        if (root != null) {
            preorderPrintTree(root);
        }
    }
    private static void preorderPrintTree(Node root) {
        System.out.print(root.key + " ");
        if (root.left != null) {
            preorderPrintTree(root.left);
        }
        if (root.right != null) {
            preorderPrintTree(root.right);
        }
    }
```

- `preorderPrintTree()` is a static, recursive method that takes as a parameter the root of the tree/subtree that you want to print.

Implementing Preorder Traversal

```
public class LinkedTree {
    ...
    private Node root;

    public void preorderPrint() {
        if (root != null) {
            preorderPrintTree(root);
        }
    }
    private static void preorderPrintTree(Node root) {
        System.out.print(root.key + " ");
        if (root.left != null) {
            preorderPrintTree(root.left);
        }
        if (root.right != null) {
            preorderPrintTree(root.right);
        }
    }
```

Not always the same as the root of the entire tree.

- `preorderPrintTree()` is a static, recursive method that takes the root of the tree/subtree that you want to print.

Implementing Preorder Tra

Why is this method necessary?

```
public class LinkedTree {
    ...
    private Node root;

    public void preorderPrint() {
        if (root != null) {
            preorderPrintTree(root);
        }
    }
    private static void preorderPrintTree(Node root) {
        System.out.print(root.key + " ");
        if (root.left != null) {
            preorderPrintTree(root.left);
        }
        if (root.right != null) {
            preorderPrintTree(root.right);
        }
    }
```

- `preorderPrintTree()` is a static, recursive method that takes the root of the tree/subtree that you want to print.
- `preorderPrint()` is a non-static "wrapper" method that makes the initial call. It passes in the root of the entire tree.

Implementing Preorder Tra

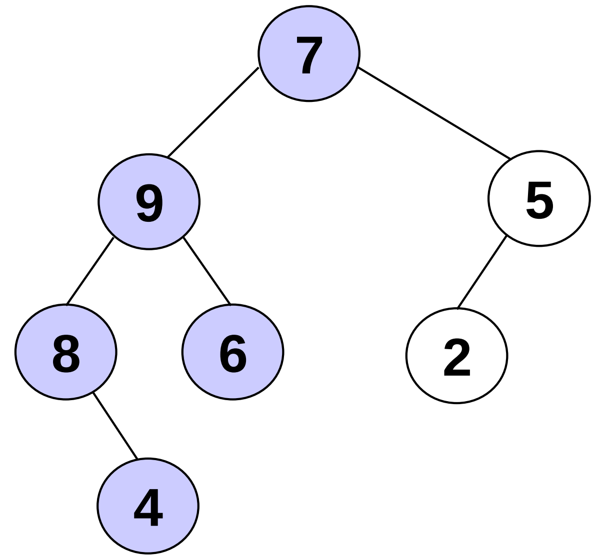
To provide access to the root node of the tree!

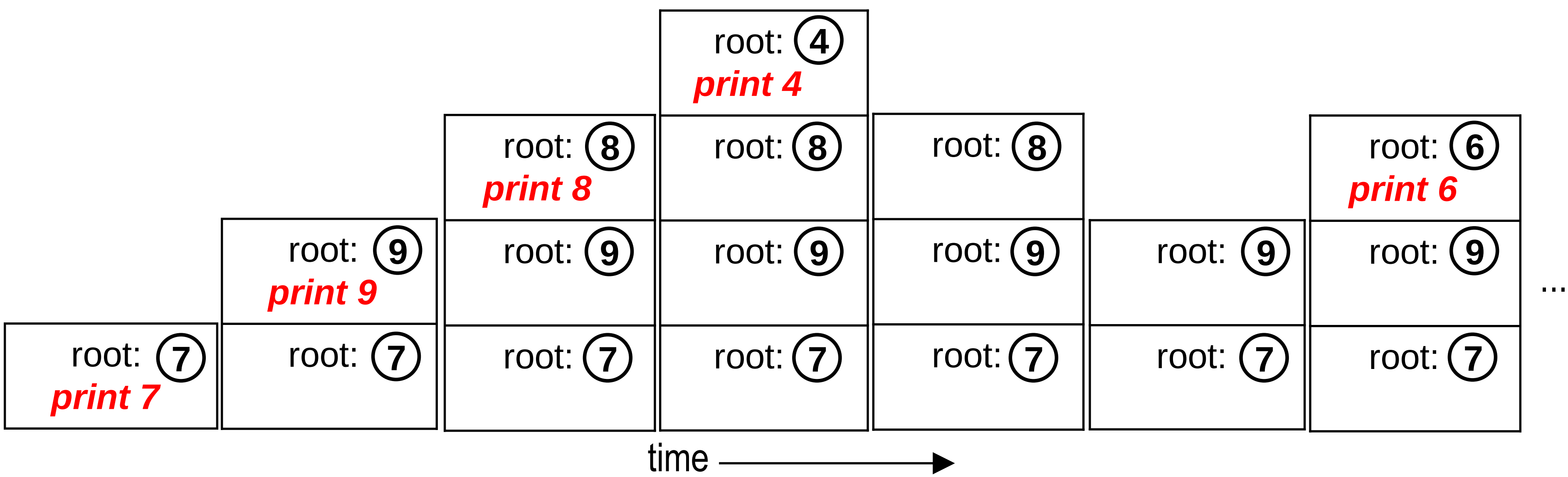
```
public class LinkedTree {
    ...
    private Node root;

    public void preorderPrint() {
        if (root != null) {
            preorderPrintTree(root);
        }
    }
    private static void preorderPrintTree(Node root) {
        System.out.print(root.key + " ");
        if (root.left != null) {
            preorderPrintTree(root.left);
        }
        if (root.right != null) {
            preorderPrintTree(root.right);
        }
    }
```

- `preorderPrintTree()` is a static, recursive method that takes the root of the tree/subtree that you want to print.
- `preorderPrint()` is a non-static "wrapper" method that makes the initial call. It passes in the root of the entire tree.

Tracing Preorder Traversal

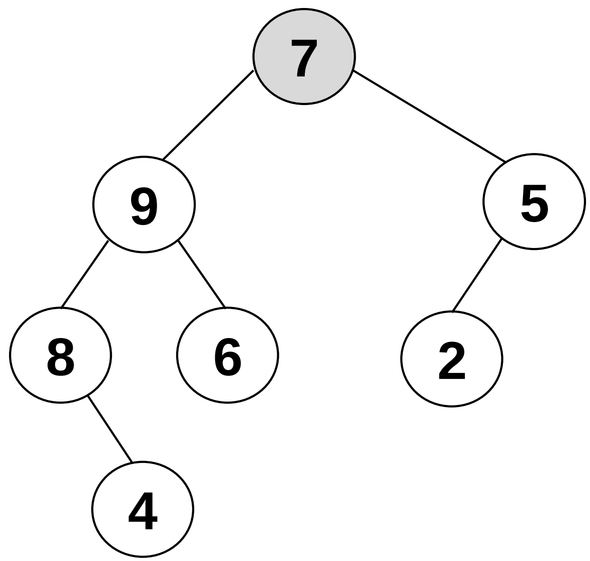
```
void preorderPrintTree(Node root) {
   System.out.print(root.key + " ");
   if (root.left != null) {
       preorderPrintTree(root.left);
   }
   if (root.right != null) {
       preorderPrintTree(root.right);
   }
}
```





Postorder Traversal

- postorder traversal of the tree whose root is N:
  1) recursively perform a postorder traversal of N's left subtree
  2) recursively perform a postorder traversal of N's right subtree
  3) **visit the root, N**



- Postorder traversal of the tree above:

  4 8 6 9 2 5 7

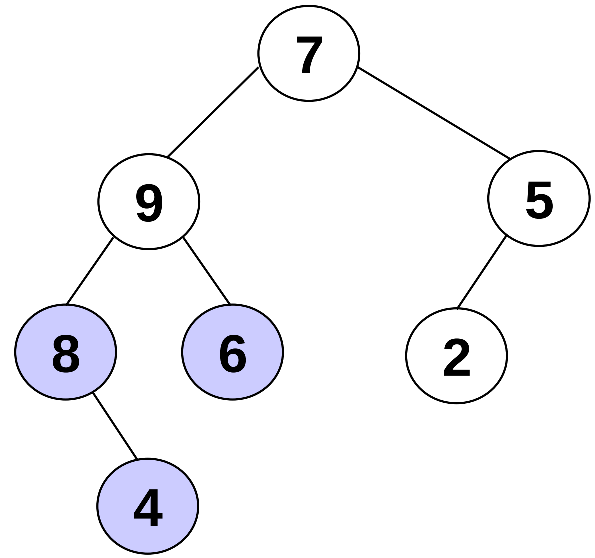
Implementing Postorder Traversal

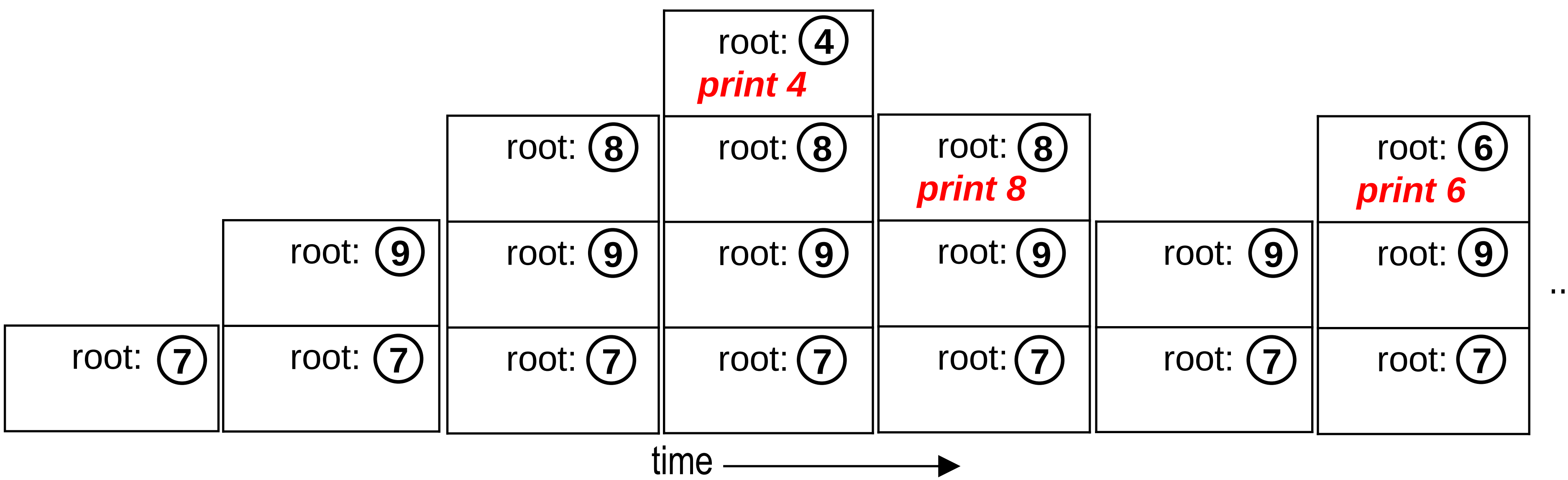
```
public class LinkedTree {
    …
    private Node root;

    public void postorderPrint() {
        if (root != null) {
            postorderPrintTree(root);
        }
    }

    private static void postorderPrintTree(Node root) {
        if (root.left != null) {
            postorderPrintTree(root.left);
        }
        if (root.right != null) {
            postorderPrintTree(root.right);
        }
        System.out.print(root.key + " "); // visit
    }
```

- Note that the root is printed *after* the two recursive calls.

Tracing Postorder Traversal

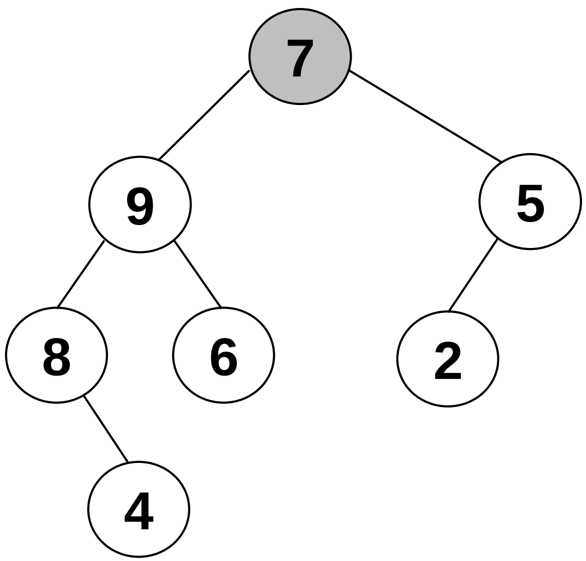
```
void postorderPrintTree(Node root) {
   if (root.left != null) {
       postorderPrintTree(root.left);
   }
   if (root.right != null) {
       postorderPrintTree(root.right);
   }
   System.out.print(root.key + " ");
}
```





Inorder Traversal

- inorder traversal of the tree whose root is N:
  1) recursively perform an inorder traversal of N's left subtree
  2) **visit the root, N**
  3) recursively perform an inorder traversal of N's right subtree



- Inorder traversal of the tree above:

  8 4 9 6 7 2 5

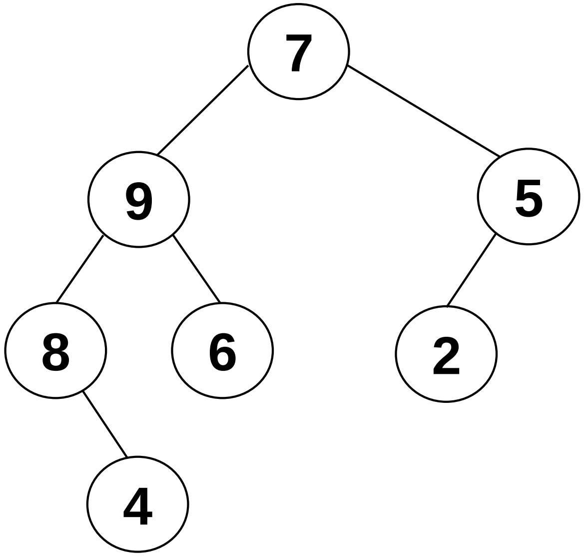
Implementing Inorder Traversal

```
public class LinkedTree {
    …
    private Node root;

    public void inorderPrint() {
        if (root != null) {
            inorderPrintTree(root);
        }
    }

    private static void inorderPrintTree(Node root) {
        if (root.left != null) {
            inorderPrintTree(root.left);
        }
        System.out.print(root.key + " ");
        if (root.right != null) {
            inorderPrintTree(root.right);
        }
    }
}
```

- Note that the root is printed *between* the two recursive calls.

Level-Order Traversal

- Visit the nodes one level at a time, from top to bottom and left to right.

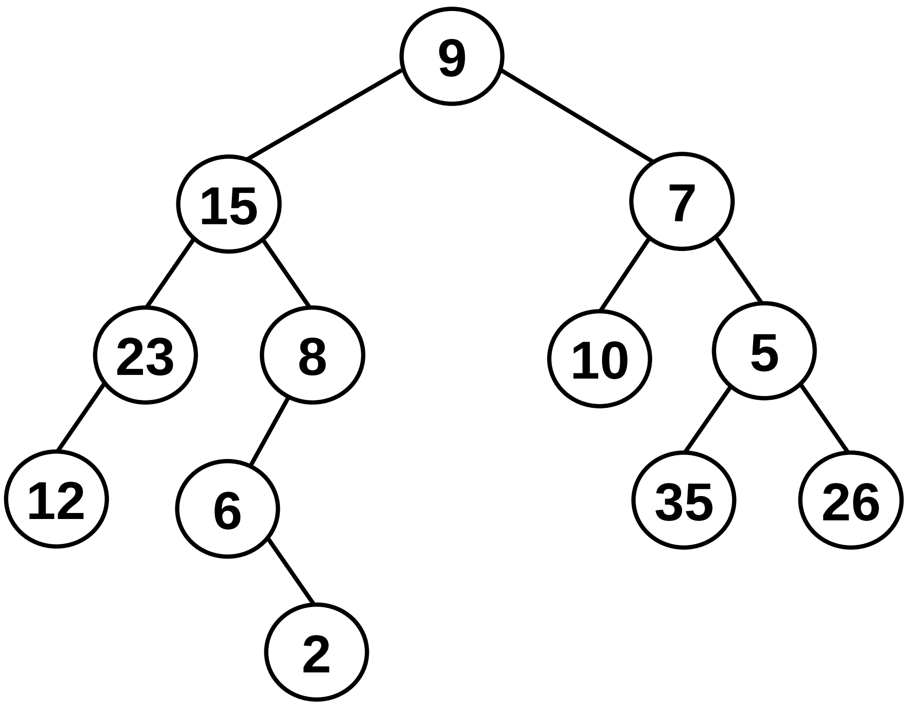


- Level-order traversal of the tree above: **7 9 5 8 6 2 4**
- We can implement this type of traversal using a queue.
  - more on this later!

Tree-Traversal Summary

preorder:	root, left subtree, right subtree
postorder:	left subtree, right subtree, root
inorder:	left subtree, root, right subtree
level-order:	top to bottom, left to right

- Perform each type of traversal on the tree below:



pre: 9 15 23 12 8 6 2 7 10 5 35 26

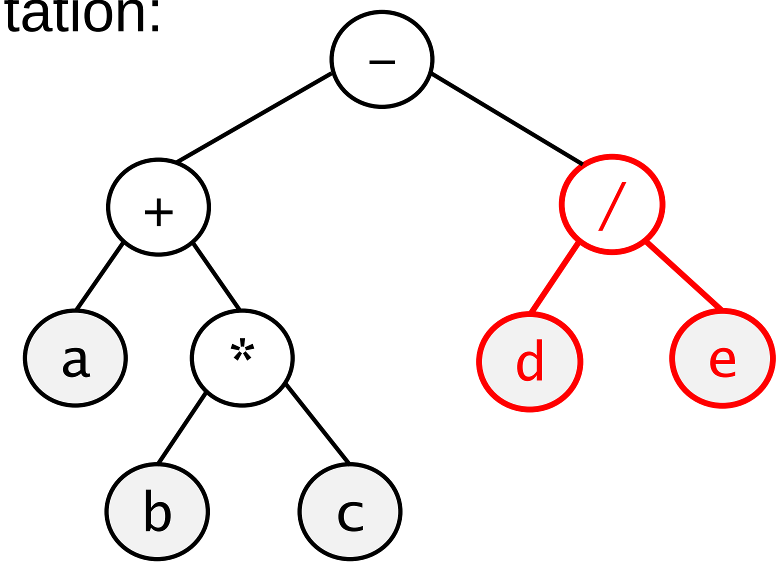
post: 12 23 2 6 8 15 10 35 26 5 7 9

in: 12 23 15 6 2 8 9 10 7 35 5 26

level: 9 15 7 23 8 10 5 12 6 35 26 2

Using a Binary Tree for an Algebraic Expression

- We'll restrict ourselves to fully parenthesized expressions and to the following binary operators: `+`, –, `*`, `/`
- Example expression: `((a + (b * c)) – (d / e))`
- Tree representation:

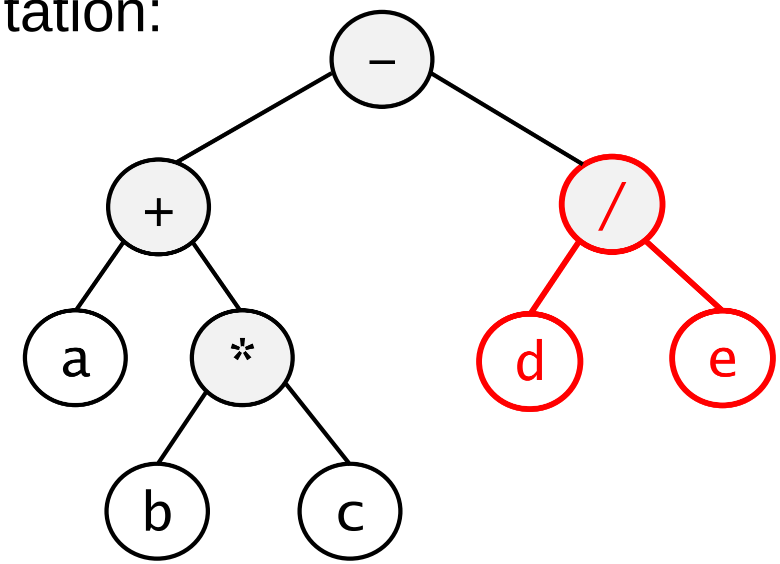


- Leaf nodes are variables or constants

- Interior nodes are operators.

Using a Binary Tree for an Algebraic Expression

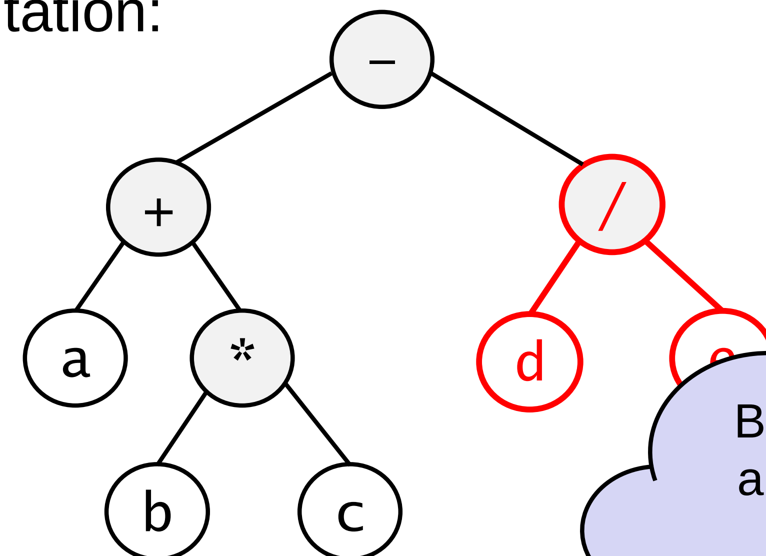
- We'll restrict ourselves to fully parenthesized expressions and to the following binary operators: +, –, *, /
- Example expression: `((a + (b * c)) – (d / e))`
- Tree representation:



- Leaf nodes are variables or constants
- Interior nodes are operators.

Using a Binary Tree for an Algebraic Expression

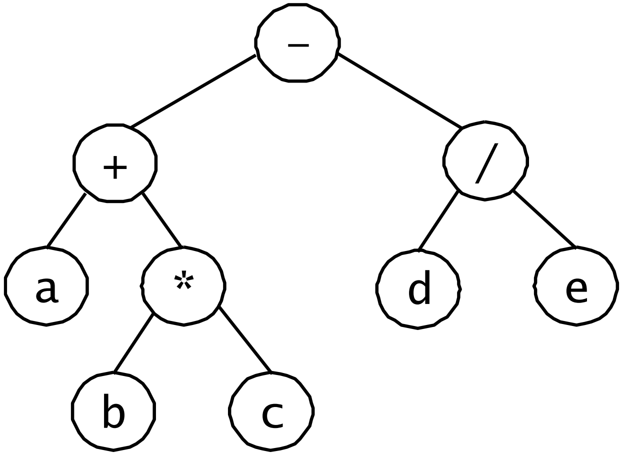
- We’ll restrict ourselves to fully parenthesized expressions and to the following binary operators: `+`, –, `*`, `/`
- Example expression: `((a + (b * c)) – (d / e))`
- Tree representation:



Because operators are binary, either a node has two children or it is a leaf node!

- Leaf nodes are variables or constants.
- Interior nodes are operators.

Traversing an Algebraic-Expression Tree



- Inorder gives conventional algebraic notation.
  - print ‘(’ before the recursive call on the left subtree
  - print ‘)’ after the recursive call on the right subtree
  - for tree at right: `((a + (b * c)) – (d / e))`
- Preorder gives functional notation.
  - print ‘(’s and ‘)’s as for inorder, and commas after the recursive call on the left subtree
  - for tree above: `subtr(add(a, mult(b, c)), divide(d, e))`
- Postorder gives the order in which the computation must be carried out on a stack/RPN calculator.
  - for tree above: `push a, push b, push c, multiply, add,…`