Recursive Functions on Binary Trees

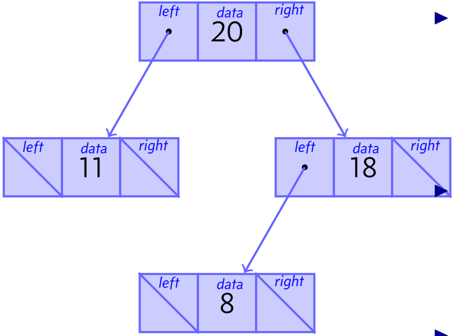
Dr. Mattox Beckman

Illinois Institute of Technology
Department of Computer Science

Objectives

- Define the parts of a binary tree
- Write some simple recursive functions to manipulate trees
- Analyze three traversal patterns

Binary Trees!



- A *binary tree* is a collection of nodes.
- Nodes have three elements:
  1. Some *data* (this may be complicated!)
  2. A pointer to a *left child*
  3. A pointer to a *right child*
- There are two kinds of nodes!
  1. A *leaf* node has no children.
  2. A *branch* node has one or two children.
- How will we implement this?

Define the Code

```
template <class T>
class BinaryTree {
    private:
    class Node {
        T data;
        Node *left, *right;
    };
    Node *root;
    // Other stuff too....
};
```

- Here is one way.
- This is a recursive data structure.
- Do you want a quick review of induction and recursion?

Induction

A proof by induction works by making two steps do the work of an infinite number of steps. It's really a way of being very lazy!

- Pick a property $P(n)$ which you'd like to prove for all n.
- **Base case:** Prove $P(n)$, for $n = 1$, or whatever n's smallest value should be.
- **Induction Case:** You want to prove $P(n)$, for some general n. To do that, *assume* that $P(n-1)$ is true, and use that information to prove that $P(n)$ has to be true.

The idea is that there are an infinite number of n such that $P(n)$ is true. But with this technique you only had to prove two cases.

Induction Example

To Prove: Let $P(n)$ = "The sum of the first n odd numbers is n^2."

Base Case: Let $n = 1$. Then n^2 = 1, and the sum of the list $\{1\}$ is 1; therefore the base case holds.

Induction Case: Suppose you need to show that this property is true for some n. First, pretend that somebody else already did all the work of proving that $P(n-1)$ is true. Now use that to show that $P(n)$ is true, and take all the credit.

If $\{1, 3, 5, \ldots, 2n-3\} = (n-1)^2$, then add $2n - 1 \ldots$

$$\{1, 3, 5, \ldots, 2n-3, 2n-1\} = (n-1)^2 + 2n - 1$$
$$\Rightarrow n^2 - 2n + 1 + 2n - 1 \Rightarrow n^2$$

Recursion

A recursive routine has a similar structure. You have a base case, a recursive case, and a conditional to check which case is appropriate.

- Pick a function $f(n)$ which you'd like to compute for all n.
- **Base case:** Compute $f(n)$, for $n = 1$, or whatever n's smallest value should be.
- **Recursive Case:** Assume that someone else already computed $f(n-1)$ for you. Use that information to compute $f(n)$, and then take all the credit.

Iterating Recursion Example

Suppose you want a recursive routine that computes the *n*th square.

```
1 int nthsq(int n) {
2     if (n == 0)
3         return 0;
4     else
5         return  2 * n - 1 + nthsq(n-1);
6 }
```

- The conditional checks which case is active.
- Line 2–3 is the base case — it stops the recursion.
- Line 4–5 is the recursive case.

Important things about recursion

```
int nthsq(int n) {
    if (n == 0)
        return 0;
    else
        return  2 * n - 1 + nthsq(n-1);
}
```

- Your base case has to stop the computation.
- Your recursive case has to call the function with a *smaller* argument than the original call.
- Your conditional expression has to be able to tell when the base case is reached.
- Failure to do any of the above will cause an infinite loop.

Recursive Functions on Trees

- Because trees are recursive, our functions tend to be recursive too.
- What is a *base case* for a tree?
- What is the *recursive case* for a tree?

```
int something(Node *t) {
    if (t==NULL)
        // DO BASE CASE HERE
    else {
        f(t->data,            // f = "combine parts"
          something(t->left),
          something(t->right));
     }
 }
```

Example 1a — Height

Think how you would define the height of a tree.
Try to do it recursively!

Example 1b — Height

- The height of a tree is $1 + max(height(left), height(right))$.
- The height of a null node is zero.
- Now try to code it! What is the base case?

```
int height(Node *t) {
    if (t==NULL)
        // What goes here?
    else
        // What goes here?
}
```

Example 1c — Height

- The height of a tree is $1 + max(height(left), height(right))$.
- The height of a null node is zero.
- Now try to code it! What is the recursive case?

```
int height(Node *t) {
    if (t==NULL)
        return 0;
    else
        // What goes here?
}
```

Example 1d — Height

- The height of a tree is $1 + max(height(left), height(right))$.
- The height of a null node is zero.
- Now try to code it!

```
int height(Node *t) {
    if (t==NULL)
        return 0;
    else
        return 1 + max(height(t->left),height(t->right));
}
```

Example 1a — Sum

Think how you would define the sum of a tree.
Try to do it recursively!

Example 1b — Sum

- The sum of a tree is $1 + \mathit{sum}(\mathit{left}) + \mathit{sum}(\mathit{right})$.
- The sum of a null node is zero.
- Now try to code it! What is the base case?

```
int sum(Node *t) {
    if (t==NULL)
        // What goes here?
    else
        // What goes here?
}
```

Example 1c — Sum

- The sum of a tree is $1 + sum(left) + sum(right)$.
- The sum of a null node is zero.
- Now try to code it! What is the recursive case?

```
int sum(Node *t) {
    if (t==NULL)
        return 0;
    else
        // What goes here?
}
```

Example 1d — Sum

- The sum of a tree is $1 + sum(left) + sum(right)$.
- The sum of a null node is zero.
- Now try to code it!

```
int sum(Node *t) {
    if (t==NULL)
        return 0;
    else
        return t->data + sum(t->left) + sum(t->right);
}
```

Your Turn!

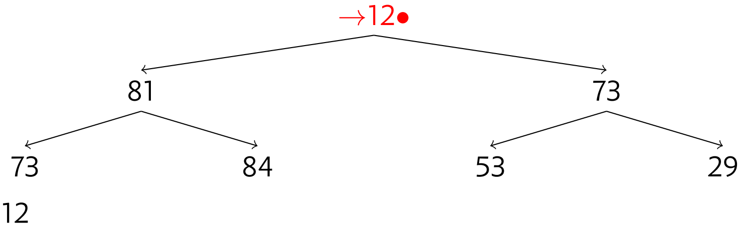
- Try the weirdo function in the activity!
  - Spend 2 minutes trying it yourself.
  - Then compare with someone next to you.
- GO!!

An interesting recursion

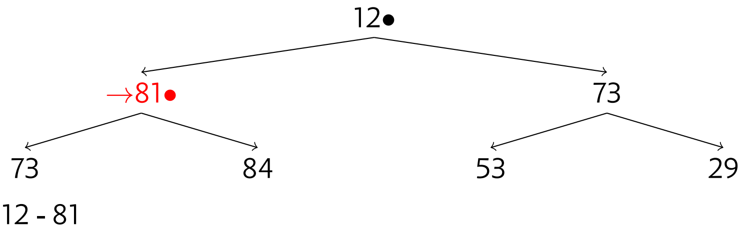
```
void preorder(Node *t) {
    print(t->data);
    preorder(t->left);
    preorder(t->right);
}
```

- What will this do?
- How many times will this function “visit” the node?

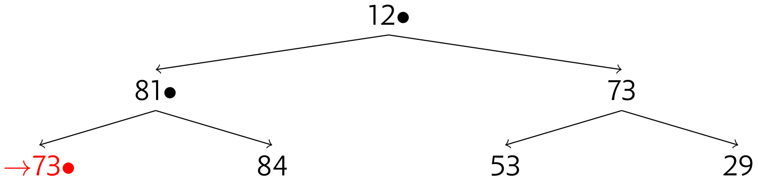
A Traversal



A Traversal

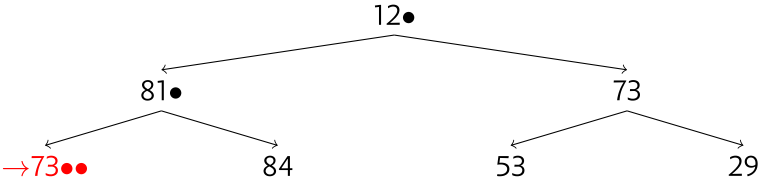


A Traversal



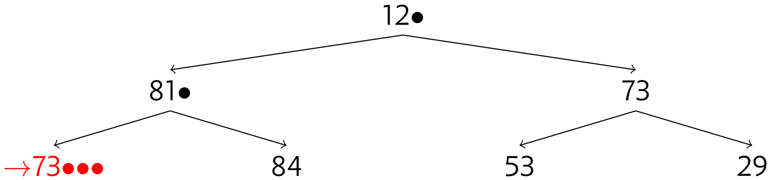
12 - 81 - 73

A Traversal



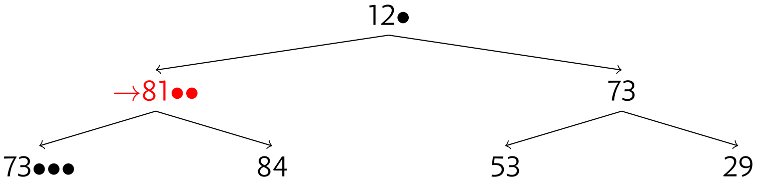
12 - 81 - 73

A Traversal



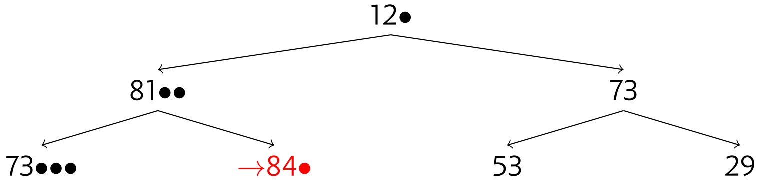
12 - 81 - 73

A Traversal



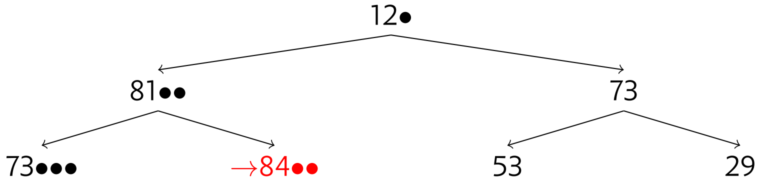
12 - 81 - 73

A Traversal



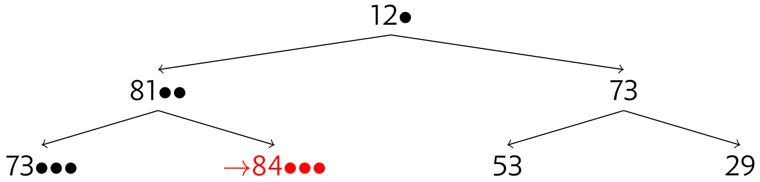
12 - 81 - 73 - 84

A Traversal



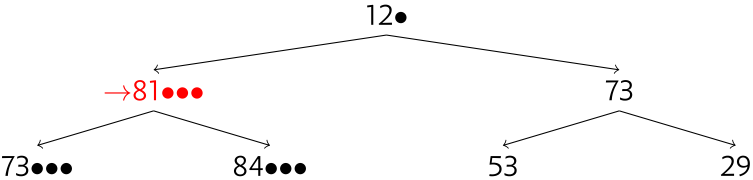
12 - 81 - 73 - 84

A Traversal



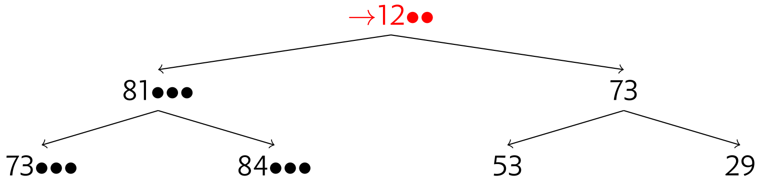
12 - 81 - 73 - 84

A Traversal



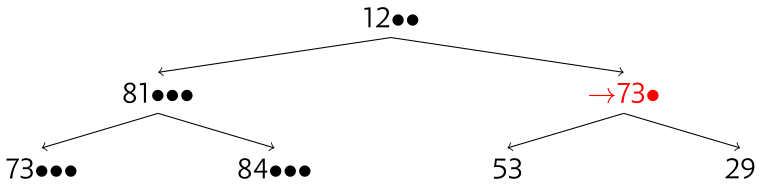
12 - 81 - 73 - 84

A Traversal



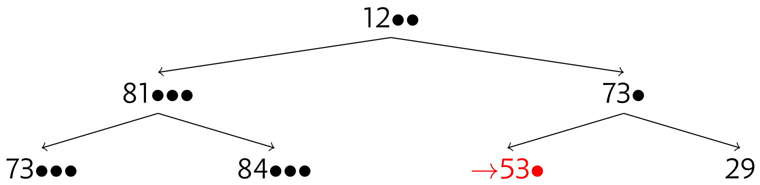
12 - 81 - 73 - 84

A Traversal



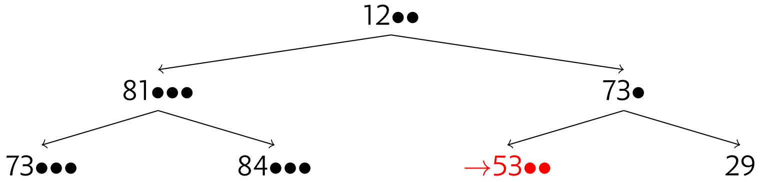
12 - 81 - 73 - 84 - 73

A Traversal



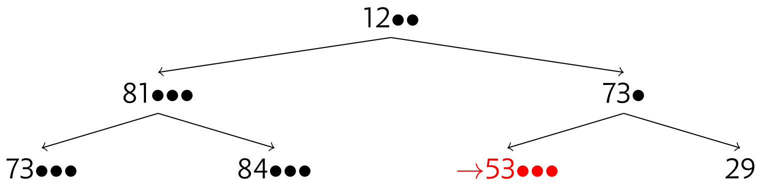
12 - 81 - 73 - 84 - 73 - 53

A Traversal



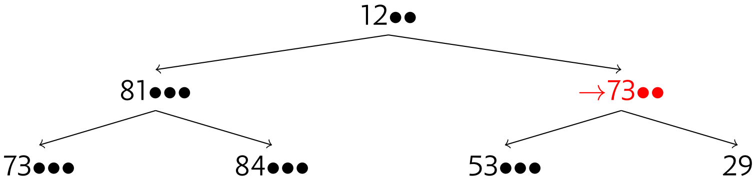
12 - 81 - 73 - 84 - 73 - 53

A Traversal



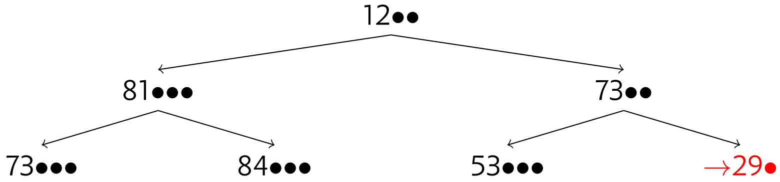
12 - 81 - 73 - 84 - 73 - 53

A Traversal



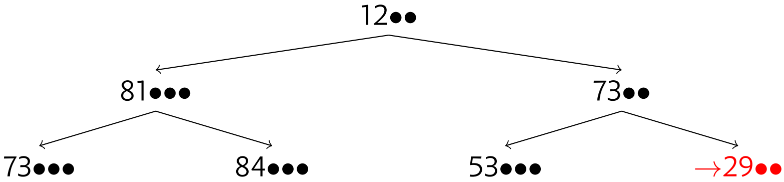
12 - 81 - 73 - 84 - 73 - 53

A Traversal



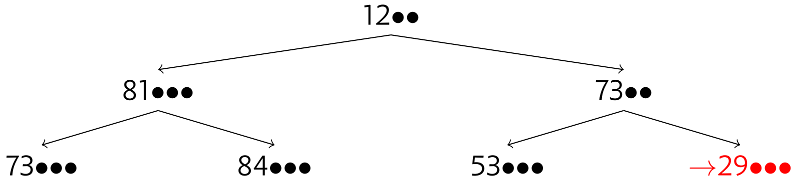
12 - 81 - 73 - 84 - 73 - 53 - 29

A Traversal



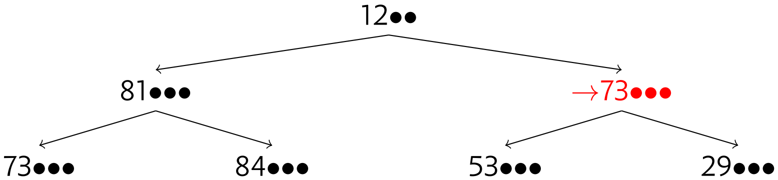
12 - 81 - 73 - 84 - 73 - 53 - 29

A Traversal



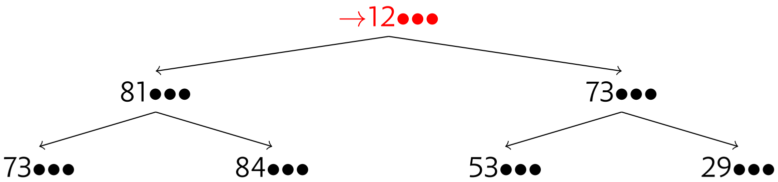
12 - 81 - 73 - 84 - 73 - 53 - 29

A Traversal



12 - 81 - 73 - 84 - 73 - 53 - 29

A Traversal



12 - 81 - 73 - 84 - 73 - 53 - 29

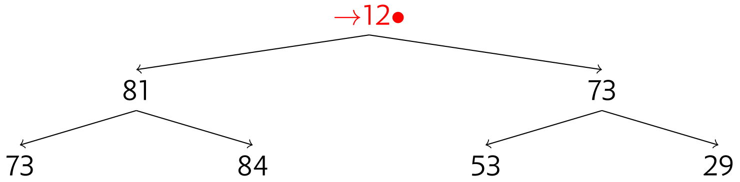
This is a *preorder* traversal. You will see it used in languages like LISP and CLOJURE.

An interesting recursion, part 2

```
void postorder(Node *t) {
    postorder(t->left);
    postorder(t->right);
    print(t->data);
}
```

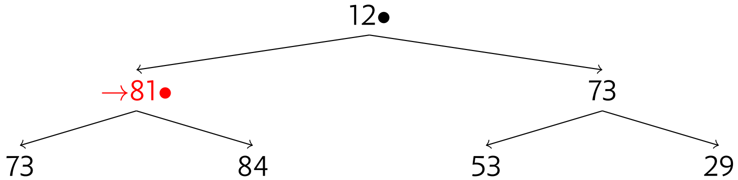
- What is different about this code?
- What will be the affect?

Postorder Traversal



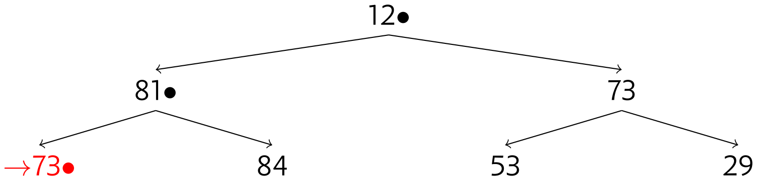
Result:

Postorder Traversal



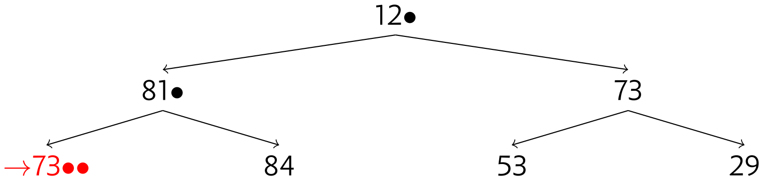
Result:

Postorder Traversal



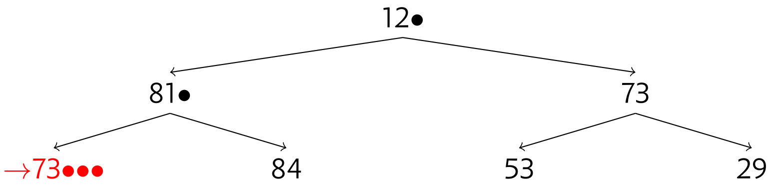
Result:

Postorder Traversal



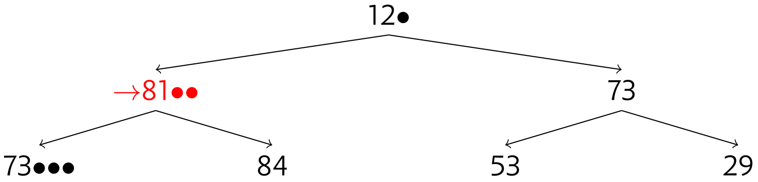
Result:

Postorder Traversal



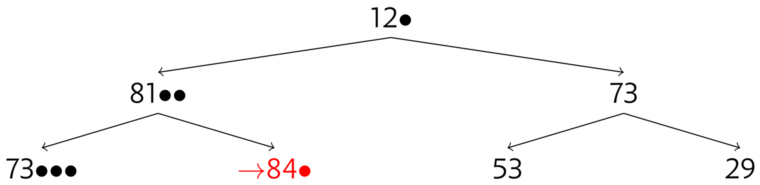
Result: 73

Postorder Traversal



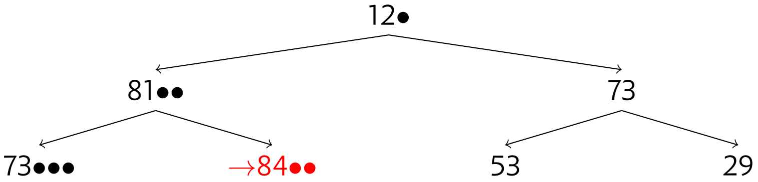
Result: 73

Postorder Traversal



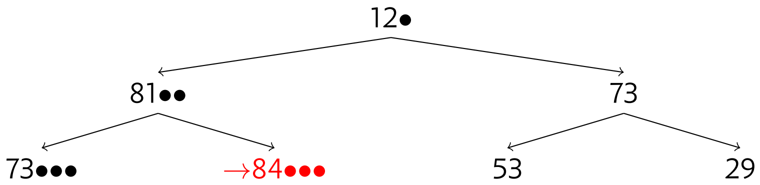
Result: 73

Postorder Traversal



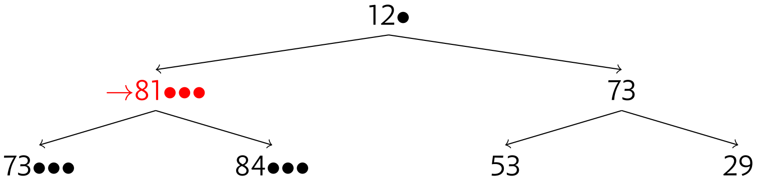
Result: 73

Postorder Traversal



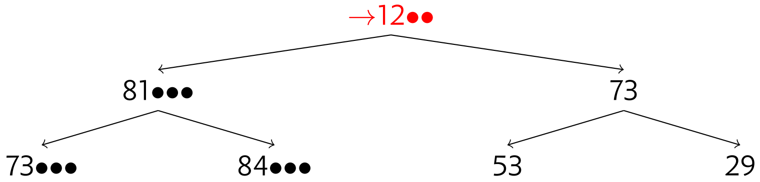
Result: 73 - 84

Postorder Traversal



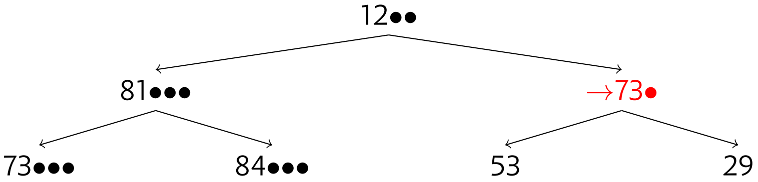
Result: 73 - 84 - 81

Postorder Traversal



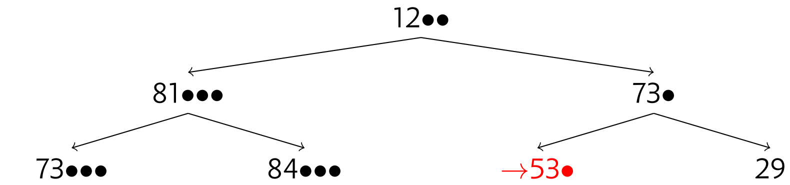
Result: 73 - 84 - 81

Postorder Traversal



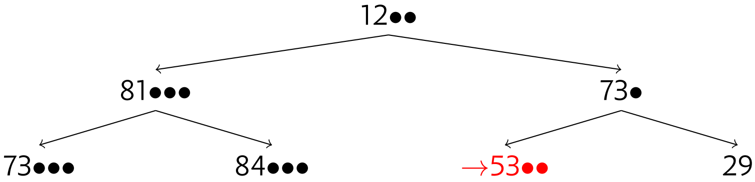
Result: 73 - 84 - 81

Postorder Traversal



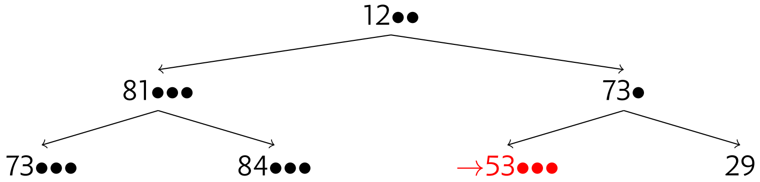
Result: 73 - 84 - 81

Postorder Traversal



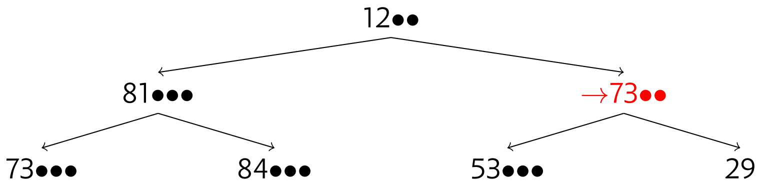
Result: 73 - 84 - 81

Postorder Traversal



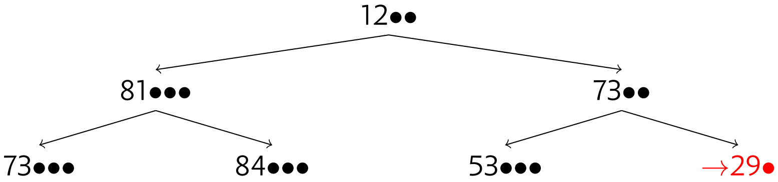
Result: 73 - 84 - 81 - 53

Postorder Traversal



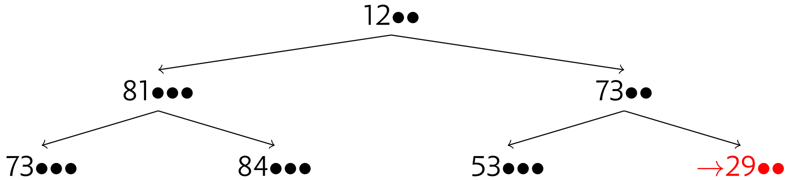
Result: 73 - 84 - 81 - 53

Postorder Traversal



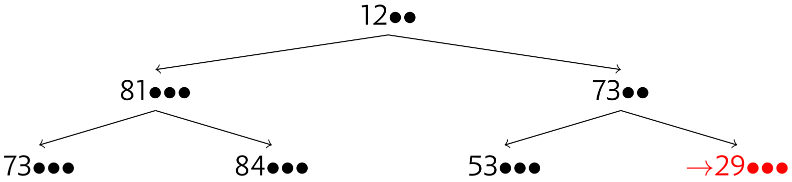
Result: 73 - 84 - 81 - 53

Postorder Traversal



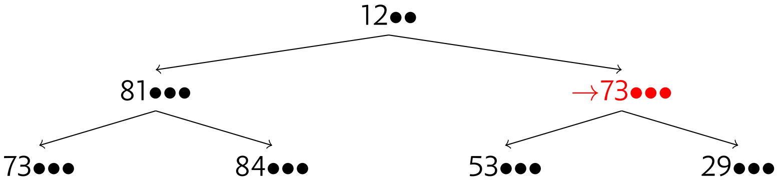
Result: 73 - 84 - 81 - 53

Postorder Traversal



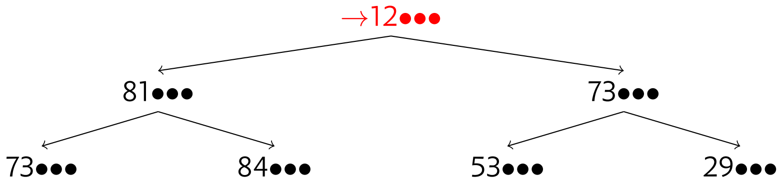
Result: 73 - 84 - 81 - 53 - 29

Postorder Traversal



Result: 73 - 84 - 81 - 53 - 29 - 73

Postorder Traversal



Result: 73 - 84 - 81 - 53 - 29 - 73 - 12

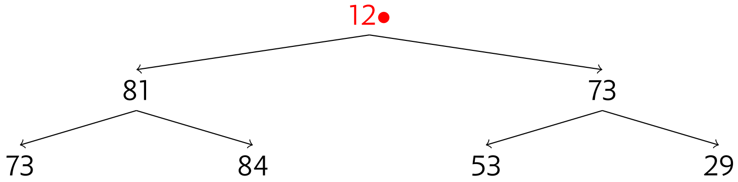
This is a *postorder* traversal. You will see it used in languages like POSTSCRIPT and in RPN calculators.

An interesting recursion, part 3

```
void inorder(Node *t) {
    inorder(t->left);
    print(t->data);
    inorder(t->right);
}
```

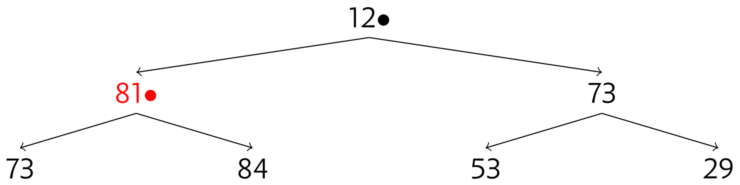
- What is different about this code?
- What will be the affect?

Inorder Traversal



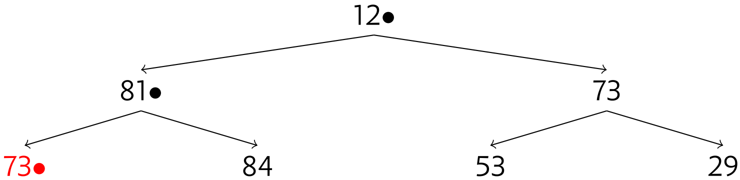
Result:

Inorder Traversal



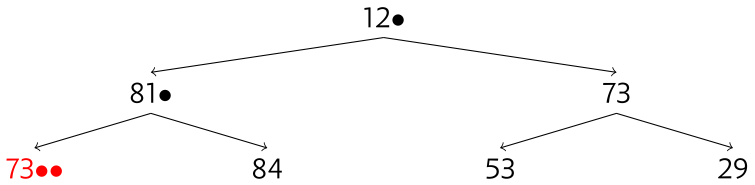
Result:

Inorder Traversal



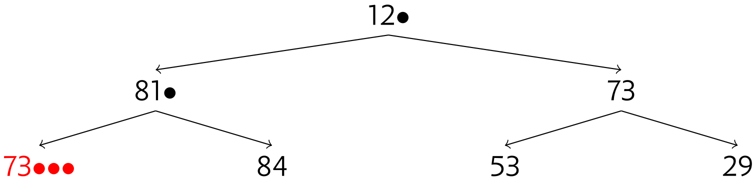
Result:

Inorder Traversal



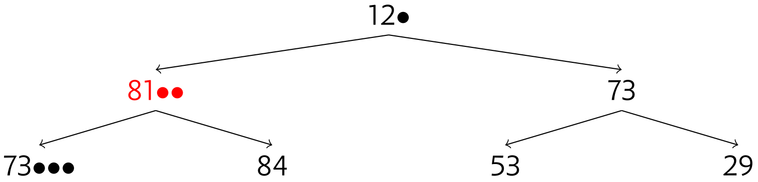
Result: 73

Inorder Traversal



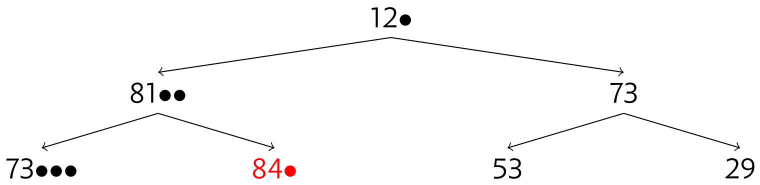
Result: 73

Inorder Traversal



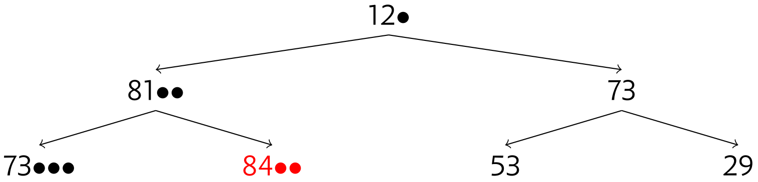
Result: 73 - 81

Inorder Traversal



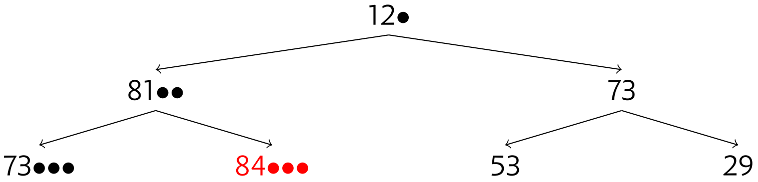
Result: 73 - 81

Inorder Traversal



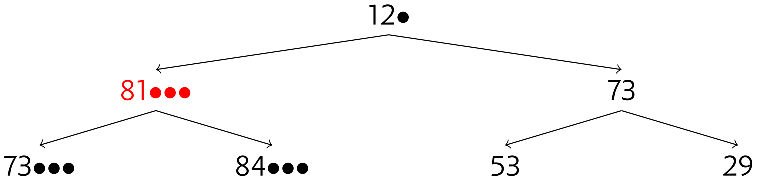
Result: 73 - 81 - 84

Inorder Traversal



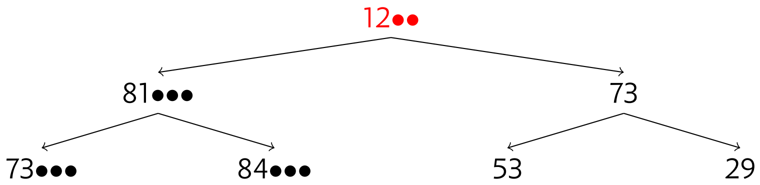
Result: 73 - 81 - 84

Inorder Traversal



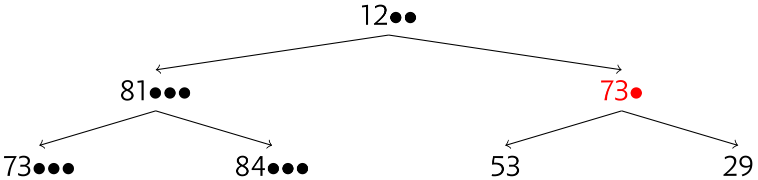
Result: 73 - 81 - 84

Inorder Traversal



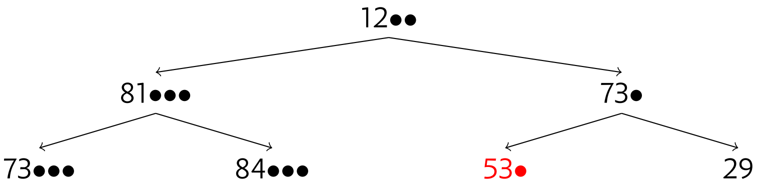
Result: 73 - 81 - 84 - 12

Inorder Traversal



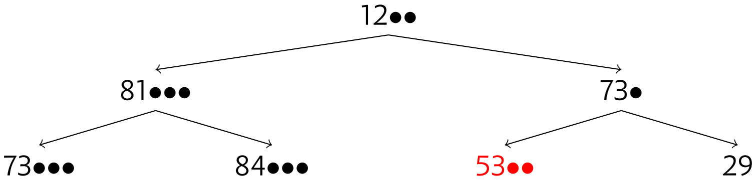
Result: 73 - 81 - 84 - 12

Inorder Traversal



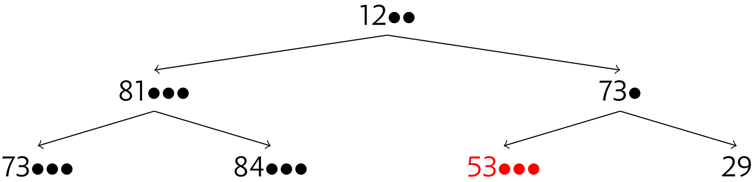
Result: 73 - 81 - 84 - 12

Inorder Traversal



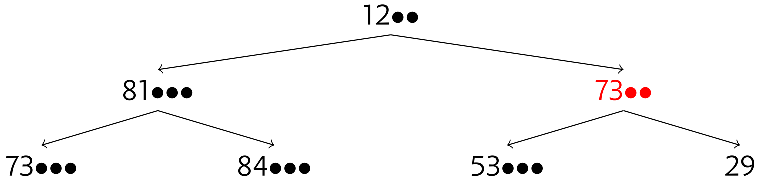
Result: 73 - 81 - 84 - 12 - 53

Inorder Traversal



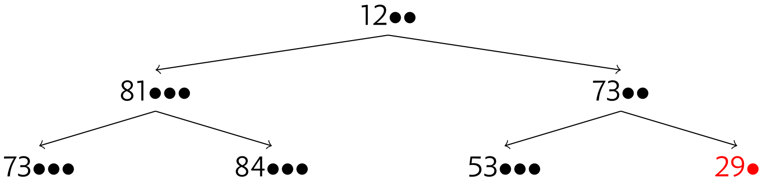
Result: 73 - 81 - 84 - 12 - 53

Inorder Traversal



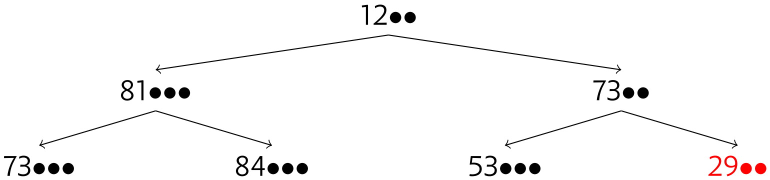
Result: 73 - 81 - 84 - 12 - 53 - 73

Inorder Traversal



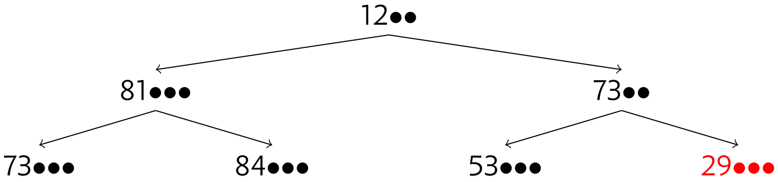
Result: 73 - 81 - 84 - 12 - 53 - 73

Inorder Traversal



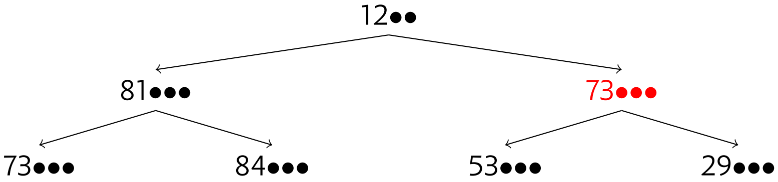
Result: 73 - 81 - 84 - 12 - 53 - 73 - 29

Inorder Traversal



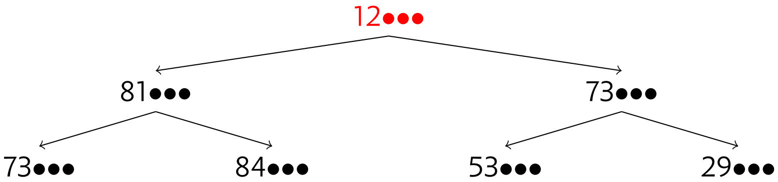
Result: 73 - 81 - 84 - 12 - 53 - 73 - 29

Inorder Traversal



Result: 73 - 81 - 84 - 12 - 53 - 73 - 29

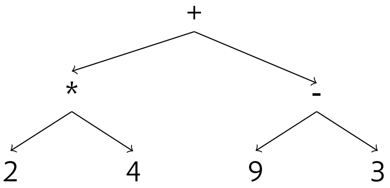
Inorder Traversal



Result: 73 - 81 - 84 - 12 - 53 - 73 - 29

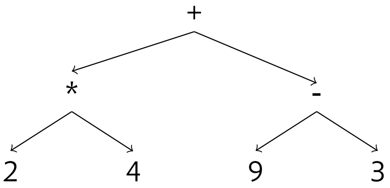
This is an *inorder* traversal. You will see it used in algebraic notation. You will lose the structure of the tree with this!

Interesting Fact



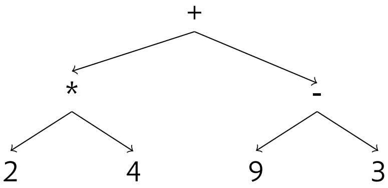
- If you know how many children each node should have, you can take the preorder or postorder traversal and reconstruct the tree.
- You cannot do that with inorder.
- Find the traversals!
  - Preorder:
  - Postorder:
  - Inorder:

Interesting Fact



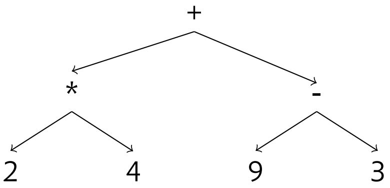
- If you know how many children each node should have, you can take the preorder or postorder traversal and reconstruct the tree.
- You cannot do that with inorder.
- Find the traversals!
  - Preorder: + * 2 4 - 9 3
  - Postorder:
  - Inorder:

Interesting Fact



- If you know how many children each node should have, you can take the preorder or postorder traversal and reconstruct the tree.
- You cannot do that with inorder.
- Find the traversals!
  - Preorder: + * 2 4 - 9 3
  - Postorder: 2 4 * 9 3 - +
  - Inorder:

Interesting Fact



- If you know how many children each node should have, you can take the preorder or postorder traversal and reconstruct the tree.
- You cannot do that with inorder.
- Find the traversals!
  - Preorder: + * 2 4 - 9 3
  - Postorder: 2 4 * 9 3 - +
  - Inorder: 2 * 4 + 9 - 3