Joanna Klukowska
joannakl@cs.nyu.edu

Lecture 5: Introduction to Trees and Binary Trees

Reading materials

Goodrich, Tamassia, Goldwasser: Chapter 8

OpenDSA: Chapter 18 and 12

Binary search trees visualizations:

http://visualgo.net/bst.html

http://www.cs.usfca.edu/~galles/visualization/BST.html

Topics Covered

Joanna Klukowska
joannakl@cs.nyu.edu

1 Tree Terminology

A **tree** is a structure in which each node can have multiple successors (unlike the linear structures that we have been studying so far, in which each node always had at most one successor).

The first node in a tree s called a **root**, it is often called the top level node (YES, in computer science root of a tree is at the top of a tree). In a tree, there is always a unique path from the root of a tree to every other node in a tree - this has an important consequence: there are no cycles in a tree (think of a cycle as a closed path that allows us to go in a cycle infinitely many times).

Any node within a tree can be viewed as a root of its own **subtree** - just take any node, cut of the branch/edge that connects above to the rest of a tree, and it becomes a root with a smaller (possibly empty) tree of its own.

Given a node in a tree, its successors (nodes connected to it in a level below) are called its **children**. **Descendants** of a node are its children, and the children of its children, and the children of the children of its children, and ... - all the nodes below that are connected to that node. (In some definitions, the node itself is its own descendant.)

Given a node in a tree, its predecessor (node that connects to it in a level above - there is only one such node) is called its **parent**. **Ascendants** of a node are its parent, and the parent of the parent, and ... - all the nodes along the path from itself to the root. (In some definitions, the node itself is its own ascendant.)

The nodes that share the same parent node, are called **siblings**.

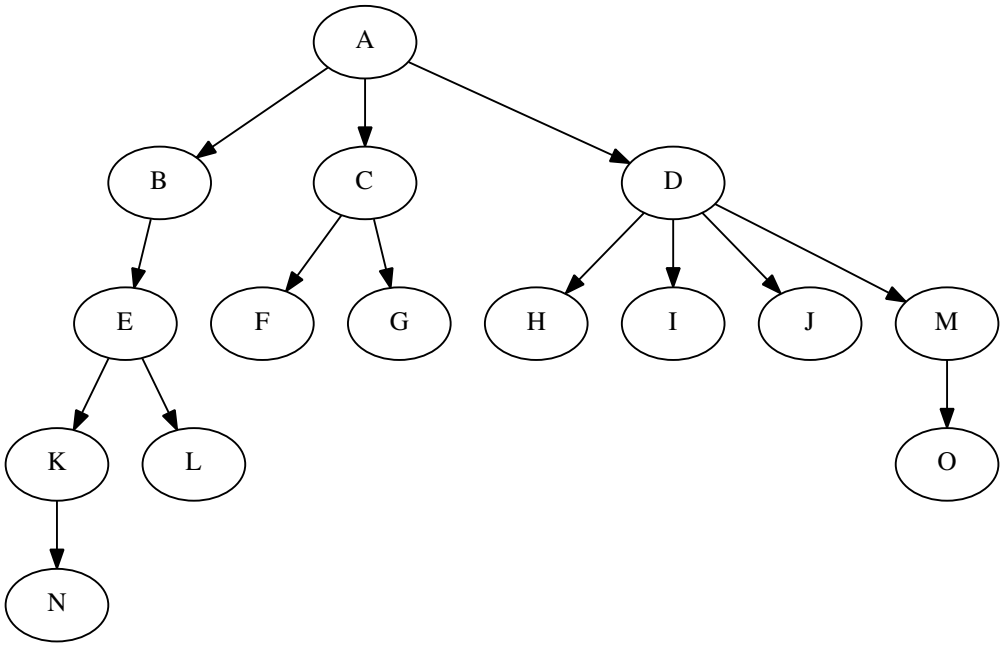
The nodes at the end of each path that leads from root downwards are called **leaves**. The other way to think about it is that leaves are the nodes that do not point to any other nodes (or point only to null). In a linear structure there was only one such node indicating the end of the list. In trees we have many such nodes. The leaf nodes do not have any children nodes. The leaf nodes are also called **external** nodes in contrast to all the other non-leaf nodes that are called **internal** nodes.

An **edge** in a tree is formed by a pair of nodes such that one is a parent of the other. A **path** in a tree is a sequence of nodes such that any two consecutive nodes form an edge.

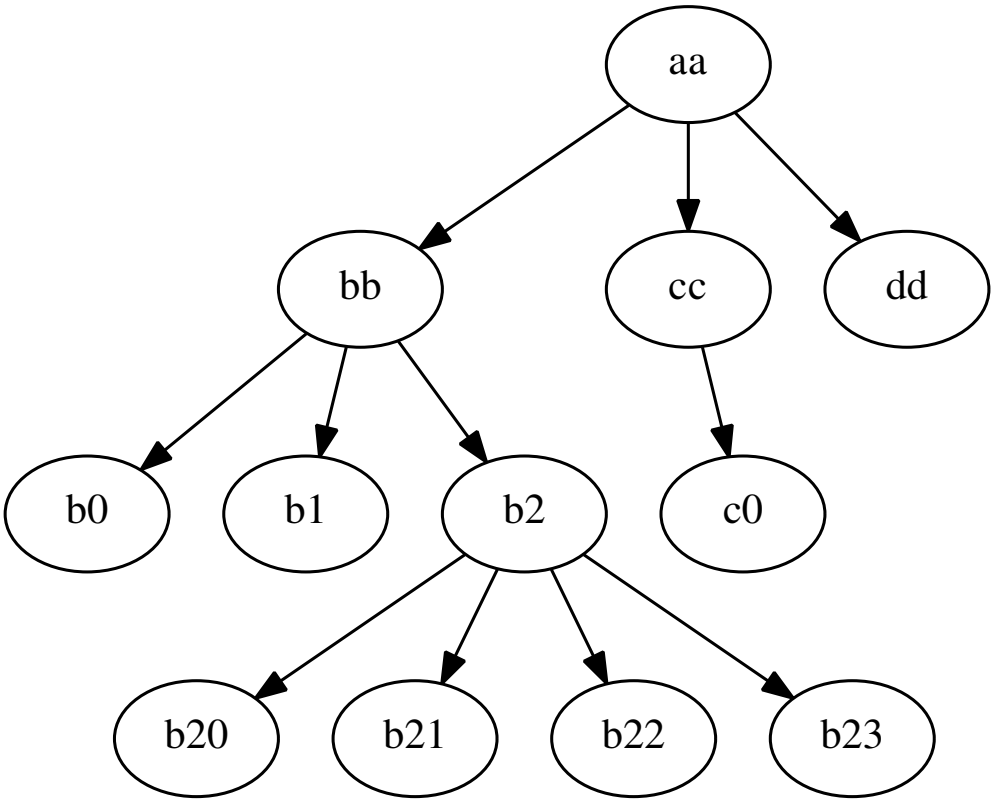
The **level** of a node refers to the distance of a node from the root. Root is a distance zero from itself, so the level of the root is 0 (or root is at level zero in the tree). Children of the root are at level 1, "grandchildren" or children of the children of the root are at level 2, etc. The **depth of a tree** is equal to the level of the deepest node. The **depth of a node** is the number of ascendants of that node in a tree (or the number of levels above the node).

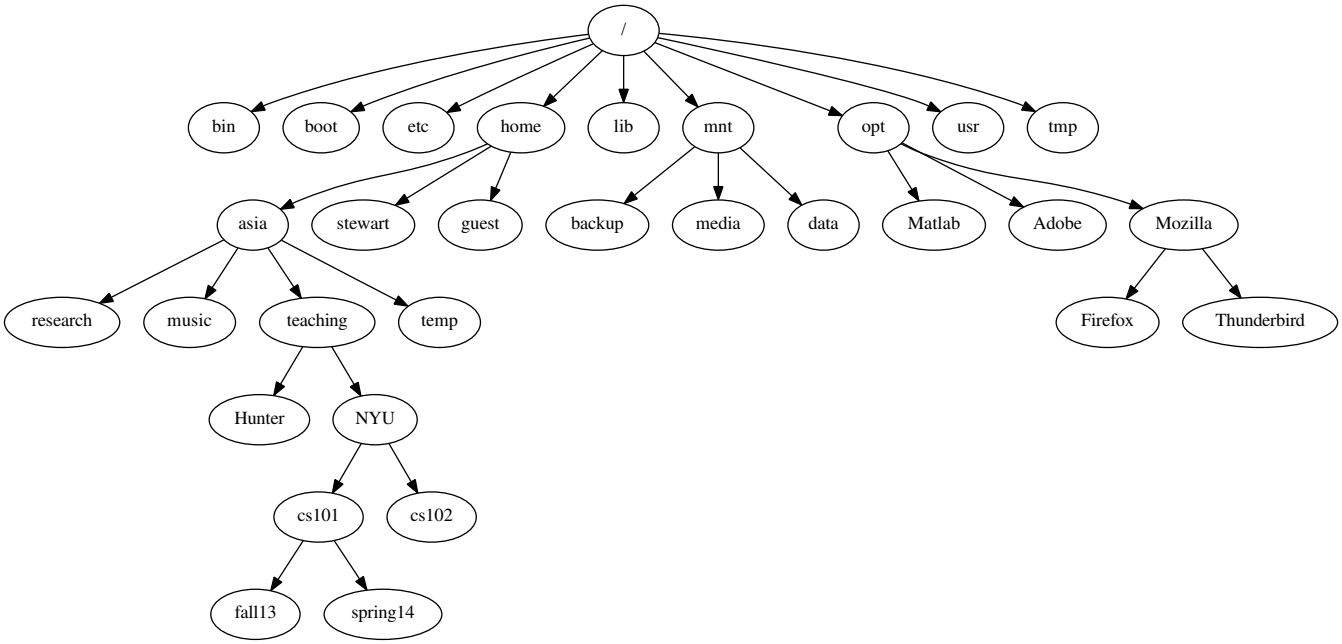
The **height** of a tree is calculated from the leaves up to the root. The leaf-nodes have a height of zero. For the internal nodes, the height is calculated as one more than the maximum of the heights of its children. The height of a tree is the height of the root. (In some definitions, the height of the leaf nodes is defined to be one, not zero.)

In general trees the order of the children does not really matter. A **tree is ordered** when the ordering of children is relevant.



Joanna Klukowska
joannakl@cs.nyu.edu

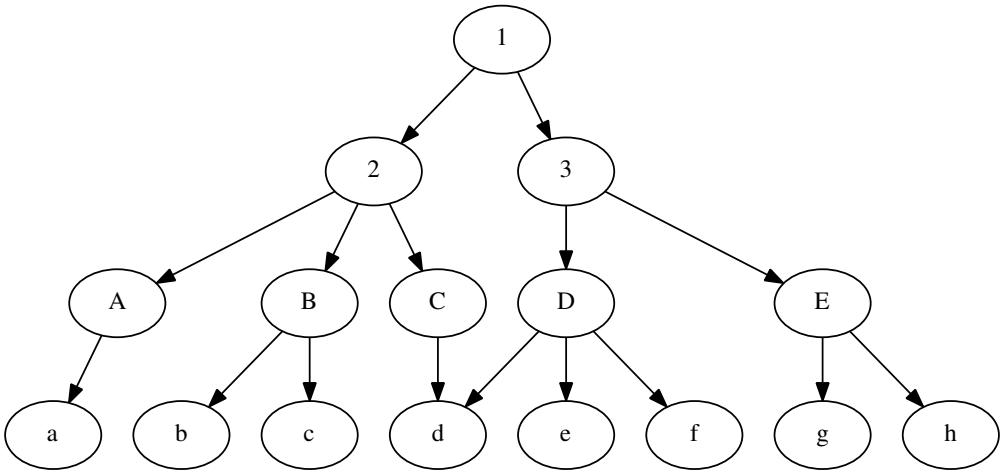




Linux directory structure

Joanna Klukowska
joannakl@cs.nyu.edu

THIS IS NOT A TREE !!! Why?



Not A Tree !!!

1.1 Binary Trees

A **binary tree** is a special kind of tree in which each node can have at most two children: they are distinguished as a **left child** and a **right child.** The order of the nodes matters (we cannot just swap left and right), so it is an ordered tree. The subtree rooted at the left child of a node is called its **left subtre**e and the subtree rooted at the right child of a node is called its **right subtree**.

In binary trees there are at most 2^L nodes at level L.

Given N nodes, the "shortest" binary tree that we can construct has $\lfloor \log_2 N \rfloor + 1$ levels. We will come back to this idea when we talk about efficiency of trees.

Given N nodes, the "tallest" binary tree that we can construct has $N-1$ levels. Why?

There are two different definitions for a **full** tree.

Definition 1: In a **full** binary tree each node has exactly zero or two children. (This is the definition used in our textbook.)

Definition 2: In a **full** binary tree each level has maximal possible number of nodes.

Those definition are not equivalent and you need to be careful when you read about trees in different resources!

In a **complete** binary tree with L levels, the levels $0 \leq l \leq L-1$ have the maximal number of nodes, i.e., 2^l, and the nodes at level L are placed as far left as possible.

Joanna Klukowska
joannakl@cs.nyu.edu

2 Node Structure

2.1 Binary Tree Node

The node of a binary tree needs to store a data item and references to its children. (Optionally, it may also store a reference to its parent.)

```
class BTNode <E> {

    private T data;
    private BTNode <E> left;
    private BTNode <E> right;

    public BTNode ( E data ) {
        this.data = data;
    }

    public BTNode (E data, BTNode <E> left, BTNode <E> right ) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
    ...
    //desired setters and getters if any
    ...
}
```

2.2 General Tree Node

In a general tree we do not know the maximum degree that a tree may have (the largest number of children per node), and because it is inefficient to create a structure with a very large fixed number of child entries, the best implementation of a general tree uses a linked list of siblings and a single child, as follows.

```
class TreeNode <E>{

    E data
    TreeNode<E> firstChild
    TreeNode<E> nextSibling

}
```

3 Computing the Size of a Tree

Trees are inherently recursive structures: if we pick any node in a tree and disconnect it from its parent, we are still left with a tree (a smaller one, but a tree). This makes recursive algorithms very intuitive.

3.1 Binary Tree Size

Recursive approach

How are we going to compute the number of nodes in the tree? Simple! If the tree has no nodes, then its size is zero. If the tree has at least one node it is the root node. We can ”ask” its children for the sizes of trees rooted at them, and then add the two numbers together and add one for the root. This gives us total number of nodes in the tree. Here is the pseudocode for recursive algorithm for the size method of a BST class.

Joanna Klukowska
joannakl@cs.nyu.edu

```
int recSize ( BTNode<E> root )
    if root == null
       return 0
    else
       return recSize(root.left) + recSize(root.right) + 1;
```

Such method would be a private method in the class and would be called from a public wrapper method. Why? Because we should not allow the client of the class to have access to the root of the tree (what if they set it to null, for example?), so we need to have a way of calling the recursive method ourselves.

```
int size ()
    return recSize( rootOfTheTree)
```

Iterative approach

The same algorithm implemented recursively is much more complicated. When we looked at iterative and recursive implementations of the size method for the linked list they were not significantly different in complexity and, at least for some of us, the iterative method was a bit more intuitive. This is not true when implementing the size method for trees, because at each node we have multiple branches, so we need to keep track of unexplored branches as we explore the others.

Here is the pseudocode for the iterative implementation of a size method.

```
int size ( )
    set counter to 0
    if tree is not empty (root is not null)
       create an empty stack
       push the root of this tree onto the stack
       while stack is not empty
          set current reference to top of the stack
            remove the top from the stack
          increment counter
          if current has a left child
            push left child onto the stack
          if current has a right child
            push right child onto the stack
    return counter
```

The stack allows us to put the unexplored nodes on hold while we explore other nodes. An empty stack indicates that all the nodes in the tree have been counted (or that we have a bug in our code, but lets be optimistic).

Can you think of a way of writing the above algorithm using a queue?

3.2 General Tree Size

Try to design an algorithm that counts the number of nodes in a general tree with the node structure discussed before.

You may want to start by first trying to write a function that counts the number of children given a reference to a single node and then extend it to a function that counts all of the nodes in a tree rooted at a given node.

4 Traversing a Binary Tree

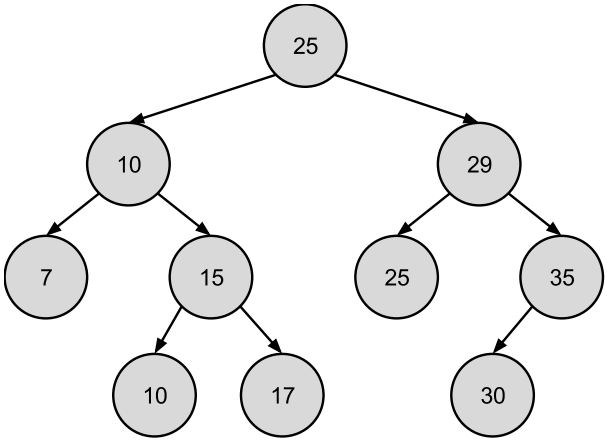
We often need to perform an operation on every node in a tree and sometimes the order matters. There are multiple approaches to tree traversals and we will discuss three of them: inorder, preorder and postorder.

Joanna Klukowska
joannakl@cs.nyu.edu

4.1 InOrder Traversal

The first one is **inorder traversal**. As the name suggests it visits the nodes according to their ordering (the same ordering that determines their placement into a BST).

Example: Given the following tree, the inorder traversal should visit (and process) the nodes in the increasing order of integers.



Inorder traveral: 7, 10, 10, 15, 17, 25, 25, 29, 30, 35.

4.1.1 Recursive Approach

In a perfectly balanced binary search tree one can think of the root as (approximately) middle element: the nodes with values smaller than the root are in its left subtree - so we need to process them before the root, and the nodes with values larger than the root are in its right subtree - so we need to process them after the root. The same is true about every other node in the tree, not only the root. And once again, this yields a very intuitive recursive algorithm for inorder tree traversal:

- process the nodes in the left subtree of a node
- process the node itself
- process the nodes in the right subtree of a node

In the above, the word ”process” is intentionally vague. The actual action depends on what exactly we need to do. If we want to print the content of the binary tree, then ”process” means ”print”. If we need to perform some other computation based on the data stored in the nodes, then ”process” means ”perform the computation”.

Here is the pseudocode for a recursive inorder traversal algorithm.

```
recInorderTraversal ( BTNode<E> node )

    if node is not null
        recInorderTraversal( node.left )
        process the node
        recInorderTraversal( node.right )
```

4.1.2 Iterative Approach

The iterative approach is significantly different and requires an extra data structures to keep track of all the nodes ”on hold”: as we descend down the tree to find the smallest node to be printed first, we need to keep track of all the nodes on the way so we know where to return. As with the iterative `size()` method implementation we going to make use of a stack.

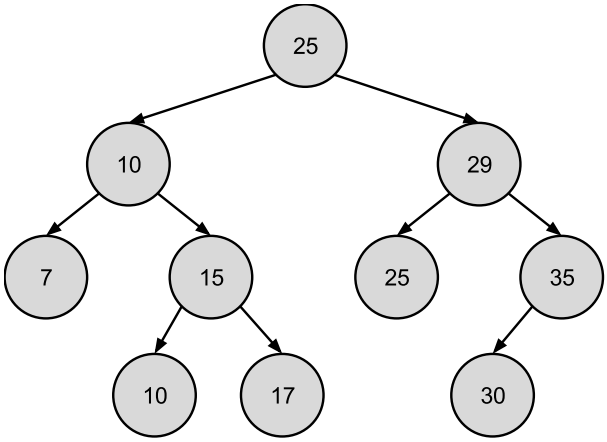
Here is the pseudocode for an iterative inorder traversal algorithm.

Joanna Klukowska
joannakl@cs.nyu.edu

```
inorderTraversal ( )

    if tree is not empty (root is not null)
        create an empty stack
        set current to root of this tree
        set done to false
        while not done
            if current is not null
              push current onto the stack
              current = current.left
            else if stack is not empty
              current = top of the stack
              remove the item from top of the stack
              process current
              current = current.right
            else
              set done to true
```

This is not as intuitive as the recursive version, but it might be easier to follow with an example. We are going to apply this algorithm to the following tree:



State of the inOrder traversal program after each change to current when run with the above tree.

`iter` indicates iteration number of the while loop

`current` indicates the value of the node pointed to by the current pointer

`stack` indicates the content of the stack used for keeping track of nodes that we need to return to

`processed` indicates which nodes have been already processed

```
iter:  0 current:   25 stack: []           processed: []
iter:  1 current:   10 stack: [25]         processed: []
iter:  2 current:    7 stack: [25, 10]     processed: []
iter:  3 current: null stack: [25, 10, 7]  processed: []
iter:  4 current:    7 stack: [25, 10]     processed: []
iter:  4 current: null stack: [25, 10]     processed: [7]
iter:  5 current:   10 stack: [25]         processed: [7]
iter:  5 current:   15 stack: [25]         processed: [7, 10]
iter:  6 current:   10 stack: [25, 15]     processed: [7, 10]
iter:  7 current: null stack: [25, 15, 10] processed: [7, 10]
iter:  8 current:   10 stack: [25, 15]     processed: [7, 10]
iter:  8 current: null stack: [25, 15]     processed: [7, 10, 10]
iter:  9 current:   15 stack: [25]         processed: [7, 10, 10]
```

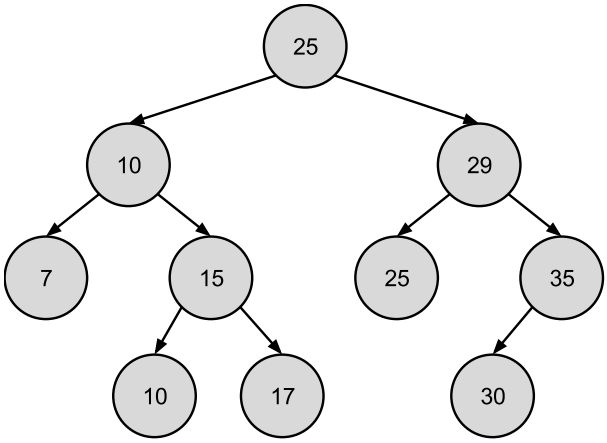
Joanna Klukowska
joannakl@cs.nyu.edu

```
iter:  9 current:   17 stack: [25]          processed: [7, 10, 10, 15]
iter: 10 current: null stack: [25, 17]      processed: [7, 10, 10, 15]
iter: 11 current:   17 stack: [25]          processed: [7, 10, 10, 15]
iter: 11 current: null stack: [25]          processed: [7, 10, 10, 15, 17]
iter: 12 current:   25 stack: []            processed: [7, 10, 10, 15, 17]
iter: 12 current:   29 stack: []            processed: [7, 10, 10, 15, 17, 25]
iter: 13 current:   25 stack: [29]          processed: [7, 10, 10, 15, 17, 25]
iter: 14 current: null stack: [29, 25]      processed: [7, 10, 10, 15, 17, 25]
iter: 15 current:   25 stack: [29]          processed: [7, 10, 10, 15, 17, 25]
iter: 15 current: null stack: [29]          processed: [7, 10, 10, 15, 17, 25, 25]
iter: 16 current:   29 stack: []            processed: [7, 10, 10, 15, 17, 25, 25]
iter: 16 current:   35 stack: []            processed: [7, 10, 10, 15, 17, 25, 25, 29]
iter: 17 current:   30 stack: [35]          processed: [7, 10, 10, 15, 17, 25, 25, 29]
iter: 18 current: null stack: [35, 30]      processed: [7, 10, 10, 15, 17, 25, 25, 29]
iter: 19 current:   30 stack: [35]          processed: [7, 10, 10, 15, 17, 25, 25, 29]
iter: 19 current: null stack: [35]          processed: [7, 10, 10, 15, 17, 25, 25, 29, 30]
iter: 20 current:   35 stack: []            processed: [7, 10, 10, 15, 17, 25, 25, 29, 30]
iter: 20 current: null stack: []            processed: [7, 10, 10, 15, 17, 25, 25, 29, 30, 35]
```

4.2 PreOrder Traversal

In the **preorder traversal** of the tree, we visit the node before exploring its children.

Example: Given the following tree, the preorder traversal visits the root, then visits its left subtree and then visits its right subtree.



Preorder traveral: 25, 10, 7, 15, 10, 17, 29, 25, 35, 30.

4.2.1 Recursive Approach

The preorder traversal algorithm is intuitively described by the sequence of three steps mentioned already above:

- process the node itself
- process the nodes in the left subtree of the node
- process the nodes in the right subtree of the node

The pseudocode for a recursive preorder traversal algorithm does not differ much:

```
recPreorderTraversal ( BTNode<E> node )

    if node is not null
       process the node
       recInorderTraversal( node.left )
       recInorderTraversal( node.right )
```

Joanna Klukowska
joannakl@cs.nyu.edu

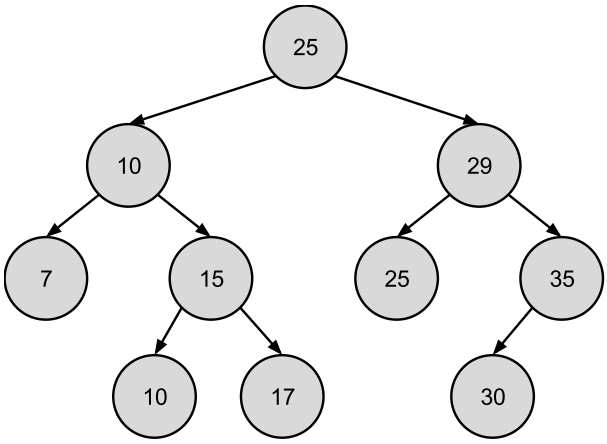
4.2.2 Iterative approach

Exercise: try to derive the pseudocode for the preorder traversal on your own to see if you understand the concepts.

4.3 PostOrder Traversal

Finally, in the **postorder traversal** of the tree, we visit the node after exploring its children.

Example: Given the following tree, the postorder traversal visits its left subtree, then visits its right subtree and then visits the root.



Postorder traversal: 7, 10, 17, 15, 10, 25, 30, 35, 29, 25.

4.3.1 Recursive Approach

The postorder traversal algorithm is intuitively described by the sequence of three steps mentioned already above:

- process the nodes in the left subtree of a node
- process the nodes in the right subtree of a node
- process the node itself

The pseudocode for a recursive preorder traversal algorithm does not differ much:

```
recInorderTraversal ( BTNode node )

    if node is not null
        recInorderTraversal( node.left )
        recInorderTraversal( node.right )
        process the node
```