CSC 148 Winter 2017

Week 7

Recursive structures

Trees

Bogdan Simion

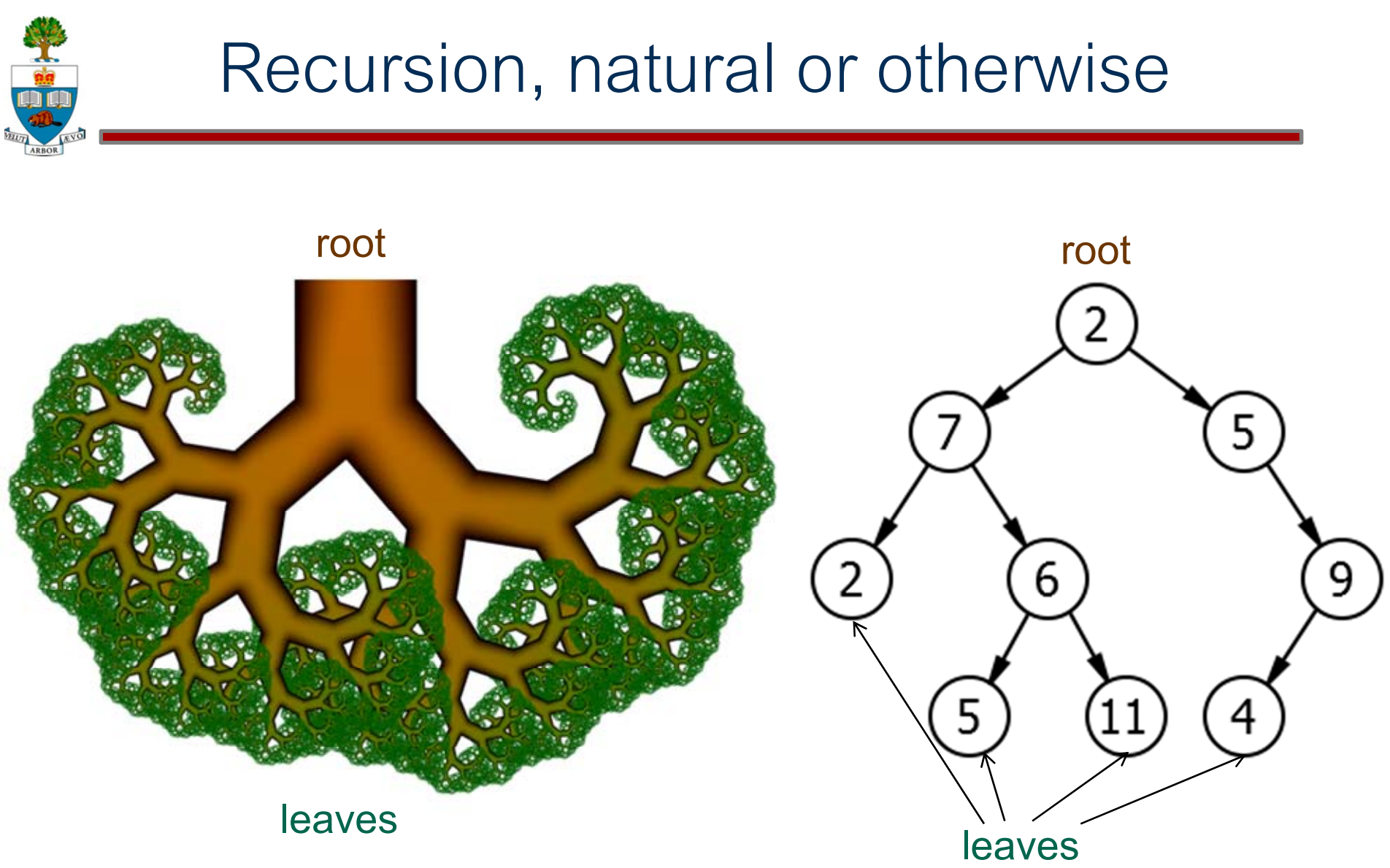
bogdan@cs.toronto.edu

http://www.cs.toronto.edu/~bogdan

University of Toronto, Department of Computer Science

Recursion, natural or otherwise

Recursion, natural or otherwise



Recursion, natural or otherwise

Tree terminology

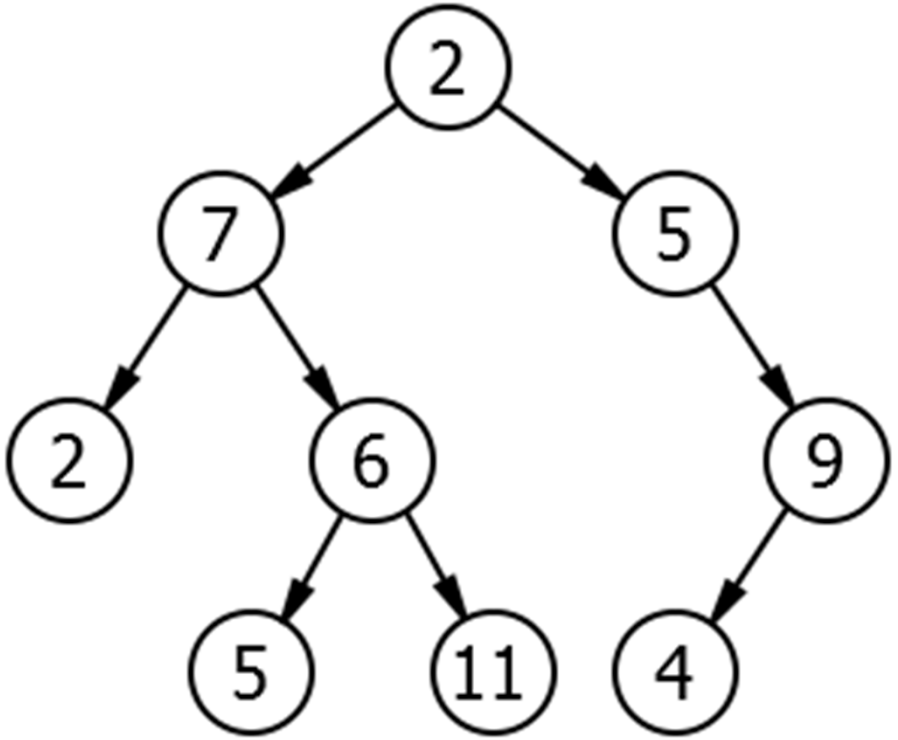
- Set of nodes (possibly with values or labels), with directed edges between some pairs of nodes
- One node is distinguished as root
- Each non-root node has exactly one parent
- Each node has zero or more children
- A path is a sequence of nodes $n_1, n_2, \ldots, n_k$, where there is an edge from n_i to n_{i+1}, $i<k$
- The length of a path is the number of edges in it
- There is a unique path from the root to each node. In the case of the root itself this is just n_1, if the root is node n_1
- There are no cycles; no paths that form loops.

More tree terminology

- leaf: node with no children
- internal node: node with one or more children
- subtree: tree formed by any tree node together with its descendants and the edges leading to them.
- height: 1+ the maximum path length in a tree. A node also has a height, which is 1+ the maximum path length of the tree rooted at that node
- depth: length of the path from the root to a node, so the root itself has depth 0
- arity, branching factor: maximum number of children for any node

Examples

- root?
- parent?
- children?
- leaves?
- internal nodes?
- subtree?
- path?
- height?
- depth?
- arity, branching factor?



General tree implementation

```
class Tree:
    """
    A bare-bones Tree ADT that identifies the root with the entire tree.
    """
    def __init__(self, value=None, children=None):
        """
        Create Tree self with content value and 0 or more children

        @param Tree self: this tree
        @param object value: value contained in this tree
        @param list[Tree|None] children: possibly-empty list of children
        @rtype: None
        """
        self.value = value
        # copy children if not None
        self.children = children.copy() if children else []
```

- How does this compare to the LinkedList/LinkedListNode?
- Any major difference you can spot here?

General form of recursion

if <condition to detect a base case>:

    <do something without recursion>

else: *# <general case / recursive step>*

    <do something that involves recursive call(s)>

Special methods

- Implement __eq__, __str__
  - When are two trees equivalent?
  - String representation of a Tree object?

- Helper function: descendants_from_list(t, list_, arity)
  - populate a Tree t with items from list_, arity children per node
  - not a method of Tree class

- Implement __contains__
  - Search for a given value in the Tree..

How many leaves?

```
def leaf_count(t):
    """
    Return the number of leaves in Tree t.
    @param Tree t: tree to count number of leaves of
    @rtype: int
    >>> t = Tree(7)
    >>> leaf_count(t)
    1
    >>> t = descendants_from_list(Tree(7),
                                  [0, 1, 3, 5, 7, 9, 11, 13], 3)
    >>> leaf_count(t)
    6
    """
    pass
```

Idea: if t is a leaf => 1

otherwise => sum the number of leaves in t's children

Height of this Tree

```
def height(t):
    """
    Return 1 + length of the longest path of t.
    @param Tree t: tree to find the height of
    @rtype: int
    >>> t = Tree(13)
    >>> height(t)
    1
    >>> t = descendants_from_list(Tree(13),
                                  [0, 1, 3, 5, 7, 9, 11, 13], 3)
    >>> height(t)
    6
    """
    # 1 more edge than the maximum height of a child, except
    # what do we do if there are no children?
    pass
```

Idea: if t is a leaf => 1

otherwise => 1 + max of the heights of t's children

Number of nodes in a tree

```
def count(t):
    """
    Return the number of nodes in Tree t.
    @param Tree t: tree to find the number of nodes in
    @rtype: int
     >>> t = Tree(17)
     >>> count(t)
     1
     >>> t4 = descendants_from_list(Tree(17),
                                    [0, 2, 4, 5, 8, 10, 11], 4)
    >>> count(t)
    8
    """
    pass
```

Idea: if t is a leaf => 1

otherwise => 1 + the number of nodes in t's children

Arity, branch factor

```
def arity(t):
    """
    Return the maximum branching factor (arity) of Tree t.
    @param Tree t: tree to find the arity of
    @rtype: int
     >>> t = Tree(23)
     >>> arity(t)
     0
     >>> tn2 = Tree(2, [Tree(4), Tree(4.5), Tree(5), Tree(5.75)])
     >>> tn3 = Tree(3, [Tree(6), Tree(7)])
     >>> tn1 = Tree(1, [tn2, tn3])
     >>> arity(tn1)
     4
    """
    pass
```

Idea: if t is a leaf => 0 (no further branching)

otherwise => max(number of children, children's arities)

Work on it in the lab ...

Pass in a function

```
def list_if(t, p):
    """
    Return a list of values in Tree t that satisfy predicate p(value).
    Assume predicate p is defined on t's values.

    @param Tree t: tree to list values that satisfy predicate p
    @param (object)->bool p: predicate to check values with
    @rtype: list[object]

    >>> def p(v): return v > 4
    >>> t = descendants_from_list(Tree(0), [1, 2, 3, 4, 5, 6, 7, 8], 3)
    >>> list_ = list_if(t, p)
    >>> list_.sort()
    >>> list_
    [5, 6, 7, 8]
    >>> def p(v): return v % 2 == 0
    >>> list_ = list_if(t, p)
    >>> list_.sort()
    >>> list_
    [0, 2, 4, 6, 8]
    """
    pass
```

Practice at home!

Idea: if leaf => [t.value] if p(t.value) else []

otherwise => ([t.value] if p(t.value) else []) +

gather_lists([list_if(c, p) for c in t.children]))

List the leaves

```
def list_leaves(t):
    """
    Return a list of values in the leaves of Tree t

    @param Tree t: tree to list the leaf values of
    @rtype: list[object]

    >>> t = Tree(0)
    >>> list_leaves(t)
    [0]
    >>> t = descendants_from_list(Tree(0), [1, 2, 3, 4, 5, 6, 7, 8], 3)
    >>> list_ = list_leaves(t)
    >>> list_.sort()     # so list_ is predictable to compare
    >>> list_
    [3, 4, 5, 6, 7, 8]
    """
    pass
```

Idea: if leaf => [t.value]

otherwise => *gather_lists*([list_leaves(c) for c in t.children]))

Practice at home!

Traversal

- The functions and methods we have seen get information from every node of the tree -- in some sense they traverse the tree

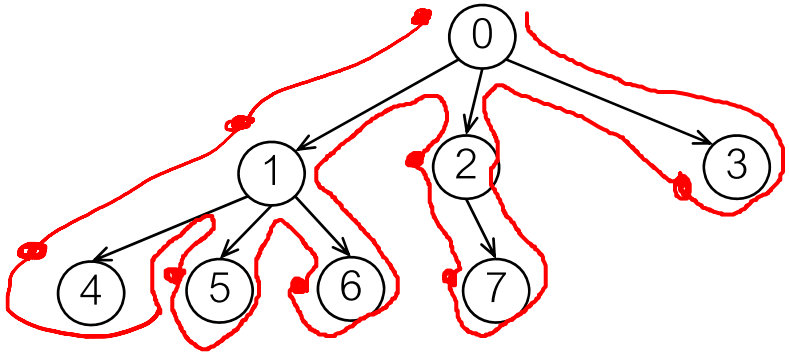
- Sometimes the order of processing tree nodes is important:
  - Do we process the root of the tree (and the root of each subtree...) before or after its children?
  - Or, perhaps, we process along levels that are the same distance from the root?

Preorder visit

- Visit each node of Tree t as follows:
  - do something with the node's value, e.g., print it
  - visit its first child in preorder
  - visit its second child in preorder
  - ...

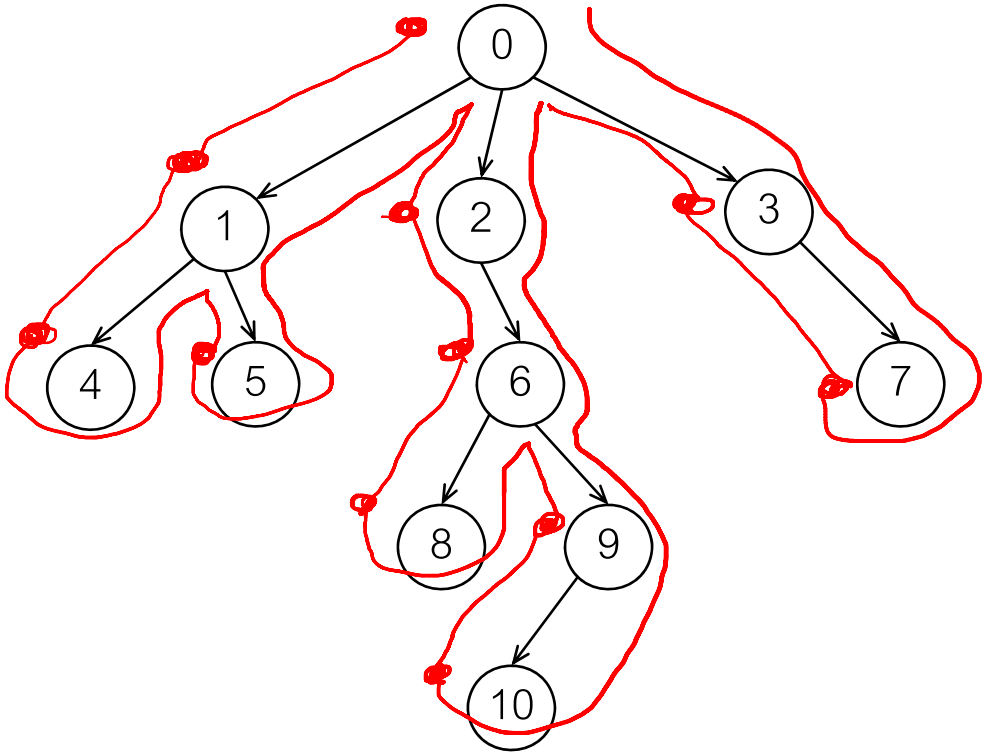
t = descendants_from_list(Tree(0), [1, 2, 3, 4, 5, 6, 7], 3)

What is the sequence of nodes being visited in preorder?



Preorder visit – more examples

- Visit each node of Tree t as follows:
  - do something with the node's value, e.g., print it
  - visit its first child in preorder
  - visit its second child in preorder
  - ...

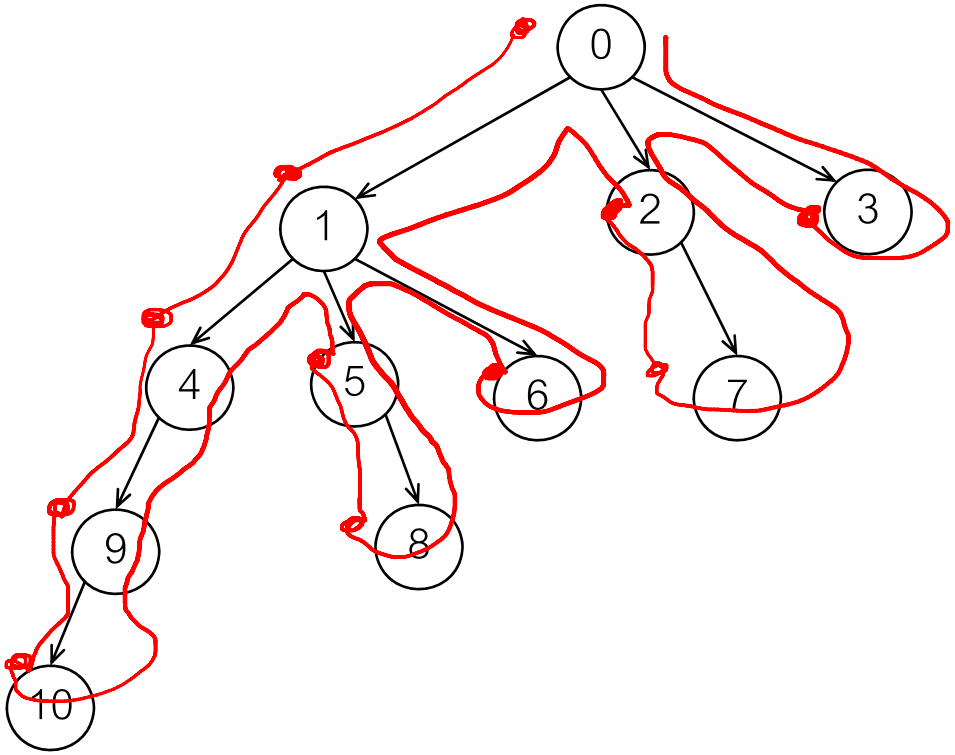


What is the sequence of nodes being visited in preorder?

0, 1, 4, 5, 2, 6, 8, 9, 10, 3, 7

Preorder visit – more examples

- Visit each node of Tree t as follows:
  - do something with the node's value, e.g., print it
  - visit its first child in preorder
  - visit its second child in preorder
  - ...



What is the sequence of nodes being visited in preorder?

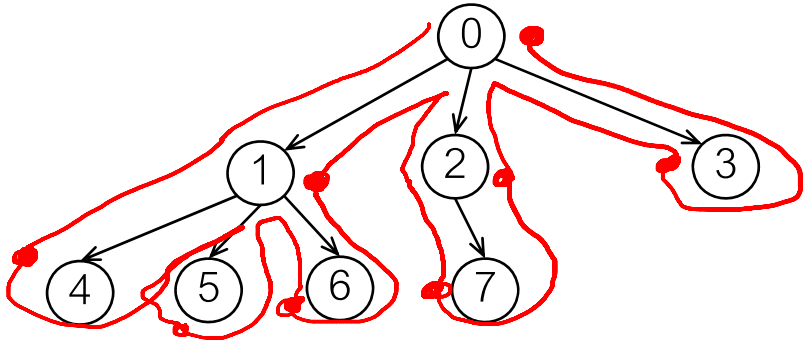
Any thoughts on how to implement this visit order?

Postorder visit

- Visit each node of Tree t as follows:
  - visit its first child in postorder
  - visit its second child in postorder
  - ...
  - do something with the node's value, e.g., print it

t = descendants_from_list(Tree(0), [1, 2, 3, 4, 5, 6, 7], 3)

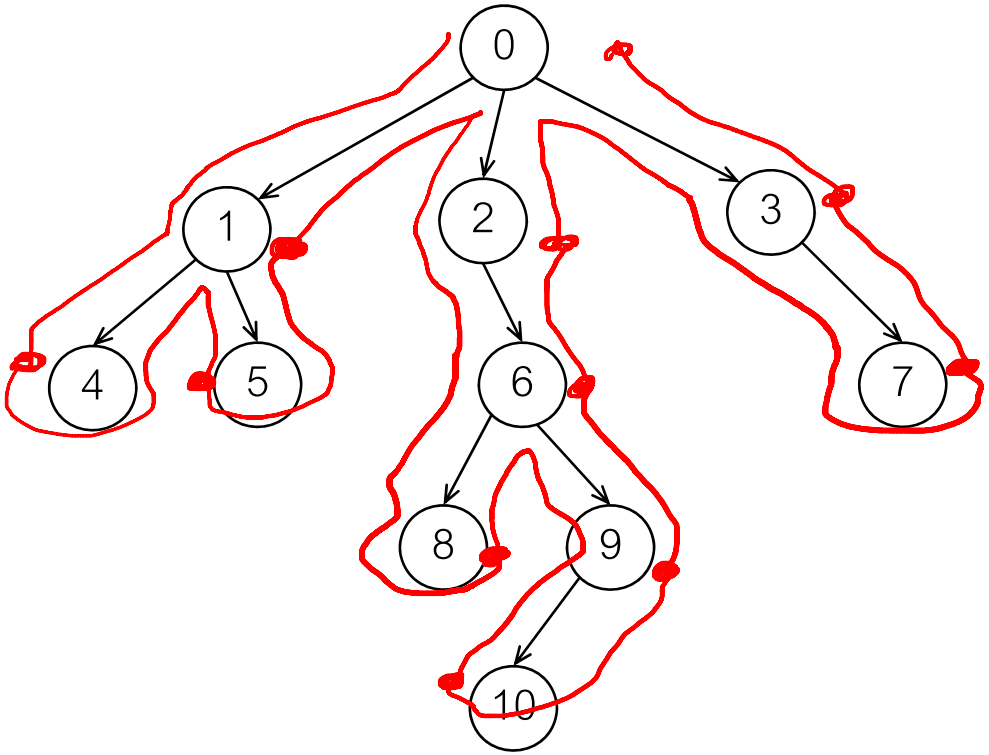
What is the sequence of nodes being visited in preorder?



4, 5, 6, 1, 7, 2, 3, 0

Postorder visit – more examples

- Visit each node of Tree t as follows:
  - visit its first child in postorder
  - visit its second child in postorder
  - ...
  - do something with the node's value, e.g., print it

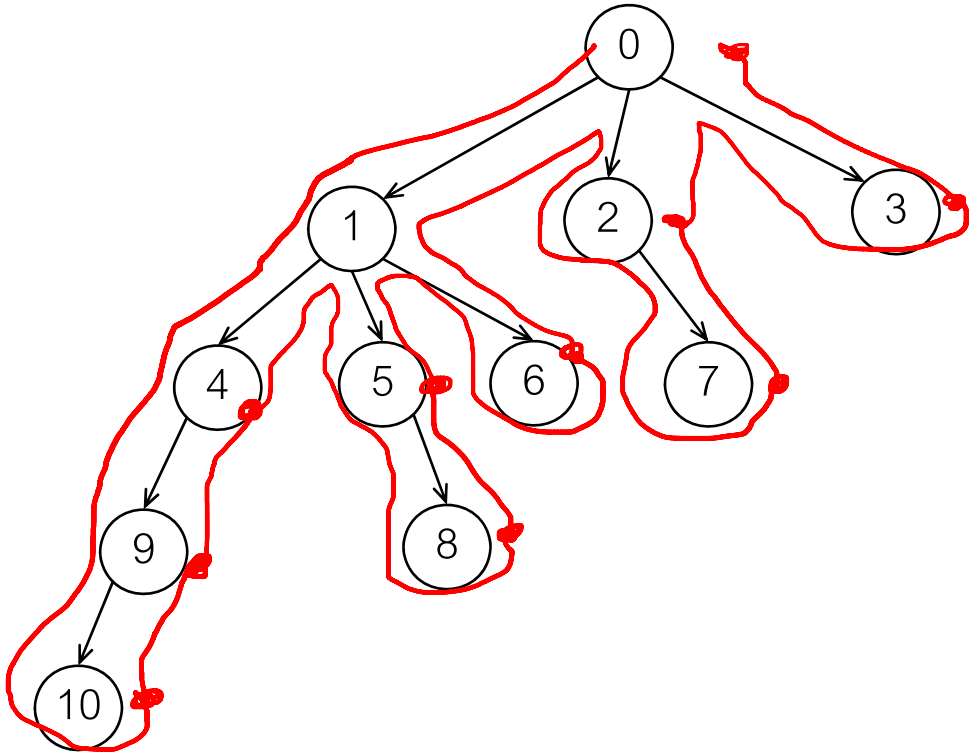


What is the sequence of nodes being visited in preorder?

4, 5, 1, 8, 10, 9, 6, 2, 7, 3, 0

Postorder visit – more examples

- Visit each node of Tree t as follows:
  - visit its first child in postorder
  - visit its second child in postorder
  - ...
  - do something with the node's value, e.g., print it



What is the sequence of nodes being visited in preorder?

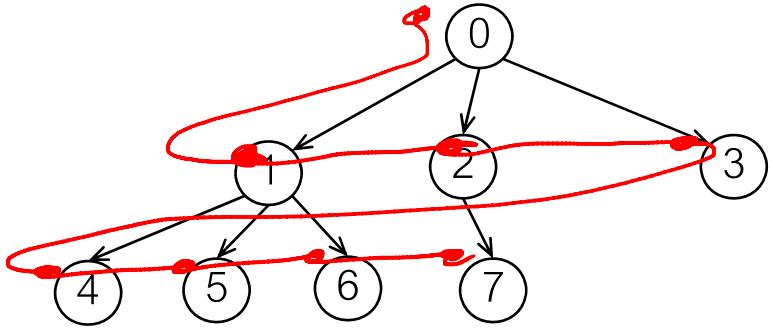
Any thoughts on how to implement this visit order?

Level order visit

- Visit each node of Tree t as follows:
  - do something with the node's value, e.g., print it
  - visit all its children (first level in the tree) and act on the nodes
  - visit all the children's children (second level in the tree) and act on the nodes
  - visit third level in the tree, etc..

t = descendants_from_list(Tree(0), [1, 2, 3, 4, 5, 6, 7], 3)

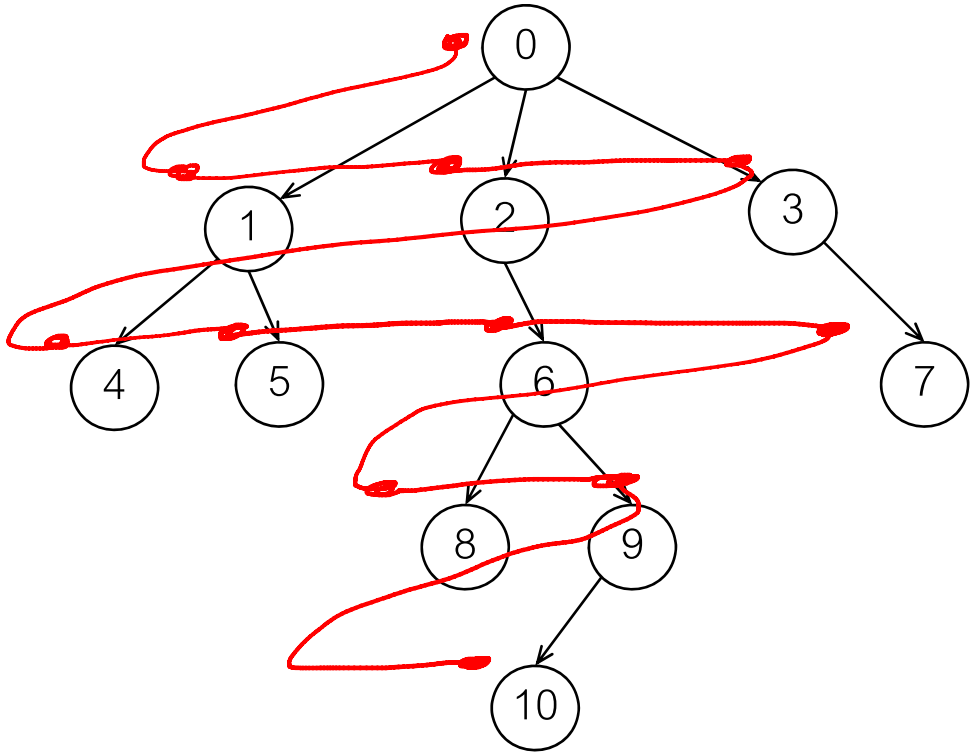
What is the sequence of nodes being visited in preorder?



Level order visit

- Visit each node of Tree t as follows:
  - do something with the node's value, e.g., print it
  - visit all its children (first level in the tree) and act on the nodes
  - visit all the children's children (second level in the tree) and act on the nodes
  - visit third level in the tree, etc..

What is the sequence of nodes being visited in preorder?

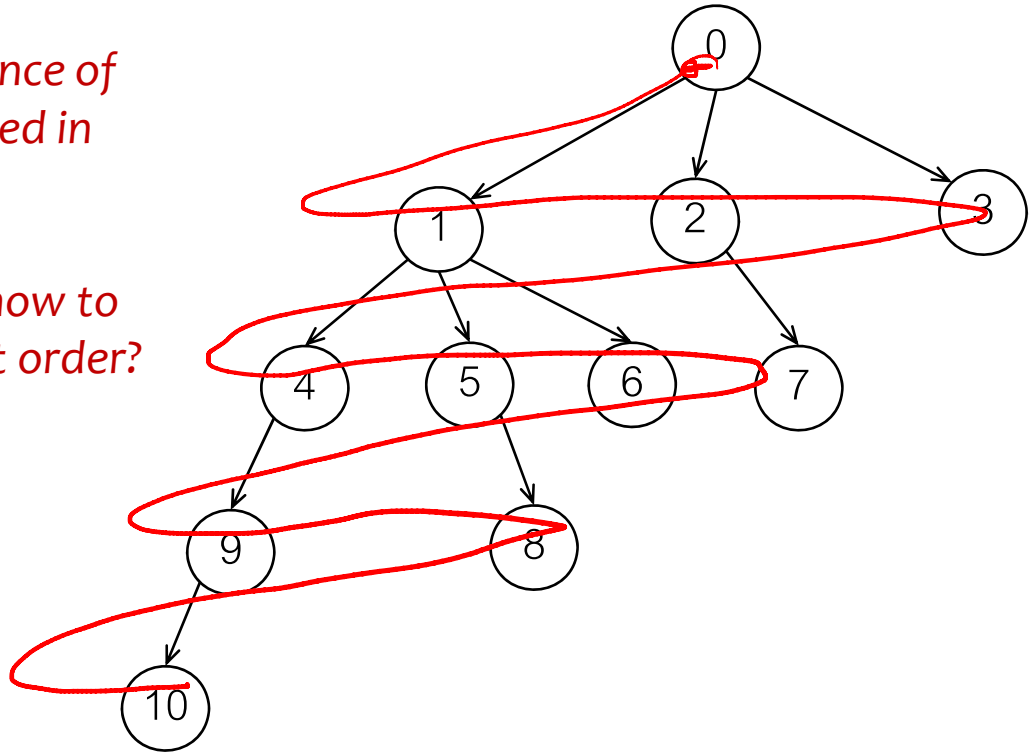


Level order visit

- Visit each node of Tree t as follows:
  - do something with the node's value, e.g., print it
  - visit all its children (first level in the tree) and act on the nodes
  - visit all the children's children (second level in the tree) and act on the nodes
  - visit third level in the tree, etc..

What is the sequence of nodes being visited in preorder?

Any thoughts on how to implement this visit order?



Queues, stacks, recursion

- You may have noticed in the code for level order visit that there were no recursive calls, and a queue was used to process a recursive structure in level order

- Careful use of a stack allows you to process a tree in preorder or postorder (no recursion needed)