CSC 148 Winter 2017

Week 8

Binary trees

Bogdan Simion

bogdan@cs.toronto.edu

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

University of Toronto, Department of Computer Science

Outline

- Binary Trees (arity = 2)
- Binary Tree Traversals
- Binary *Search* Trees (BST)

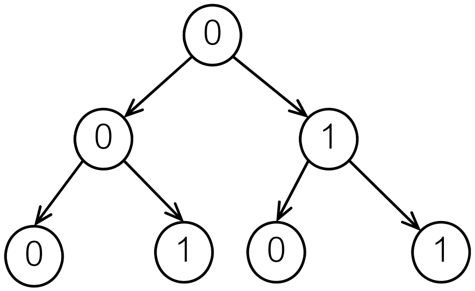
General tree implementation

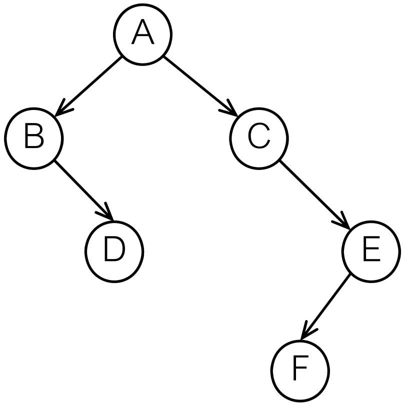
- Change our generic Tree design so that we have two named children, left and right, and can represent an empty tree with None

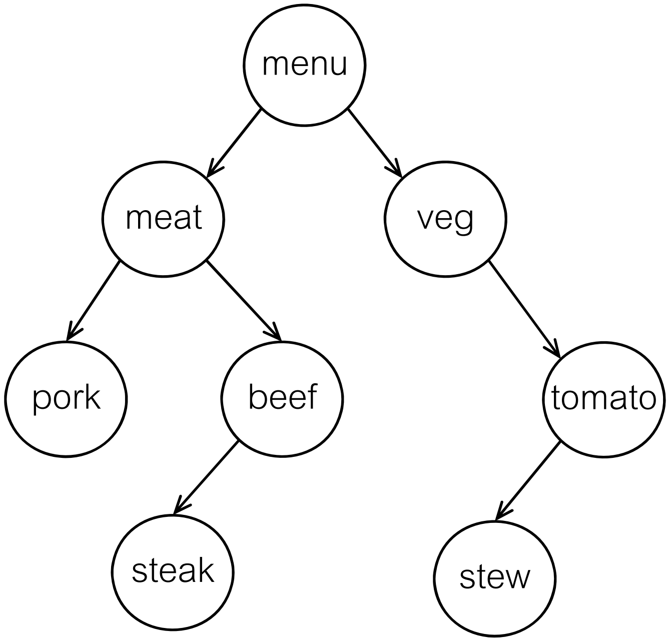
```
class BinaryTree:
    """
    A BinaryTree, i.e., arity 2.
    """
    def __init__(self, data, left=None, right=None):
        """
        Create BinaryTree self with data  and children left and right

        @param BinaryTree self: this binary tree
        @param object data: data of this node
        @param BinaryTree|None left: left child
        @param BinaryTree|None right: right child
        @rtype: None
        """
        self.data, self.left, self.right = data, left, right
```

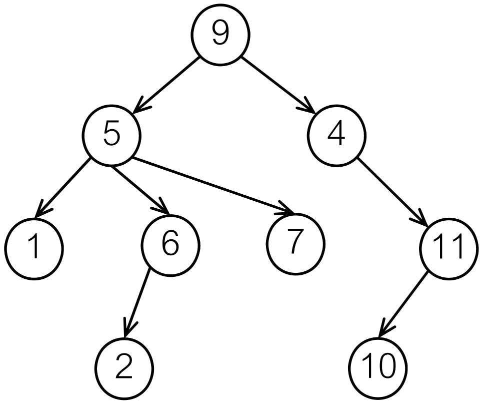
Binary tree examples

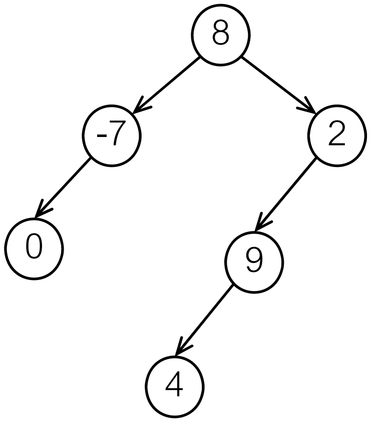
- Spot the non-binary trees:

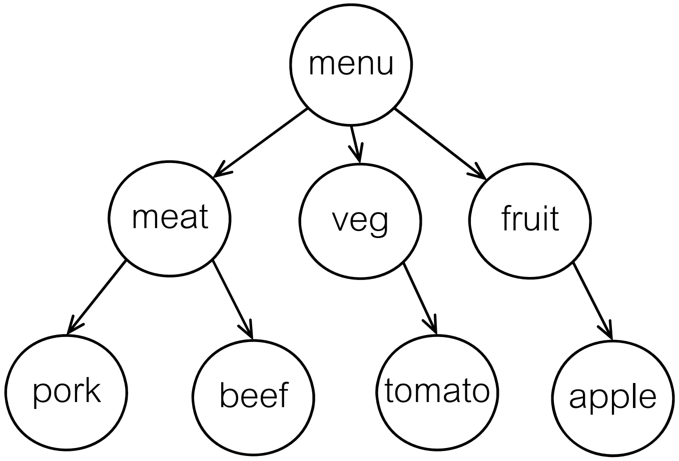












Special methods ...

- We'll want the standard special methods:
  - `__eq__`
  - `__str__`
  - `__repr__`

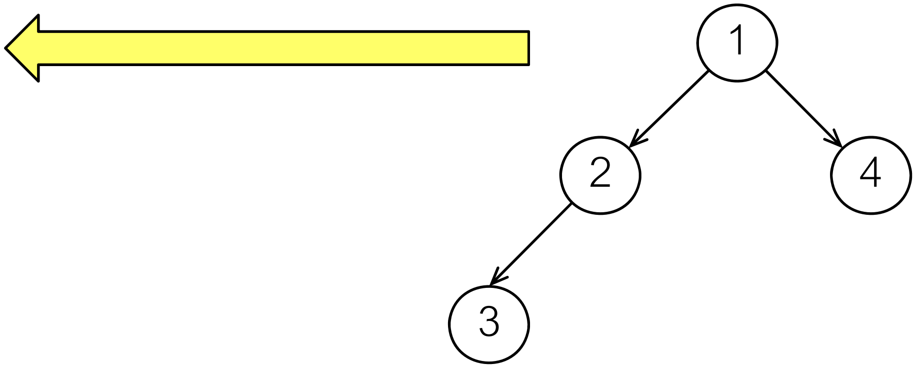
Special methods (eq)

```
def __eq__(self, other):
    """
    Return whether BinaryTree self is equivalent to other

    @param BinaryTree self: this binary tree
    @param Any other: object to check equivalence to self
    @rtype: bool
    >>> BinaryTree(7).__eq__("seven")
    False
    >>> b1 = BinaryTree(7, BinaryTree(5))
    >>> b1.__eq__(BinaryTree(7, BinaryTree(5), None))
    True
     """
     pass
```

When are two trees equivalent?

Special methods (str)

```
def __str__(self, indent=“”):
    """
    Return a user-friendly string representing BinaryTree (self) inorder.  Indent by 'indent'.

    >>> b = BinaryTree(1, BinaryTree(2, BinaryTree(3)), BinaryTree(4))
    >>> print(b)
        4
    1
        2
            3
    <BLANKLINE>
    """
    pass
```



String representation ... indent each node accordingly.

Special methods (repr)

```
def __repr__(self):
    """
    Represent BinaryTree (self) as a string that can be evaluated to produce an
    equivalent BinaryTree.

    @param BinaryTree self: this binary tree
    @rtype: str

    >>> BinaryTree(1, BinaryTree(2), BinaryTree(3))
    BinaryTree(1, BinaryTree(2, None, None), BinaryTree(3, None, None))
    """
    pass
```

Contains

- You've implemented contains on linked lists, nested Python lists, general Trees before; implement this function, then modify it to become a method

```
def contains(node, value):
    """
    Return whether tree rooted at node contains value.

    @param BinaryTree|None node: binary tree to search for value
    @param object value: value to search for
    @rtype: bool

    >>> contains(None, 5)
    False
    >>> contains(BinaryTree(5, BinaryTree(7), BinaryTree(9)), 7)
    True
    """
    pass
```

Idea: Empty tree => False
Otherwise => node.value == value
or contains(node.left, value)
or contains(node.right, value)

Height of a Binary Tree

```
def height(t):
    """
    Return 1 + length of the longest path of t.
    @param BinaryTree t: binary tree to find the height of
    @rtype: int
    >>> t = BinaryTree(13)
    >>> height(t)
     1
    """
    pass
```

Idea: if t is a leaf => 1

otherwise => 1 + max of the heights left and right

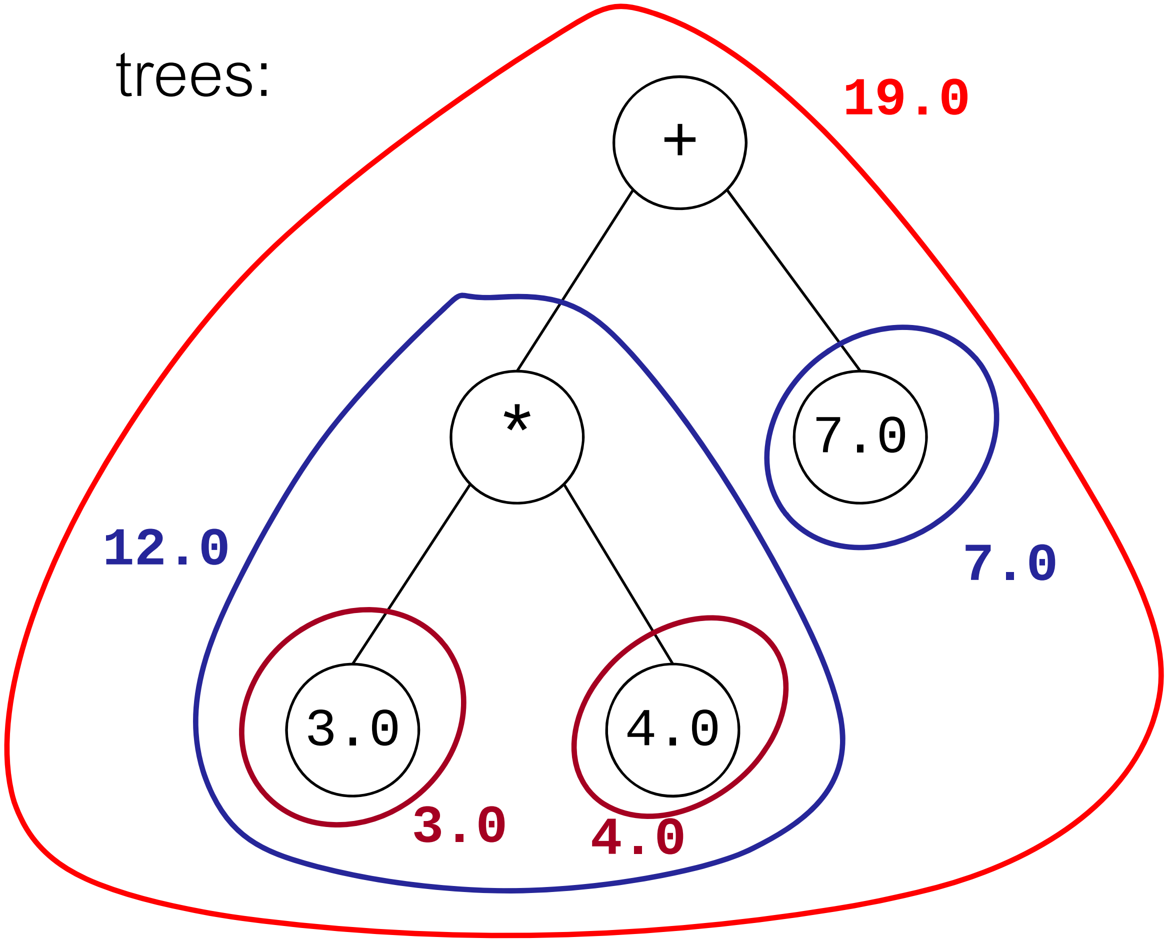
Or:

check if t is None => return 0

otherwise => return 1 + max of the heights of left and right

Arithmetic expression trees

- Binary arithmetic expressions can be represented as binary trees:



What's the strategy to evaluate an expression from a tree?

Evaluating a binary expression tree

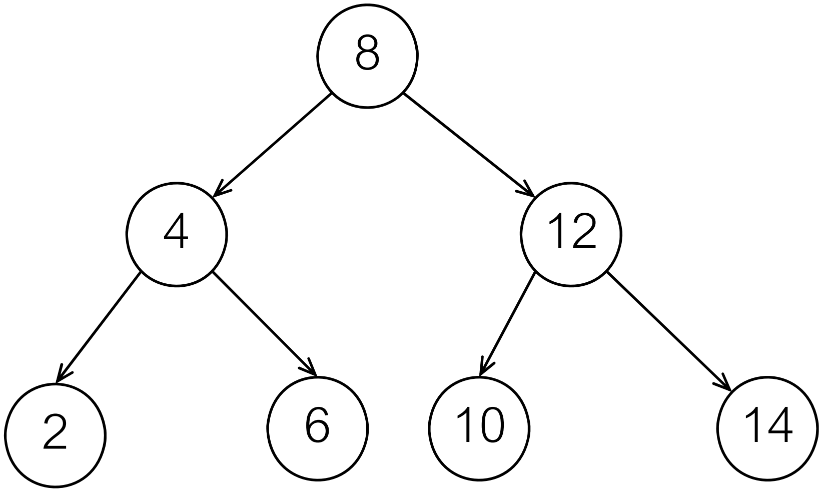
- There are no empty expressions
  - If it’s a leaf, just return the value
  - Otherwise ...
    - 1. Evaluate the left tree
    - 2. Evaluate the right tree
    - 3. Combine left and right with the binary operator
- Python built-in **eval** might be handy

```
>>> eval(“148 + 17”)
165
```

Evaluating a binary expression tree

```
def evaluate(b):
    """ Evaluate the expression rooted at b.  If b is a leaf, return its float data.  Otherwise,
        evaluate  b.left and b.right and combine them with b.data.

    Assume:  -- b is a non-empty binary tree
             -- interior nodes contain data in {"+", "-", "*", "/"}
             -- interior nodes always have two children
             -- leaves contain float data

     @param BinaryTree b: binary tree representing arithmetic expression
     @rtype: float

    >>> b = BinaryTree(3.0)
    >>> evaluate(b)
    3.0
    >>> b = BinaryTree("*", BinaryTree(3.0), BinaryTree(4.0))
    >>> evaluate(b)
    12.0
    """
```

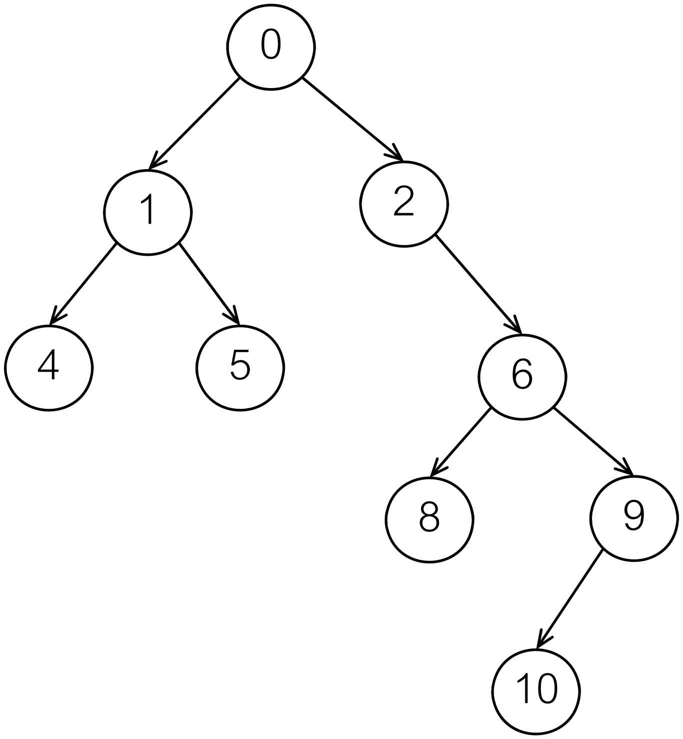
inorder



- A recursive definition:
  - visit the left subtree inorder
  - visit this node itself
  - visit the right subtree inorder
- The code is almost identical to the definition

inorder - examples

- A recursive definition:
  - visit the left subtree inorder
  - visit this node itself
  - visit the right subtree inorder

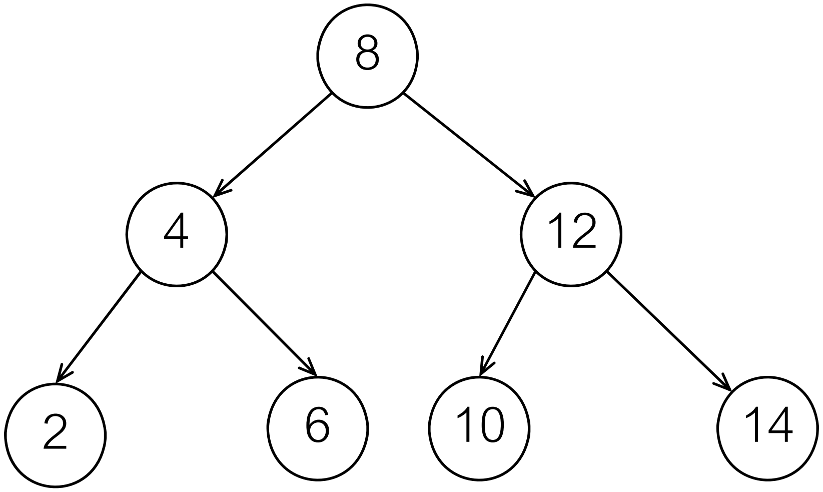


What is the sequence of nodes being visited in inorder?

inorder

```
def inorder_visit(root, act):
    """ Visit each node of binary tree rooted at root in order and act.
    @param BinaryTree root: binary tree to visit
    @param (BinaryTree)->object act: function to execute on visit (e.g., display)
    @rtype: None

     >>> b = BinaryTree(8)
     >>> b = insert(b, 4)
     >>> b = insert(b, 2)
     >>> b = insert(b, 6)
     >>> b = insert(b, 12)
     >>> def display(node): print(node.data, “  ”, end=“”)
     >>> inorder_visit(b, display)
     2   4   6   8   12
     """
     pass
```

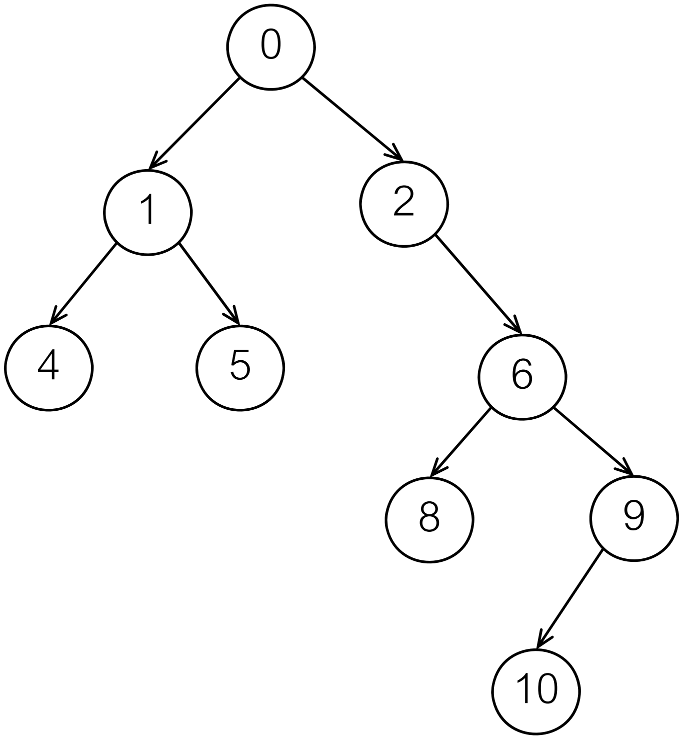
preorder



- visit this node itself
- visit the left subtree in preorder
- visit the right subtree in preorder
- Similar to general trees, except max 2 children

preorder - examples

- A <u>recursive</u> definition:
  - visit this node itself
  - visit the left subtree in preorder
  - visit the right subtree in preorder

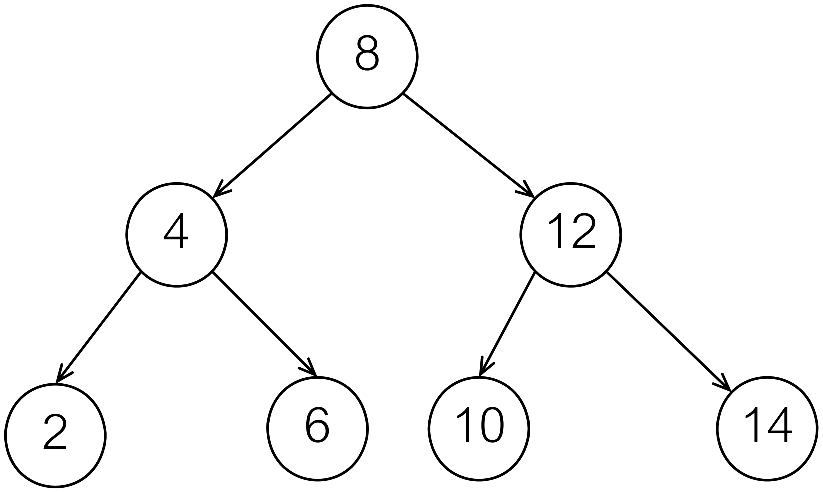


What is the sequence of nodes being visited in inorder?

preorder

```
def preorder_visit(root, act):
    """ Visit each node of binary tree rooted at root in order and act.
    @param BinaryTree root: binary tree to visit
    @param (BinaryTree)->object act: function to execute on visit (e.g., display)
    @rtype: None

     >>> b = BinaryTree(8)
     >>> b = insert(b, 4)
     >>> b = insert(b, 2)
     >>> b = insert(b, 6)
     >>> b = insert(b, 12)
     >>> def display(node): print(node.data, “  ”, end=“”)
     >>> preorder_visit(b, display)
     8   4   2   6   12
     """
     pass
```

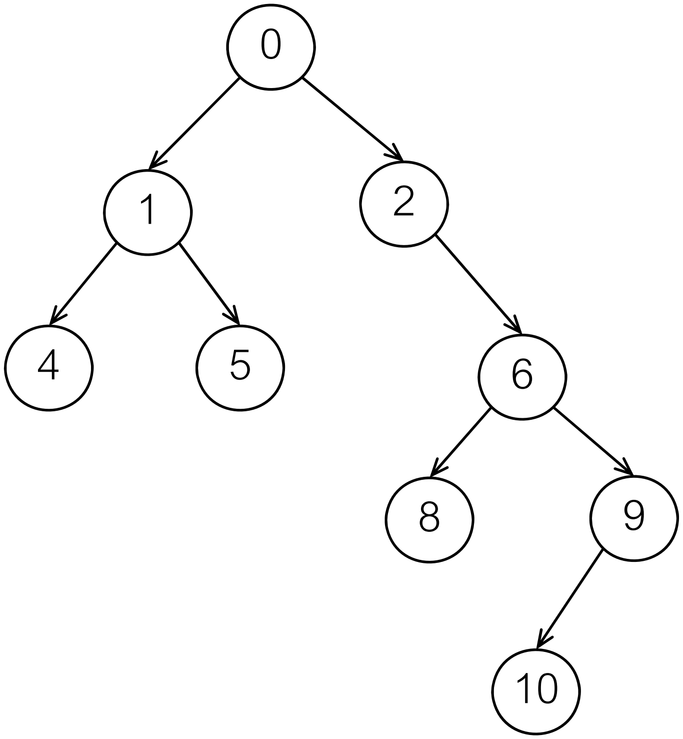
postorder



- visit the left subtree in postorder
- visit the right subtree in postorder
- visit this node itself
- Similar to general trees, except max 2 children

postorder - examples

- A recursive definition:
  - visit the left subtree in postorder
  - visit the right subtree in postorder
  - visit this node itself

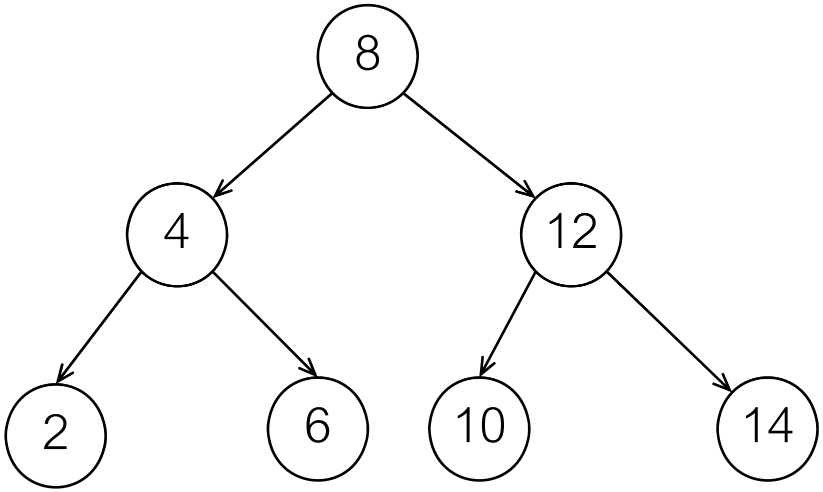


What is the sequence of nodes being visited in inorder?

postorder

```
def postorder_visit(root, act):
   """ Visit each node of binary tree rooted at root in order and act.
   @param BinaryTree root: binary tree to visit
   @param (BinaryTree)->object act: function to execute on visit (e.g., display)
   @rtype: None

    >>> b = BinaryTree(8)
    >>> b = insert(b, 4)
    >>> b = insert(b, 2)
    >>> b = insert(b, 6)
    >>> b = insert(b, 12)
    >>> def display(node): print(node.data, “  ”, end=“”)
    >>> postorder_visit(b, display)
    2   6   4   12   8
    """
    pass
```

level order



- visit this node itself
- visit this node’s children
- visit this node’s grandchildren
- visit this node’s great grandchildren
- ...
- Similar to general trees, except max 2 children

level order

```
def levelorder_visit(t, act):
    """ Visit BinaryTree t in level order and act on each node.
    @param BinaryTree|None t: binary tree to visit
    @param (BinaryTree)->Any act: function to execute on nodes during visit (e.g., display)
    @rtype: None

    >>> b = BinaryTree(8)
    >>> b = insert(b, 4)
    >>> b = insert(b, 2)
    >>> b = insert(b, 6)
    >>> b = insert(b, 12)
    >>> def display(node): print(node.data, "  ", end="")
    >>> levelorder_visit(b, display)
    8   4   12   2   6
    """
    pass
```

Thoughts? How do we implement it?

Tracing revisited

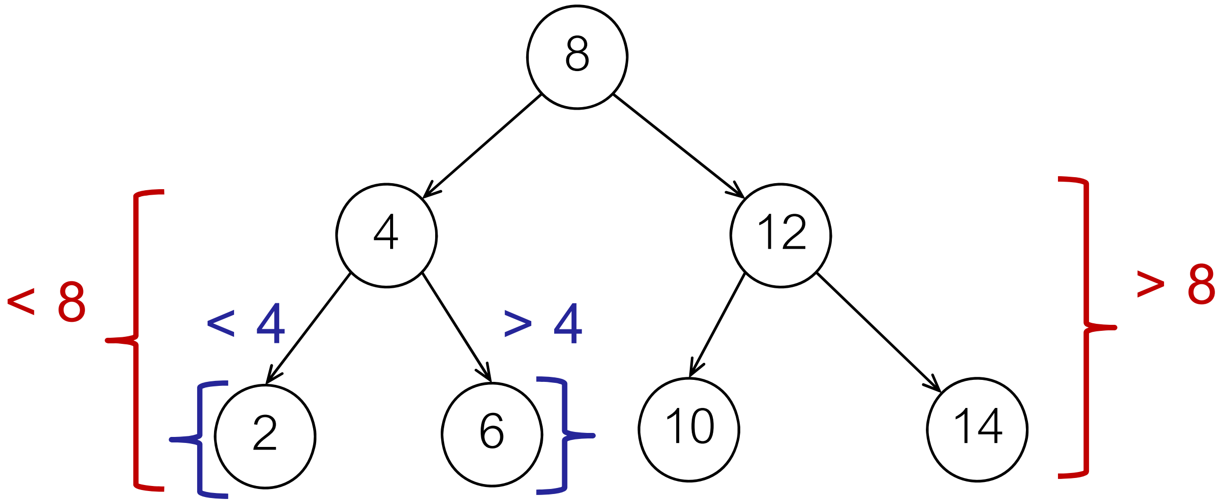
- Recursive version using iterative deepening ..
- You might be a bit bewildered by the execution of

  def visit_level(t, n, act), which means tracing is needed...

  - trace visit_level(None, 7, act) (for any function act you devise)
  - trace visit_level(t, 0, act) (for some BinaryTree with a few levels)
  - trace visit_level(t, 1, act) (for some BinaryTree with a few levels)
  - trace visit_level(t, 2, act) (for some BinaryTree with a few levels)
  - trace visit_level(t, 3, act) (for some BinaryTree with a few levels)
  - ...

Binary Search Tree - Definition

- Add ordering conditions to a binary tree:
  - data are comparable
  - data in left subtree are less than node.data
  - data in right subtree are more than node.data


8
4
12
< 8
< 4
> 4
> 8
2
6
10
14