CSC 148 Winter 2017

Week 9

Binary Search Trees (BSTs)

Bogdan Simion

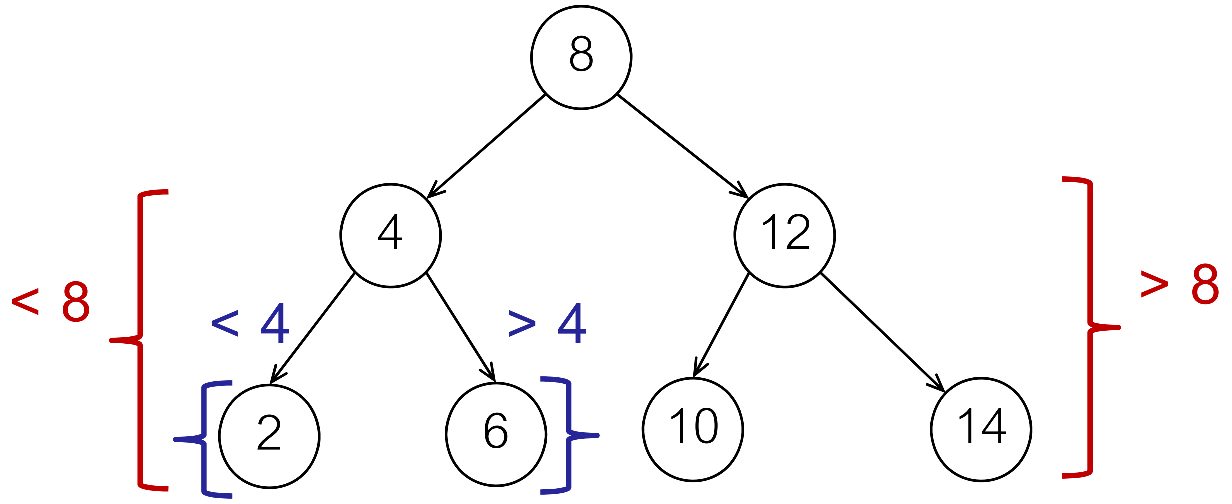
bogdan@cs.toronto.edu

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

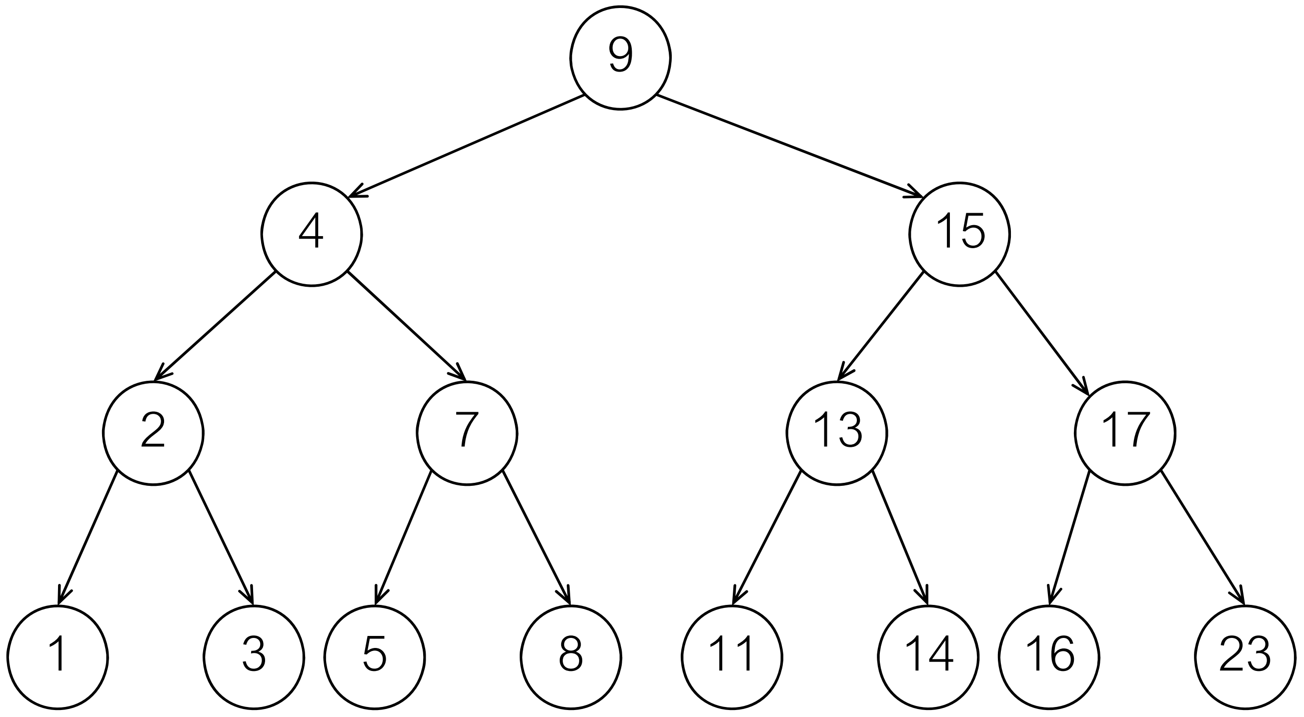
University of Toronto, Department of Computer Science

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



Find a value in a **regular** Binary Tree



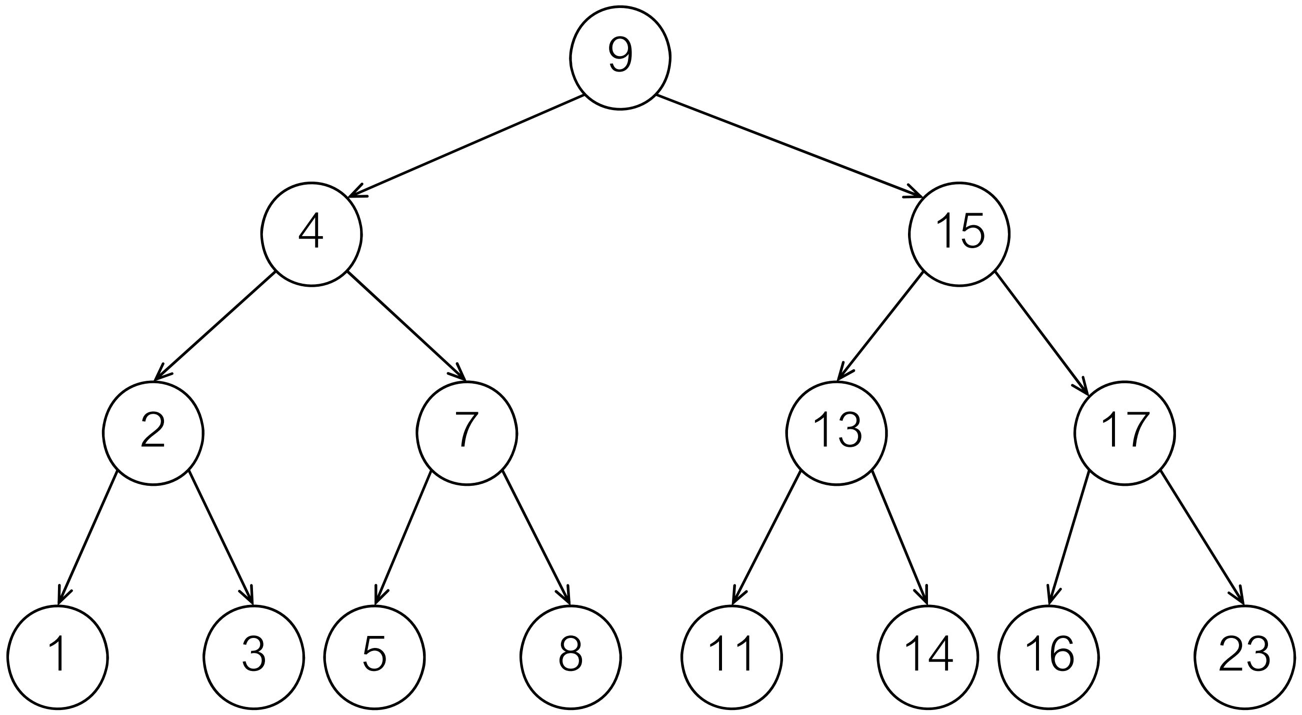
How many nodes do we visit (say, in preorder) to find out the following:

Find value 5, if present...

Find value 13, if present...

Find value 12, if present...

Find a value in a BST



How many nodes would we visit now, to find out the following:

Find value 5, if present...

Find value 13, if present...

Find value 12, if present...

Why binary search trees?

- Searches that are directed along a single path are efficient:
  - a BST with 1 node has height 1
  - a BST with 3 nodes may have height 2
  - a BST with 7 nodes may have height 3
  - a BST with 15 nodes may have height 4
  - a BST with n nodes may have height $\lceil lg\ n \rceil$.

1,000,000 nodes => height < 20!

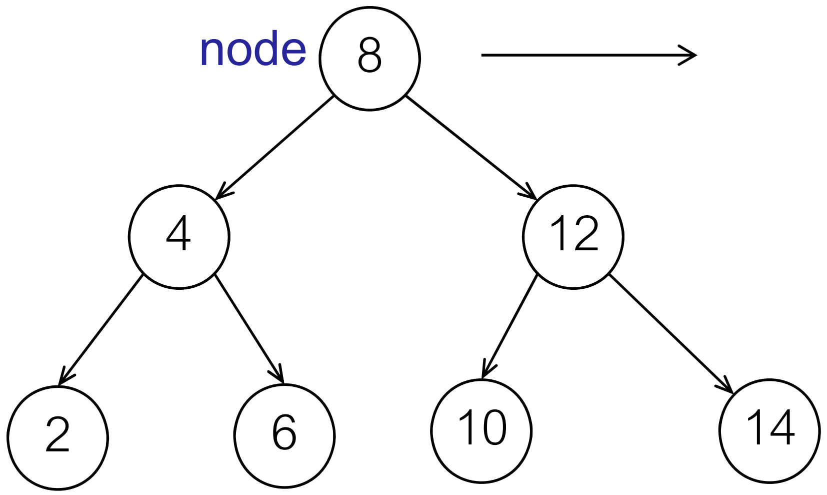
- If the BST is “balanced”, then we can check whether an element is present in about $lg\ n$ node accesses.

bst_contains

```
def bst_contains(node, value):
    """ Return whether tree rooted at node contains value.  Assume node is  the root of a BST.

    @param BinaryTree|None node: node of a Binary Search Tree
    @param object value: value to search for
    @rtype: bool

    >>> bst_contains(None, 5)
    False
    >>> bst_contains(BinaryTree(7, BinaryTree(5), BinaryTree(9)), 5)
    True
    """
    # use BST property to avoid unnecessary searching
    pass
```



Trace each, starting at node:

bst_contains(node, 6)?

bst_contains(node, 10)?

bst_contains(node, 7)?

Reminders: Test 2

- Coverage:
  - Linked Lists
  - Trees
  - Binary Trees
  - Binary Search Trees (no mutation operations)
- Recursion is fair game given the topics ..

Assignment 2 tips

- Compression and decompression program
  - Take any file and compress it
  - Take a compressed file you made and uncompress it back to the original file
- Huffman's algorithm is used to organize the file compression
  - Gives short codes to frequent symbols and longer codes to infrequent symbols

Binary Files and Bytes

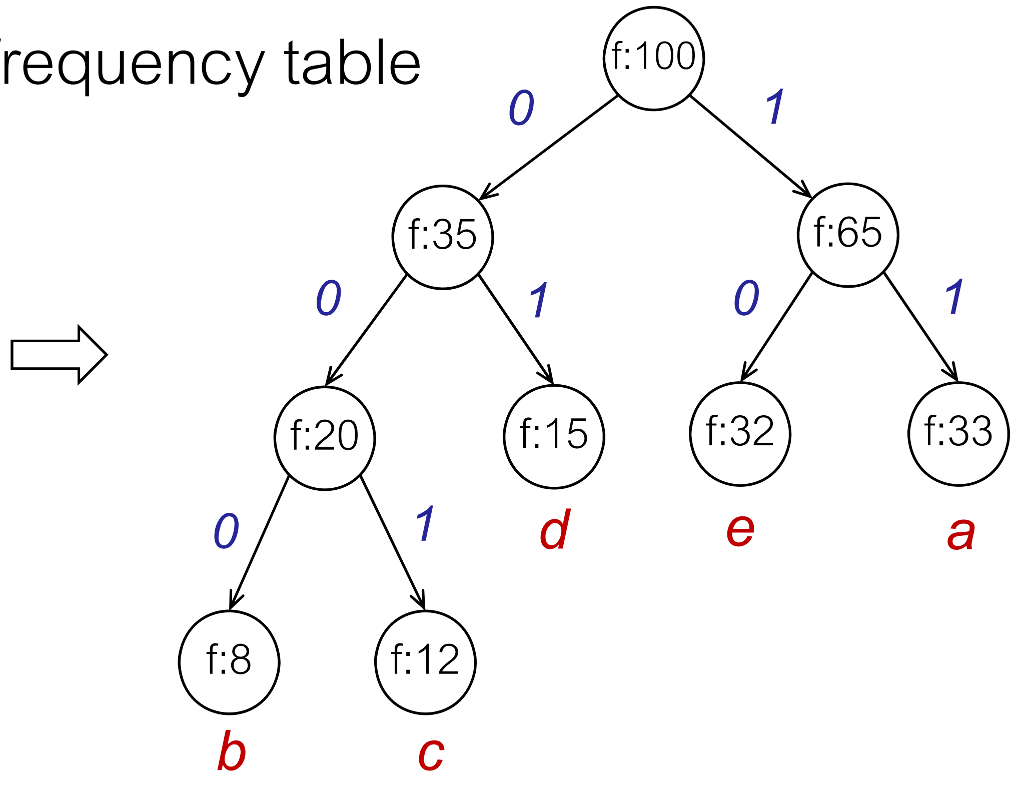
- We want to be able to compress and uncompress any kind of file: text, sound, image, executable, etc.
- Open file in binary mode, so that we can read its contents as bytes (not text)
- A byte is an integer in the range 0-255
- Use Python bytes objects to work with bytes

```
>>> b1 = bytes([80, 90, 100])
>>> b2 = bytes([5])
>>> list(b1 + b2)
[80, 90, 100, 5]
```

Huffman's algorithm

- Make sure to draw an example of Huffman's algorithm
- For example, take this frequency table

char	frequency
a	33
e	32
d	15
c	12
b	8

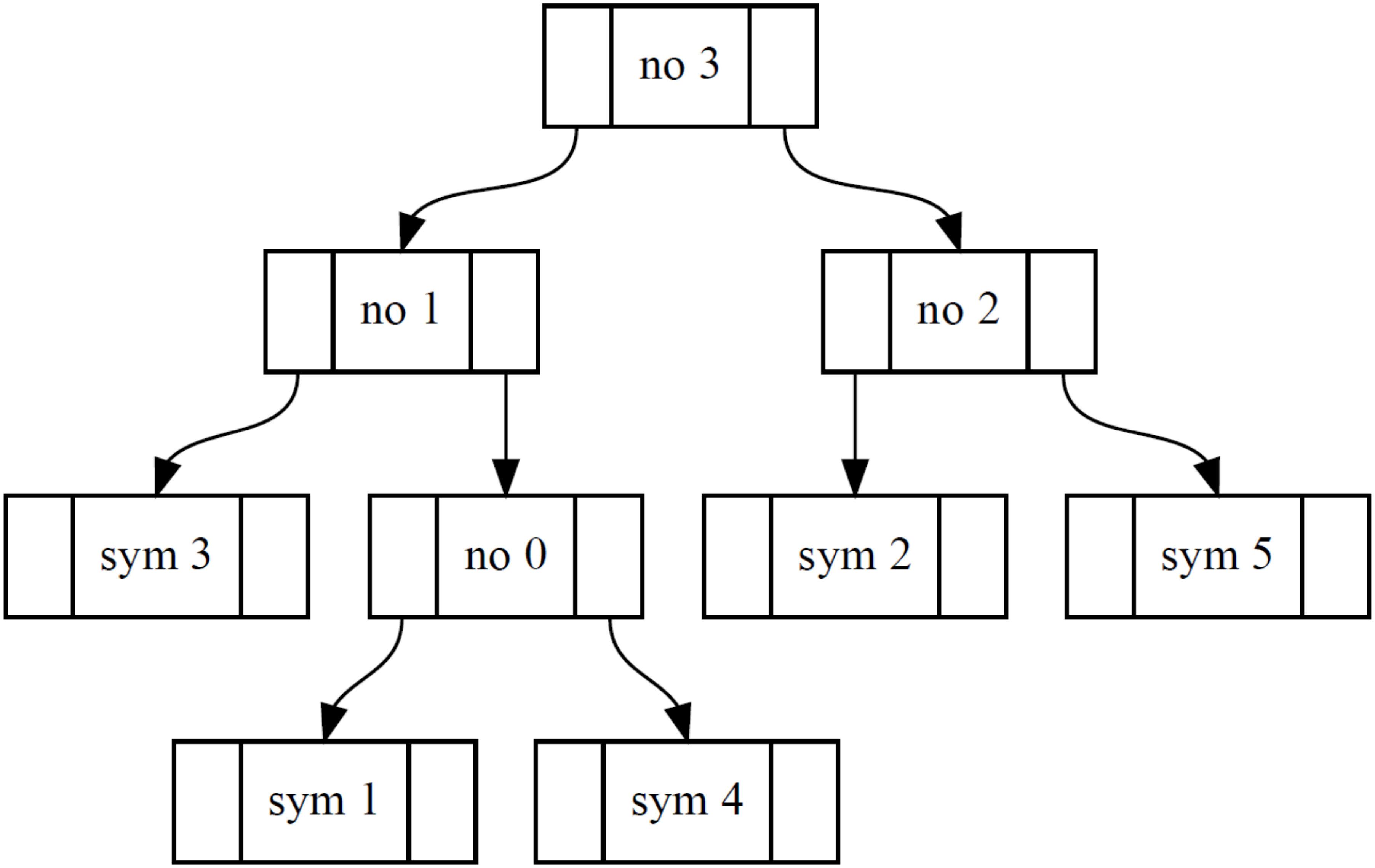


- The rule: at each step, merge the two smallest frequency trees!
- Read up on the full algorithm carefully!

Numbering nodes

- We're going to have to write a representation of a Huffman tree to a compressed file
- To do that, we will want a way to refer to nodes
- But, internal nodes have no name, so there is no way to refer to them
- This is why we number nodes
- We have decided on a postorder numbering (we could have chosen any numbering for the assignment)

Sample Numbered Huffman Tree



Codes for symbols

- Each symbol gets a code from the tree

char (byte)	frequency
a (61)	11
b (62)	000
c (63)	001
d (64)	01
e (65)	10

- Use that to compress this list of bytes: [63, 61, 65, 64]

Tree to Bytes

- With the tree numbered, we can now output a representation of the tree to a file
- If there are n internal nodes, then we write 4n bytes to the file to represent the tree
  - Each internal node takes 4 bytes
  - A 0 or 1 is written to signify a leaf or non-leaf subtree, respectively

Uncompressing

- To uncompress a file, we first have to obtain the Huffman tree that was used to compress it
- Without the tree, we have no idea of the mapping between codes and symbols
- The Huffman tree is stored in the file in postorder
- And we have the node numbers in there to help us reconstruct the tree
- We are asking you for two different functions for recovering a tree from a file ...

Recovering Tree in General

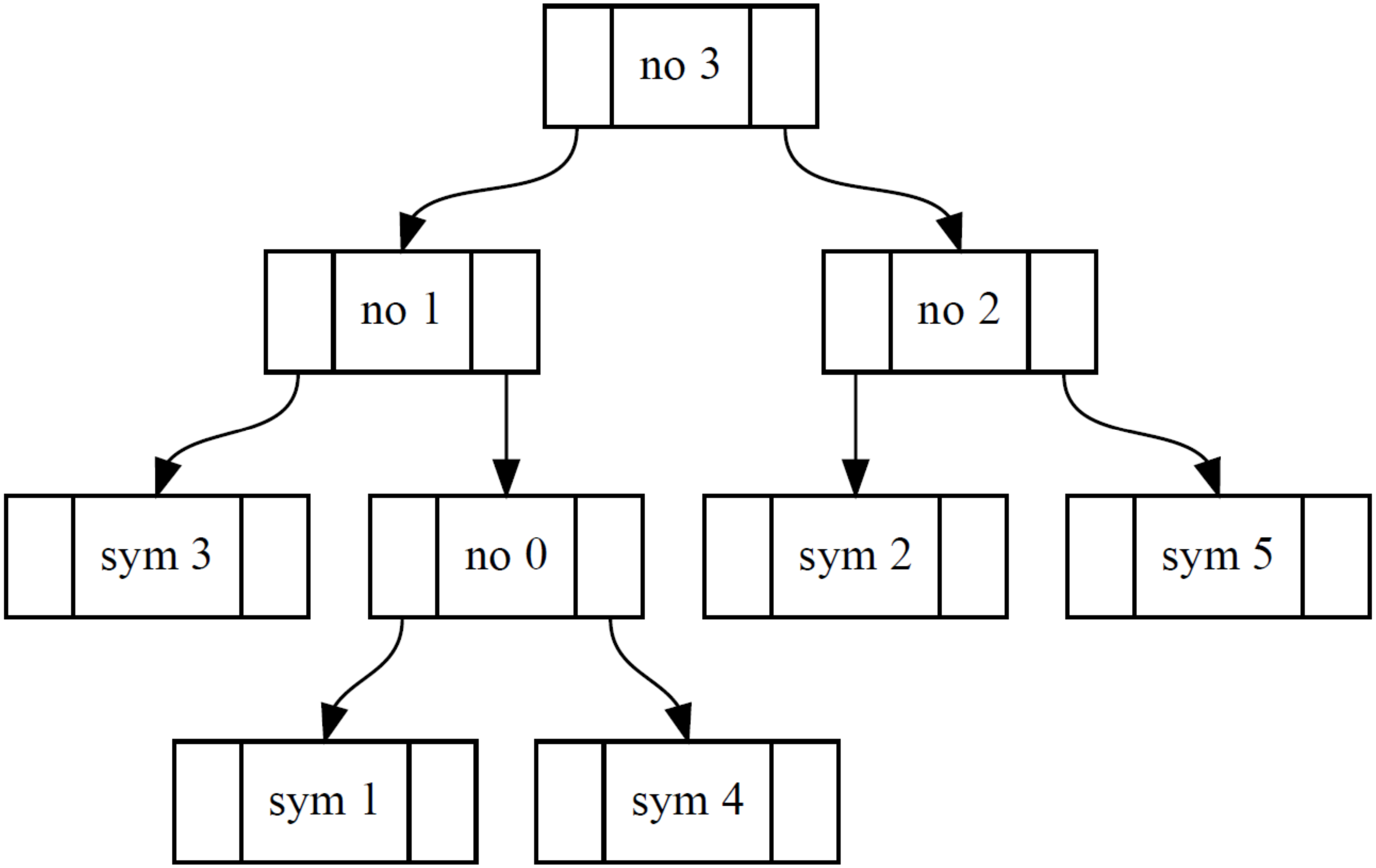
- generate_tree_general
  - In here you must reconstruct a tree no matter what order it was written to the file
  - It could be in postorder, or preorder, or inorder, or whatever ... you have to get the tree back
  - Your assignment always writes trees in postorder, so you'll have to test with your own functions that write trees in other orders
  - You're going to have to use the node numbers for sure!

Recovering Tree in Postorder

- generate_tree_postorder
  - In this function, you are not allowed to use the node-numbers at all
  - But, you are allowed to assume that the tree is written in postorder
  - Tip: you know which left/right subtrees are leafs and which are not.

    This is crucial information!

Uncompressing Using a Tree



- Uncompress 110110000

Next up ..

- BST mutation
  - Insert operation
  - Delete operation

Recall BinaryTree node implementation

```
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
```

insert operation

- Insert must ensure BST condition holds:
  - Left subtree of N => smaller values than N’s data
  - Right subtree of N => larger values than N’s data

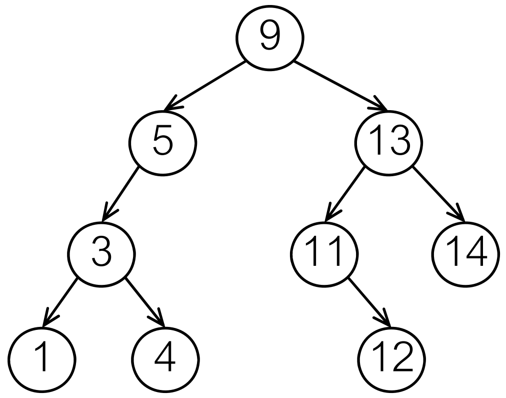
- How do we insert a new node then?

insert operation

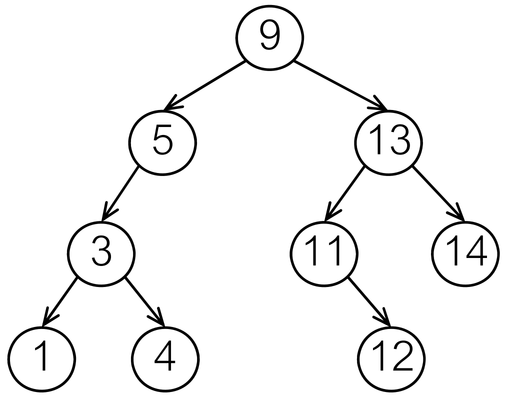
- Where do we insert a new value, while keeping it a BST?

  Thoughts?

insert value 7?



insert value 10?



insert operation

```
def bst_insert(node, data):
    """ Insert data in BST rooted at 'node' if necessary, and return new root.
    @param BinaryTree|None node: node of a Binary Search Tree
    @param object data: data to insert into BST, if necessary.
    >>> b = BinaryTree(8)
    >>> b = insert(b, 4)
    >>> print(b)
    8
        4
    """
    pass
```

Idea: check the node's data against value

if smaller, value should go somewhere in its left child subtree

if larger, value should go somewhere in its right child subtree

What if no left/right child?

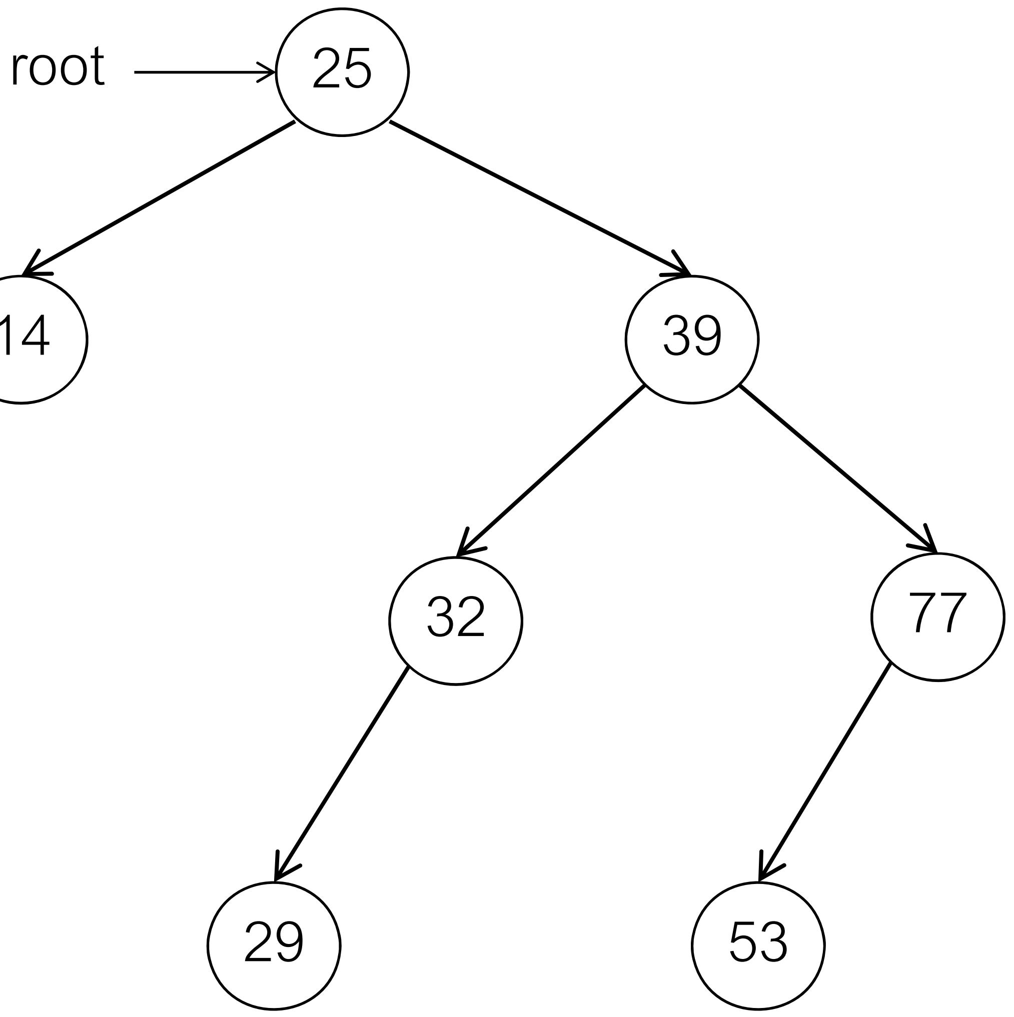
What if the value is already in the tree?

delete operation

- Input:
  - tree rooted at a node called root
  - a value V that we wish to delete from the tree (if it exists!)

- Output:
  - if V found => return corresponding node from the tree.
    - This node gets “disconnected” / ”extracted” / “removed” from the tree
  - if V not found => return None

delete operation

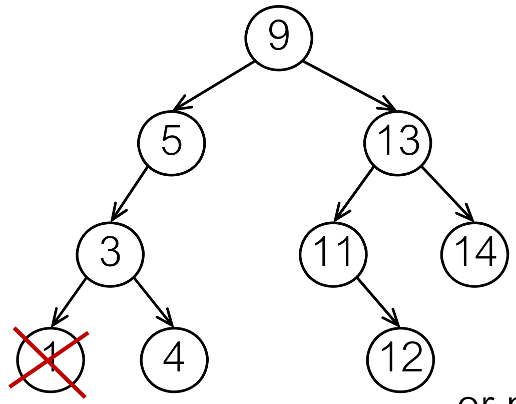


Deletion of data from BST rooted at ‘node’?

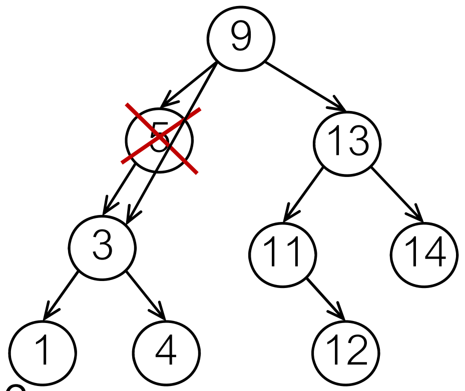
- Locate the node to delete, by traversing the tree
- Let *current* point to the current node being inspected in the tree and *parent* point to the parent of the current node
  - What to do if current is None?
  - What if data to delete is less than the data within the current node?
  - What if data to delete is more than the data within the current node?
  - What if data to delete equals current’s data?

Deletion of data from BST rooted at 'node'?

- Data to delete equals current's data => time to remove the node
- Three cases – first two are pretty straightforward though:
  - 1. Current has no children
    - e.g., Current points to node 1

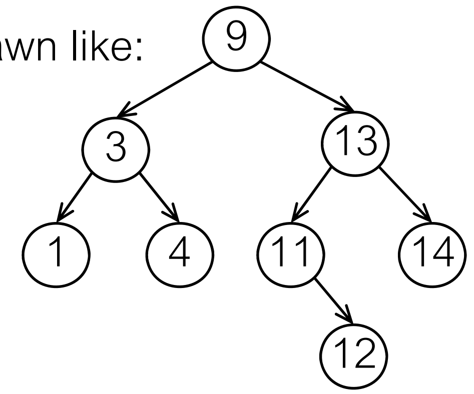


  - 2. Current has only one child
    - e.g., Current points to node 5



or more nicely

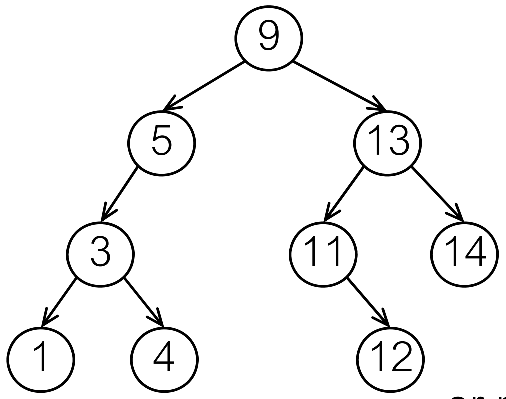
drawn like:



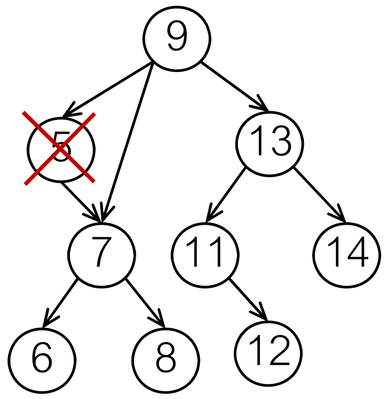
    - Does it matter which child?

Deletion of data from BST rooted at 'node'?

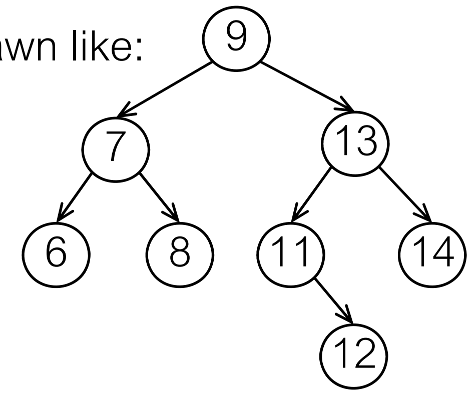
- Data to delete equals current's data => time to remove the node
- Three cases – first two are pretty straightforward though:
  - 1. Current has no children
    - e.g., Current points to node 1



  - 2. Current has only one child
    - e.g., Current points to node 5



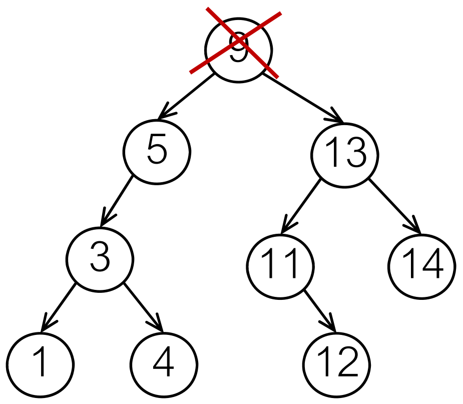
or more nicely drawn like:

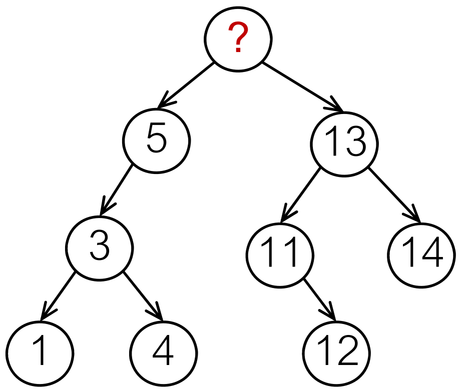


    - Does it matter which child?

Deletion of data from BST rooted at 'node'?

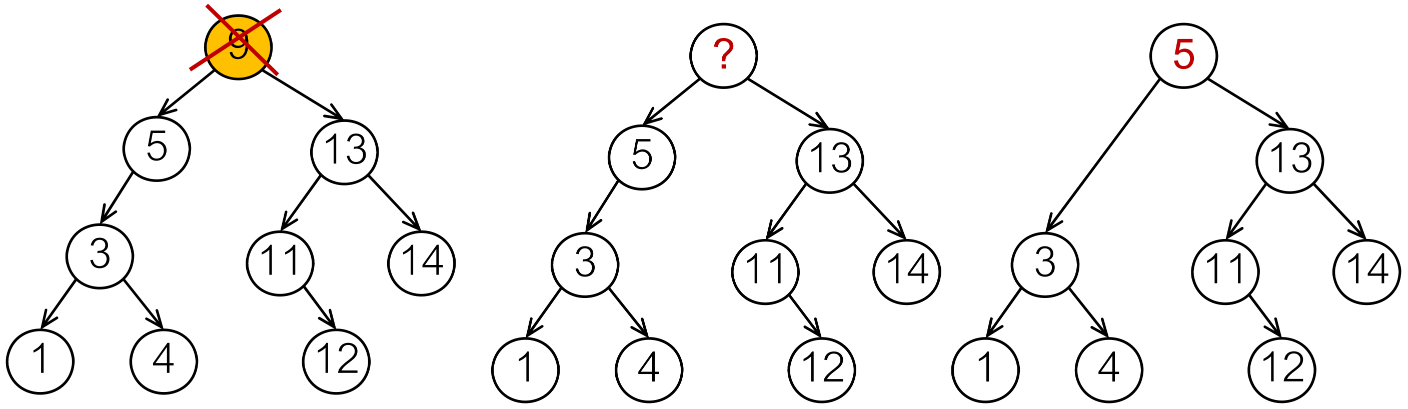
- 3. Current has both children => a bit more complex
  - BST properties must still hold after the deletion! For each node X:
    - P1) every node in the left subtree must have data smaller than X's data
    - P2) every node in the right subtree must have data larger than X's data
  - Let's say we want to remove node with value 9
    - Idea: Keep the node, clear value 9, pick another value from under this node (as a replacement), and remove THAT node.
    - How do we do this, while still keeping the BST properties? Thoughts?





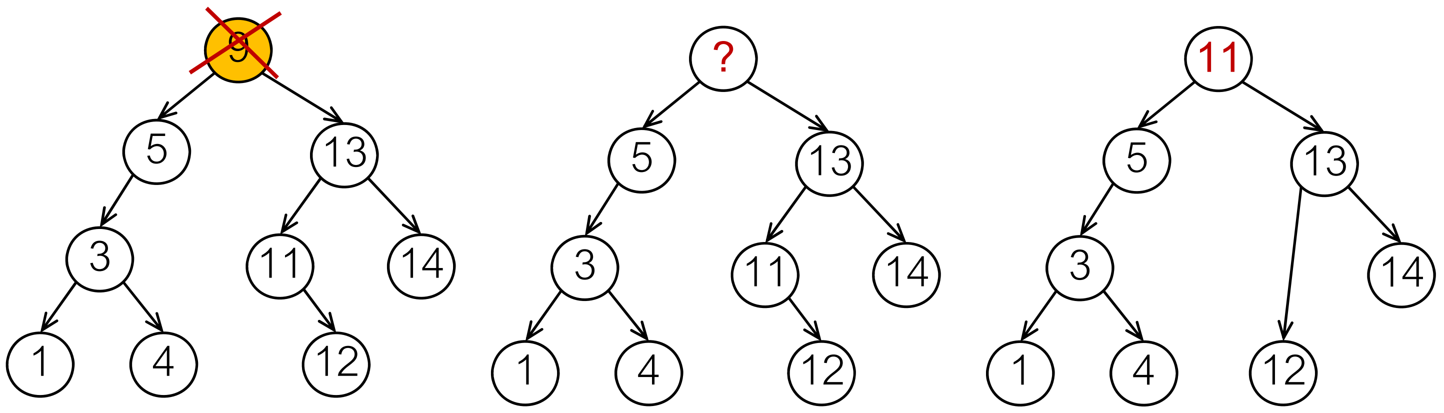
Deletion of data from BST rooted at 'node'?

- 3. Current has both children => a bit more complex
  - BST properties must still hold after the deletion! For each node X:
    - P1) every node in the left subtree must have data smaller than X's data
    - P2) every node in the right subtree must have data larger than X's data
  - Let's say we want to remove node with value 9
    - Idea: Keep the node, clear value 9, pick another value from under this node (as a replacement), and remove THAT node.
    - What if we picked a value from the left subtree?



Deletion of data from BST rooted at 'node'?

- 3. Current has both children => a bit more complex
  - BST properties must still hold after the deletion! For each node X:
    - P1) every node in the left subtree must have data smaller than X's data
    - P2) every node in the right subtree must have data larger than X's data
  - Let's say we want to remove node with value 9
    - Idea: Keep the node, clear value 9, pick another value from under this node (as a replacement), and remove THAT node.
    - What if we picked a value from the right subtree?



delete operation steps - recap

- Traverse the tree to locate the node that contains the element
  - Use current and parent to keep track of where you are in the tree
- If current is None => not found, done!
- If data to delete is less than current's data => inspect left subtree
- If data to delete is more than current's data => inspect right subtree
- If data to delete equals current's data
  - Case a) No children => easy, just remove the node
  - Case b) One child => easy, just connect current's child to current's parent
  - Case c) Two children => clear current's value, pick a replacement value from a descendant under it, and delete that descendant node
    - Max from left subtree, or min from right subtree

delete operation steps - recursive

- delete_bst(node, value)
- Base case: If node is None => return None!
- If value < node.data => delete it from left child and return this node
- If value > node.data => delete it from right child and return this node
- If value == node.data
    - Case a) If node has less than two children and you know one is None, return the other one
    - Case b) If node has two children => replace data with that of its largest child in the left subtree and delete that child, and return this node

Algorithm for delete

- Implement this and trace it!
- Draw diagrams!

More mutation – reflection!

- Change a Tree so that it is a mirror image of itself
- This changes every internal node
- Order of changes is critical
- Practice on your own!
    - Experience only comes with practice...