Coventry University

Trees

Dr. Diana Hintea

Lecturer in Computer Science

diana.hintea@coventry.ac.uk

210CT Week 5

Learning Outcomes

1. Random Stuff
2. Trees
3. Binary (Search) Trees
4. Insertion, Search and Traversal
5. Node Removal

Random Stuff

LinkedIn

Final Year Project

Random Stuff

Coursework

1. Harmonic Series(block 6)
2. Pseudocode for linear search and duplicate search (block 3) AND their corresponding Big-O (block 4)
3. Searching for an element within a certain range in an array (both sorted and unsorted) (block 5)
4. Node Delete (block 8)

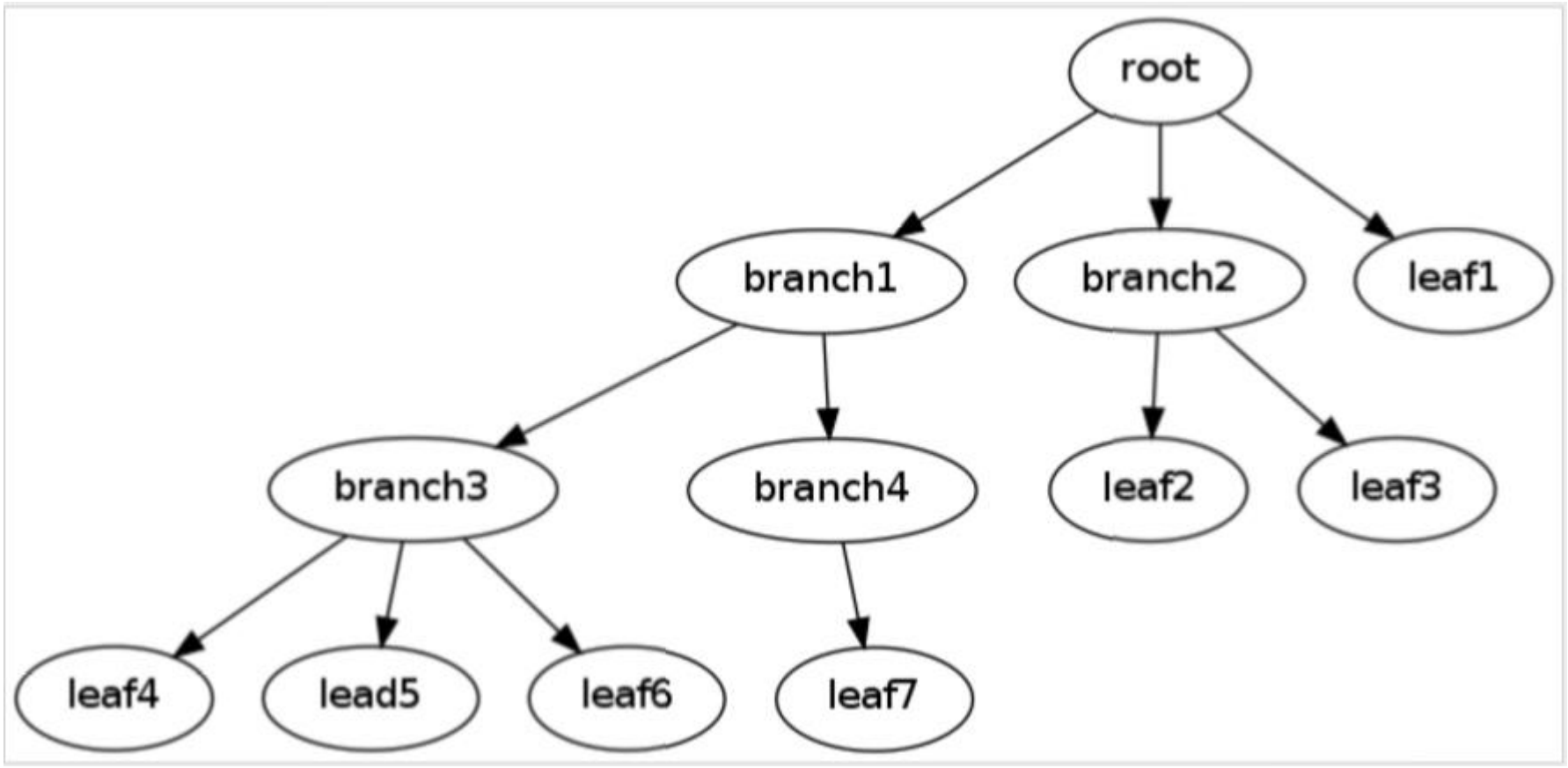
- Deadline: 30th of October 2015 11.55 pm

Trees

Introduction

- A **tree** is like a linked list, in that it has pointers to the 'next' node.
- The difference, however, is that there may be more than one 'next' pointer.
- In the case of a tree, we refer to the nodes pointed to by the 'next' pointers as either **branches** or **children**.
- The node above a child is referred to as the **parent** node.
- The topmost node is called the **root**.
- Nodes without any children are called **leaves**.

Trees


root
branch1
branch2
leaf1
branch3
branch4
leaf2
leaf3
leaf4
lead5
leaf6
leaf7

Trees

Uses of trees

- Some use of trees:

Trees

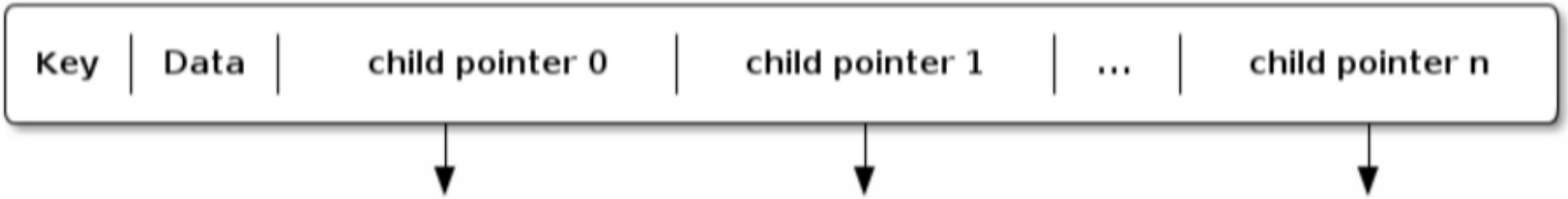
Uses of trees

- Some use of trees:
  - Speed up navigation of data
    - Think about the heap we used previously…
  - Hierarchical data
    - Space partitioning, for example.
  - Decision trees

Trees

Tree implementation

- A tree node might be imagined diagrammatically like this:



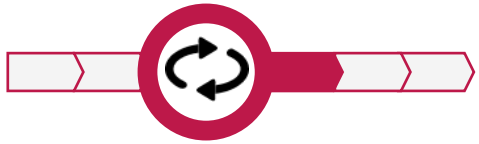
- With each child pointer pointing to another node object, or NULL.
- So far we have used simple data types that are their own key, but if the data is complex, we might need something to refer to it by.
- Imagine the value represents a whole employee record, for example.

Trees

Difference from a linked list

- The tree is somewhat like a linked list, in that it has nodes with data and pointers to other nodes.
- But the topology is different to a linked list, and has different uses.
- If a tree is well balanced, meaning that all the leaves are at about the same height, arbitrary nodes can be reached in logarithmic time.
- This makes it faster to hunt down a value in a tree than a linked list for many kinds of data.

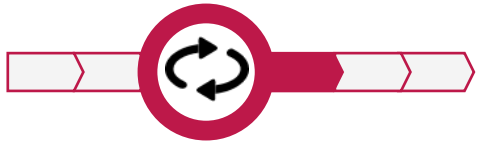
Binary (Search) Trees



What are they?

- While trees in general can have many children, a specific type of tree is examined here: **binary trees.**
- A binary tree is a tree where each node has either 0, 1 or 2 children.
- This has important implications for navigating a tree, as there are at most two choices at any step.

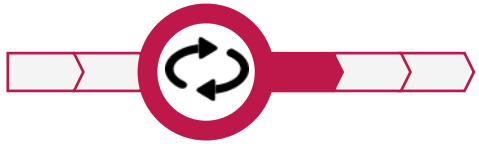
Binary (Search) Trees



A simple binary tree node

- A simple node for a tree with two children (or a binary tree) could have the following attributes:
  - n.value - the value stored at the node (assuming we are using simple values that are also keys).
  - n.parent - a pointer to the node's parent.
  - n.left - a pointer to the node's left child.
  - n.right - a pointer to the node's right child.
- The tree itself just needs a pointer to the root node.

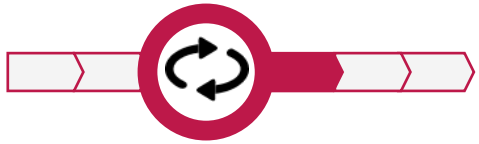
Binary (Search) Trees



Implementing the node (C++)

```
class BinTreeNode{
public:
   int data;
   BinTreeNode* left;
   BinTreeNode* right;
};
```

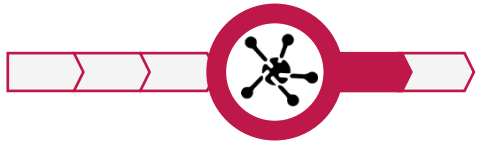
Binary (Search) Trees



Implementing the node (Python)

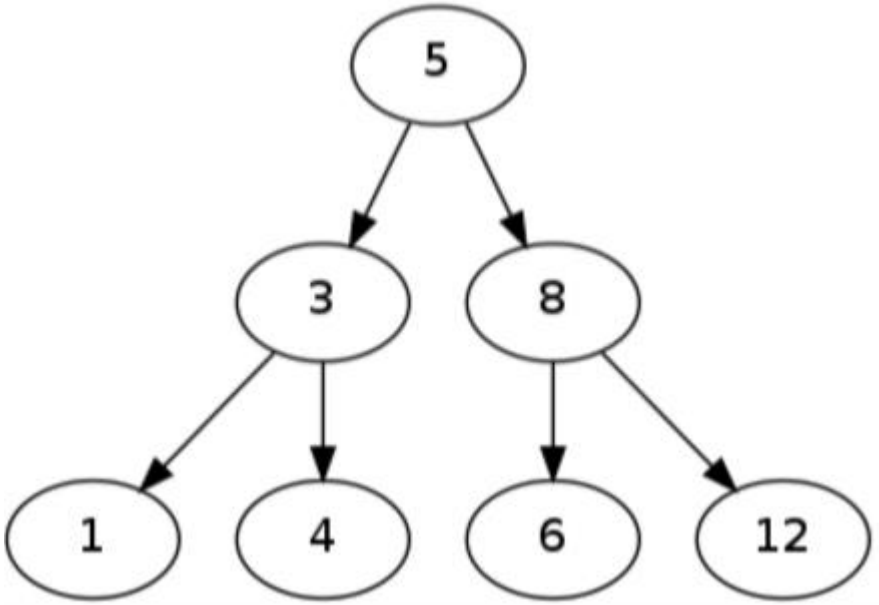
```
class BinTreeNode(object):
  def __init__(self, value):
    self.value=value
    self.left=None
    self.right=None
```

Insertion, Search and Traversal

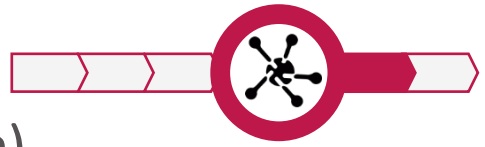


Binary Search Trees

- Used to quickly find data.
- The key in each node must be greater than all keys stored in the left sub-tree, and smaller than all keys in right sub-tree, as below:



Insertion, Search and Traversal



Inserting a Node in A Binary Search Tree

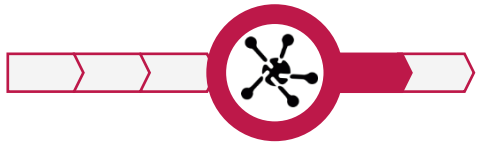
```
BIN-TREE-INSERT(tree,item)

IF tree = Ø
   tree=Node(item)
ELSE
   IF tree.value > item
      IF tree.left = 0
         tree.left=Node(item)
      ELSE
         BIN-TREE-INSERT(tree.left,item)
   ELSE
      IF tree.right = 0
         tree.right=Node(item)
      ELSE
         BIN-TREE-INSERT(tree.right,item)
return r
```

Insertion, Search and Traversal

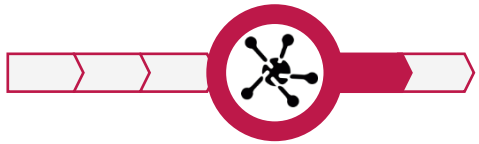
Finding an Item in a Binary Search Tree

```
BIN-TREE-FIND(tree, target)
r=tree
WHILE r ≠ Ø
    IF r.value = target
        RETURN r
    ELSE IF r.value > target
        r=r.left
    ELSE
        r=r.right
RETURN Ø
```



Efficiency

- Can you see the resemblance between finding an item in a binary search tree and using binary search?
- Each check halves (roughly) the number of items left to search.
- So, conversely, if we **double** the size of the tree, it takes how many more steps?



Efficiency

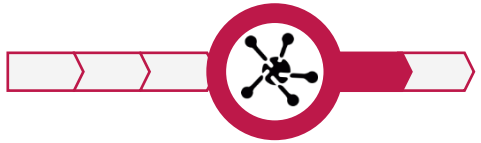
- Can you see the between finding in a binary tree and using a binary search?
- Each check halves (roughly) the number of items left to search.
- So, conversely, if we **double** the size of the tree, it takes how many more steps?
- **1!**
- So, the relationship is $log_2(N)$ and we would write it as $(logN)$.

Efficiency

- So how is this better than a list that we search with binary search?
- Insertion into an array, in sorted order is $O(N)$ (and the implementation is slow anyway due to array movements).
- And to use binary search we **need** a sorted sequence.
- Insertion into a list is still $O(N)$ because you have to find the right place.
- Inserting into a binary search tree is $O(logN)$, so much faster.

Binary Tree Traversal

- To traverse, or visit each node in a tree, there are a number of methods:
  1. **Pre order** - output item, then follow left tree, then right tree.
  2. **Post order** - follow the left child, follow the right child, output node value.
  3. **Breadth first** - start with the root and proceed in order of increasing depth/height (i.e root, second level items, third level items and so on).
  4. **In order** - for each node, display the left hand side, then the node itself, then the right. When displaying the left or right, follow the same instructions.

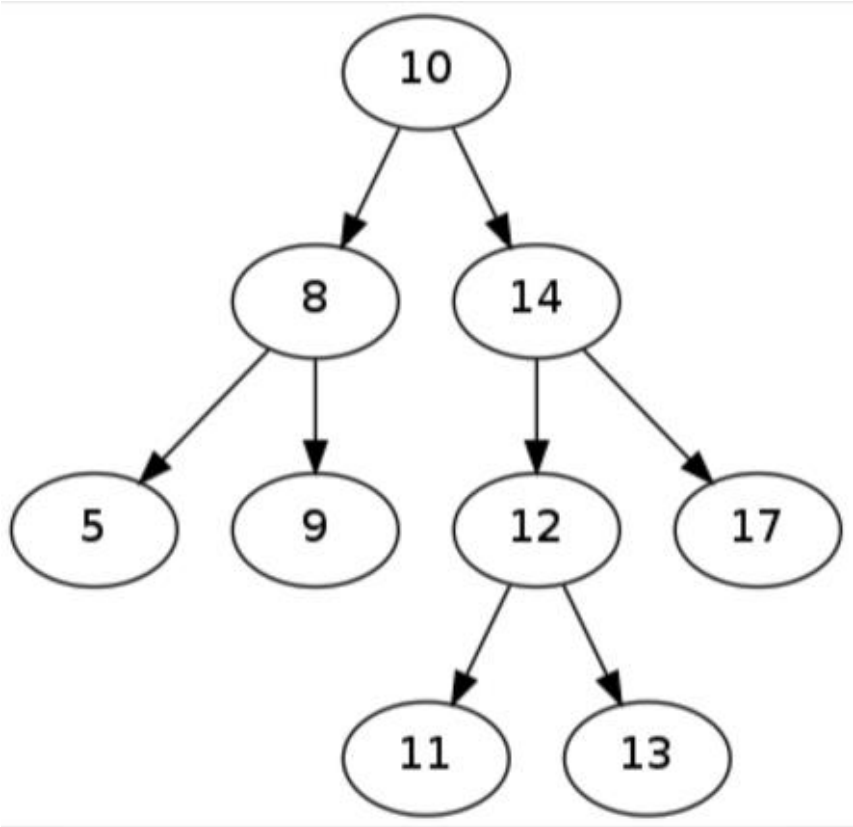


Binary Search Tree Traversal

Pre order - output item, then follow left tree, then right tree.

10, 8, 5, 9, 14, 12, 11, 13, 17

NLR



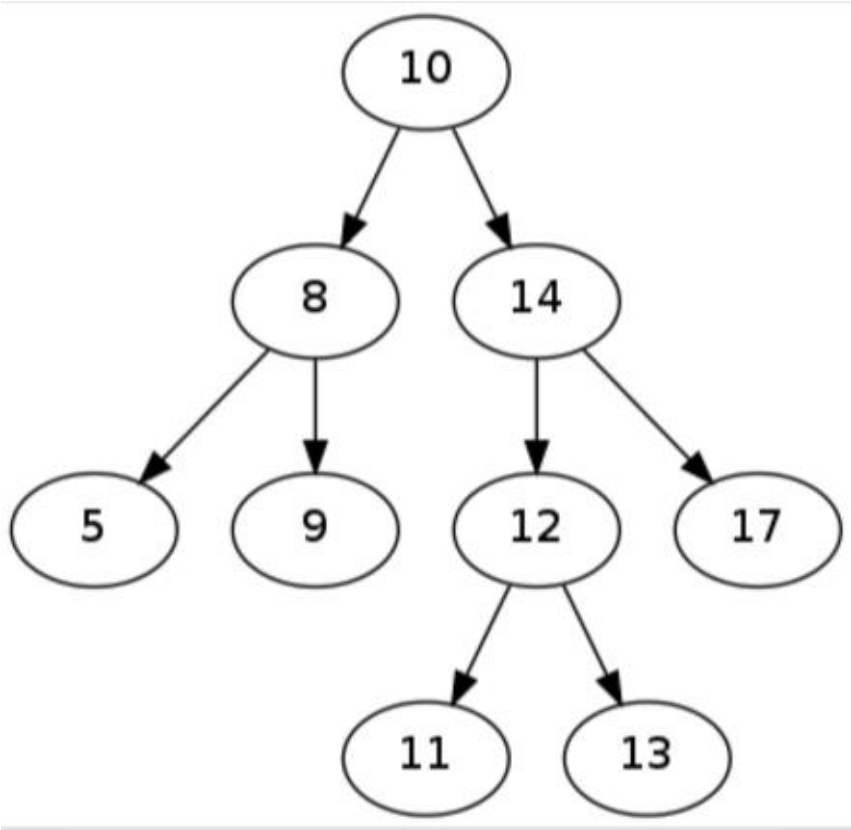
Insertion, Search and Traversal

Binary Search Tree Traversal

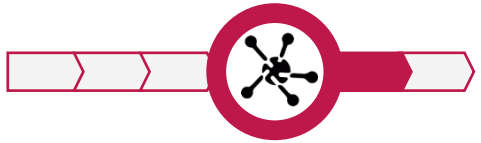
In order - for each node, display the left hand side, then the node itself, then the right. When displaying the left or right, follow the same instruction.

5, 8, 9, 10, 11, 12, 13, 14, 17

LNR



Insertion, Search and Traversal

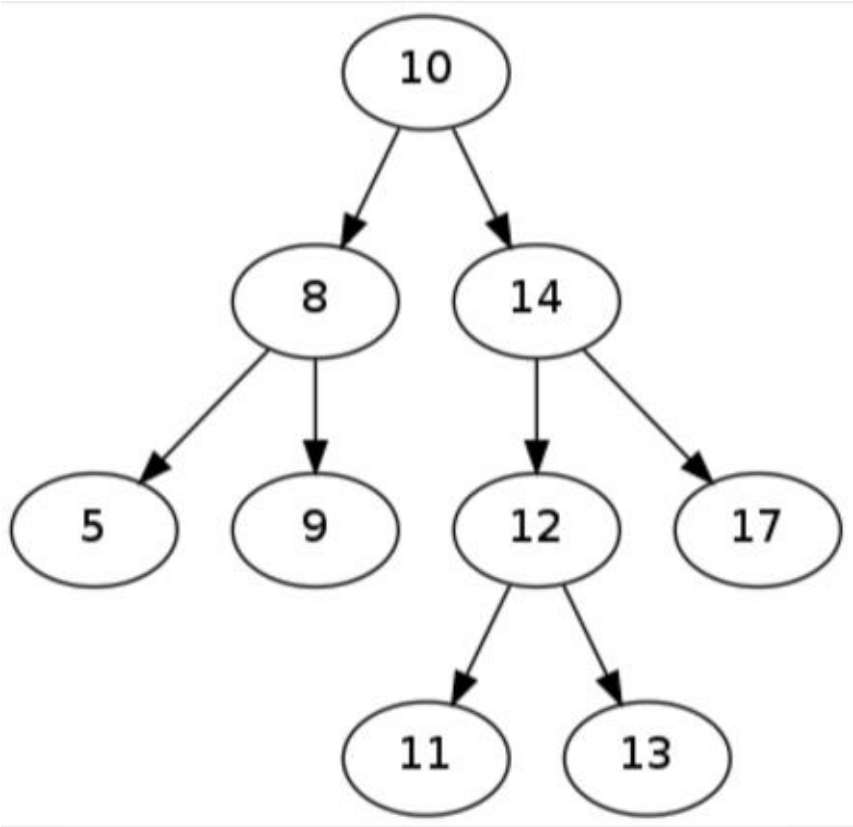


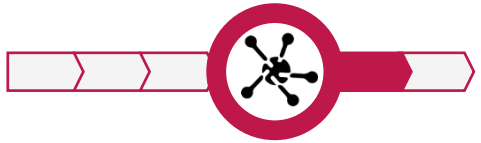
Binary Search Tree Traversal

Post order - follow the left child, follow the right child, output node value.

5, 9, 8, 11, 13, 12, 17, 14, 10

LRN

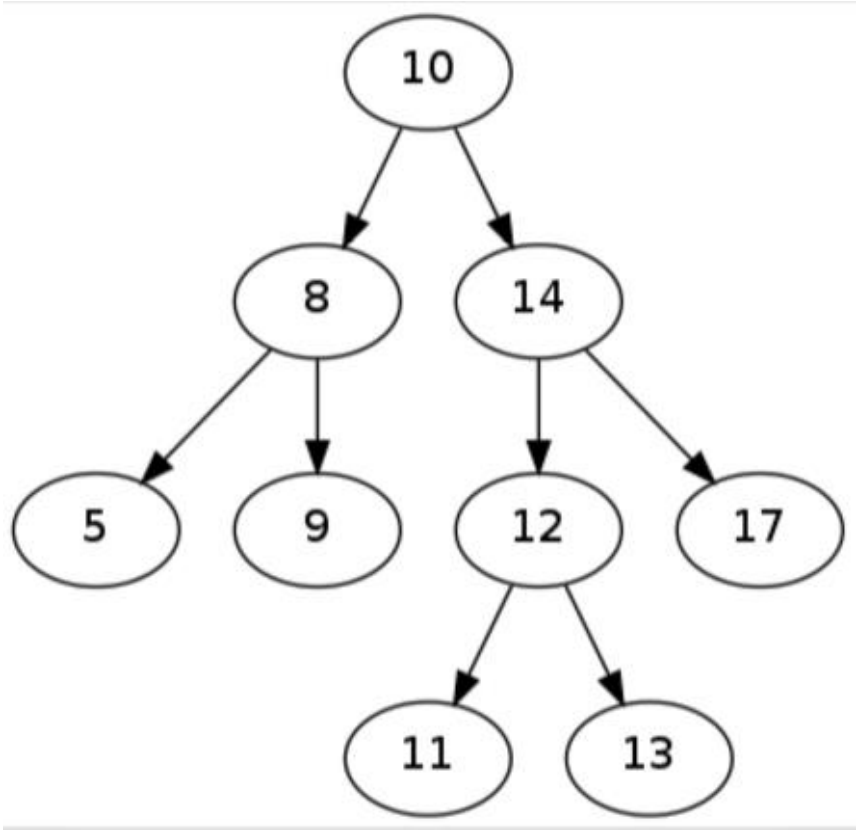




Binary Search Tree Traversal

Breadth first - start with the root and proceed in order of increasing depth/height (i.e root, second level items, third level items and so on).

10, 8, 14, 5, 9, 12, 17, 11, 13



Insertion, Search and Traversal

Binary Tree C++ Implementation

Binary Tree Python Implementation

Insertion, Search and Traversal

Pre-Homework

- Download and try the code written in class.

Actual Homework

- In your chosen language, implement the BIN-TREE-FIND function.
  - The pseudocode is given in the lecture notes.
  - Use the Python/C++ implementation given in the lecture as a starting point. Add your function to this.

Alternate Homework

- Write a BinTreeNode class that has methods for insertion and finding.
- So, rather than having a function that takes a tree as a parameter, make the functions part of the class definition.

Node removal

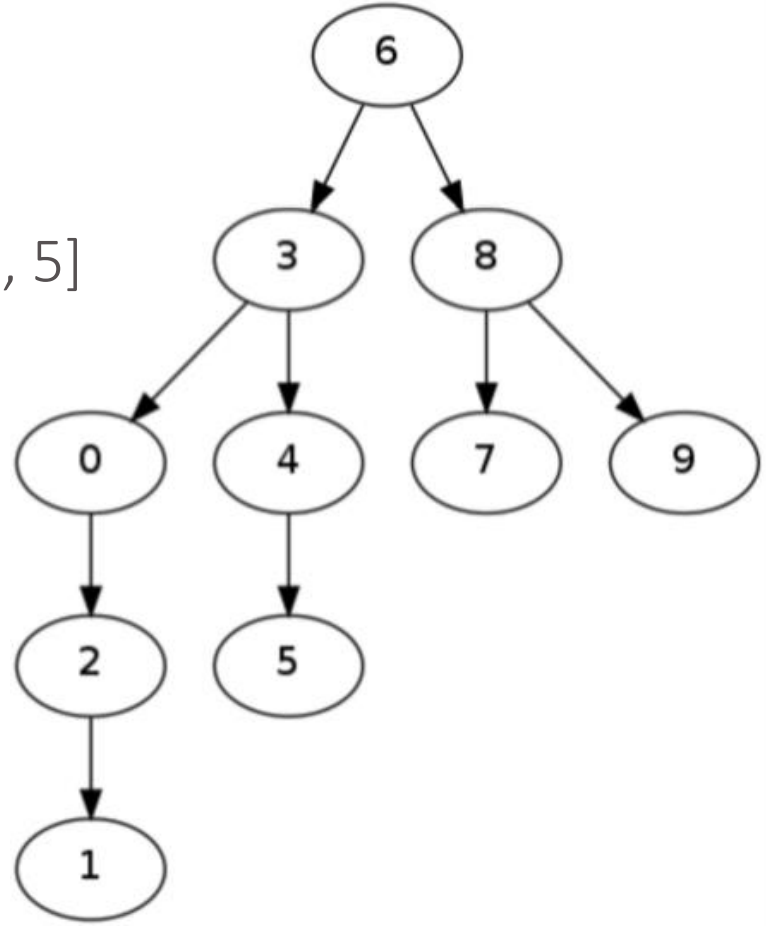
Insertion Order

- The order in which nodes are inserted can have an effect on the structure of the tree.
- If the nodes are inserted "in order" then the tree will become unbalanced.
- This removes the performance benefit of a tree as it now behaves as a single linked list.
- Completely ordered input results in a linked list instead of a tree!
- Binary tree use assumes unordered (random is best) insertion.

Node removal

Random

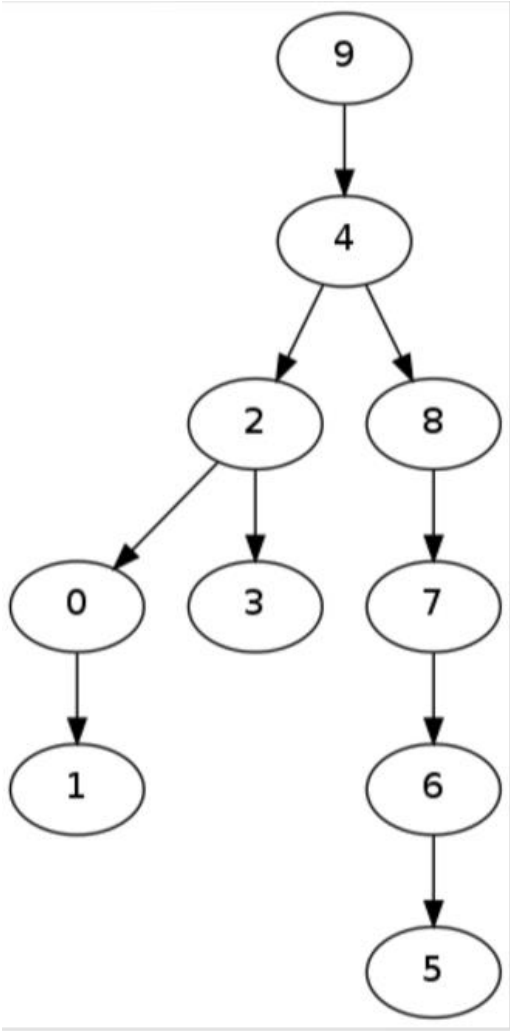
Inserting: [6, 3, 0, 2, 1, 8, 9, 4, 7, 5]



Node removal

Random

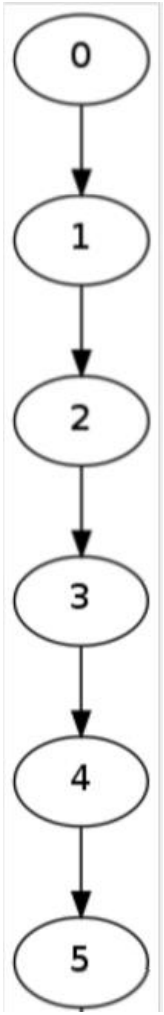
Inserting: [9, 4, 8, 7, 6, 2, 5, 3, 0, 1]



Node removal

Ordered

Inserting: [0, 1, 2, 3, 4, 5]



Node removal

Removing a Node

- More complex than search or insertion as the process depends on the number of child nodes.
- If the node to be deleted is a leaf i.e. no children, this is easy, just delete the node.
- If the node has only one child we set the child's parent variable to the node above the node to be deleted.
- If the node has two children this is more complex.

Node removal

Removing a Node

- Find either the minimum value node from the right subtree, or maximum value node from the left subtree.
- Replace the deleted node with that value.
- Remove the leaf node whose value has been copied.

Node removal

Node removal

Actual homework

- Based on this Python code or this C++ code as a starting point.
- Steps:
  1. Add a parent reference so we can look back up the tree.
  2. Write a delete function that works on a childless node.
  3. Extend delete function to work with nodes that have one child.
  4. Extend delete function to work with nodes that have two children.


Happy Halloween