CSI33 Data Structures

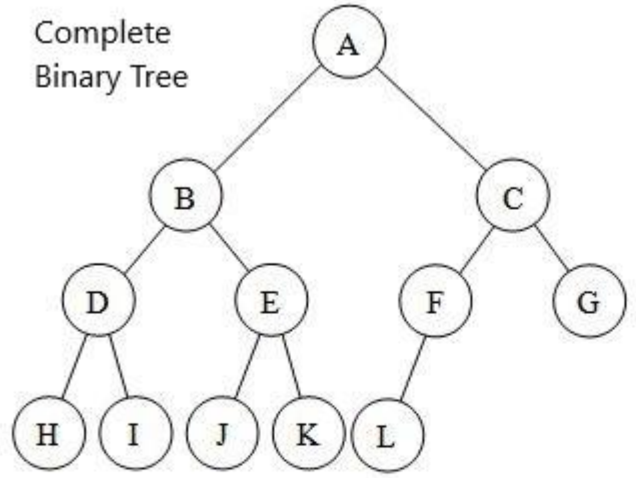
Sharon Persinger

Fall 2019

Day 15 October 28

Data Structures for Binary Trees

- The nodes of a binary tree can be stored in a list and retrieved by indexing.
- With a complete binary tree, nodes can be labelled in order top to bottom, left to right.
- The nodes can be stored in an array indexed by this number.
- If a node is a position i, its left child is at position 2*i +1 and its right child is at position 2*i + 2.
- The parent of a node at position i is at position (i-1)//2.
- We can generalize this for any binary tree by representing a missing node by None.



Linked Structure for Binary Tree

- See TreeNode.py for a class for a node for a binary tree.
- Construct a binary tree by linking up these nodes.

left = TreeNode(1)

light = TreeNode(3)

root = TreeNode(2, left, right)

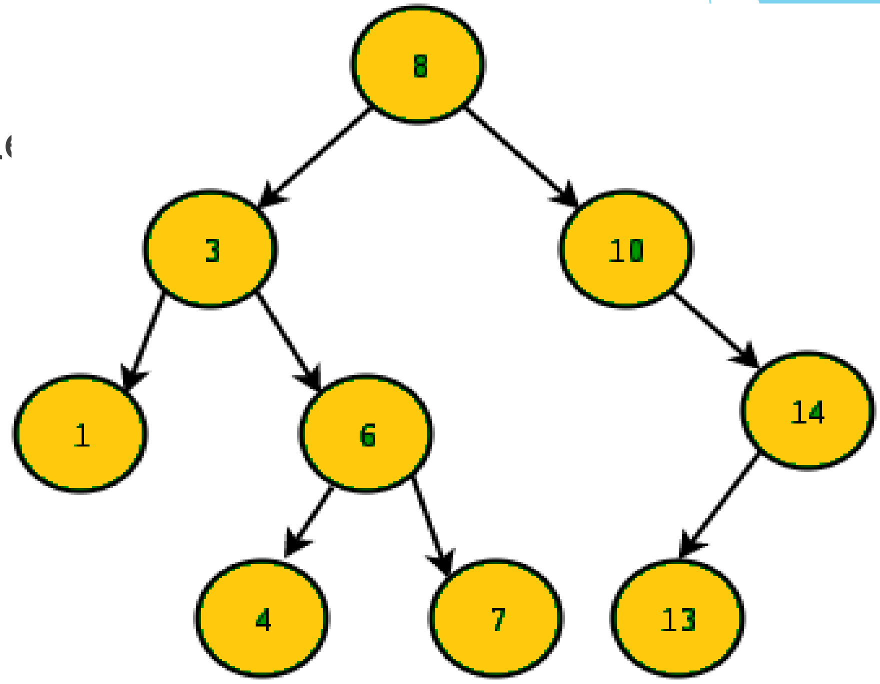
Give left a left child.

- We want to create a container class for the TreeNodes, as we created a LinkedList class.

Binary search tree – a container class for ordered sequences

- A binary search tree is a binary tree with this property:
- For every node, every element in the left subtree is less than the element at the node, and every element in the right subtree is greater than the element at the node.
- This structure makes for efficient searching, as long as we can arrange that the height of the tree isn't too large.
- Each item appears at most once.

- Exampl



BST implementation

- Use TreeNode Class.
- Constructor for empty BST.
- Write methods to insert, search, and remove items.
- BST.py

- insert – places a new item as a leaf
- Similar to search – look for the location where the item should be, using the order relation.
- Can do this iteratively or recursively.
- Look at both versions in BST.py

Recursive version of insert

```
def insert_rec(self, item):
    """insert item into binary search tree
    pre: item is not in self
    post: item has been added to self"""
    self.root = self._subtreeInsert(self.root, item)
```

- The recursive function _ _subtreeInsert moves down the tree until it finds the location to place the item as a new leaf.

```
def _subtreeInsert(self, root, item):

    if root is None:        # inserting into empty tree
        return TreeNode(item) # the item becomes the new
                                                    tree root
    if item == root.item:
        raise ValueError("Inserting duplicate item")

    if item < root.item:                # modify left subtree
        root.left = self._subtreeInsert(root.left, item)
    else:                                # modify right subtree
        root.right = self._subtreeInsert(root.right, item)

    return root # original root is root of modified tree
```

Search is similar

- Iterative version find
- Look for the item using the order
- Keep moving through the nodes following the order.
- Look at the driver for the while loop.

- Return the item since the item might be complex. We could be searching on one part of it.

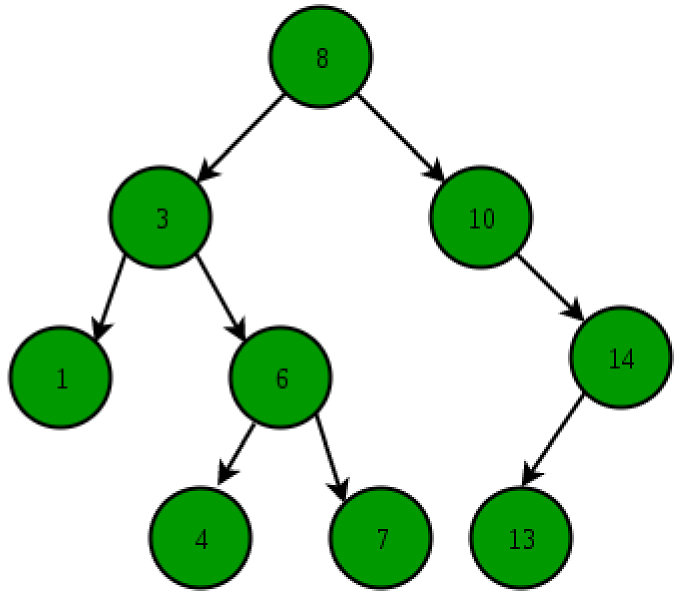
```
def find(self, item):
    """ Search for item in BST
    post: Returns item from BST if found, None otherwise"""

    node = self.root
    while node is not None and not(node.item == item):
        if item < node.item:
            node = node.left
        else:
            node = node.right

    if node is None:
        return None
    else:
        return node.item
```

Remove an item – How do we patch up the hole?

- Look at examples -
  - Remove 13 - Easy if the item is at a leaf.
  - Remove 14 - Easy if the item has only one child.
  - Remove 3 - ?
    - Find an item that can be used to patch the hole – replace the data in the node and preserve the BSP property. We want the node whose data comes immediately before the data being removed (or immediately after).
    - The predecessor is the largest value in the left subtree of the node. So the predecessor has no right subtree.



Look at delete

```
def delete(self, item):
        """remove item from binary search tree
        post: item is removed from the tree"""

        self.root = self._subtreeDelete(self.root, item)
```

```
def _subtreeDelete(self, root, item):
    if root is None:   # Empty tree, nothing to do
      return None
    if item < root.item:                    # modify left
        root.left = self._subtreeDelete(root.left, item)
    elif item > root.item:                  # modify right
        root.right = self._subtreeDelete(root.right, item)
    else:                                   # delete root
        if root.left is None:               # promote right subtree
            root =  root.right
        elif root.right is None:            # promote left subtree
            root = root.left
        else:
            # root node can't be deleted, overwrite it with max of
            #    left subtree and delete max node from the subtree
            root.item, root.left = self._subtreeDelMax(root.left)
    return root
```

Find the max element

```
    def _subtreeDelMax(self, root):

        if root.right is None:  # root is the
max
            return root.item, root.left  #
return max and promote left subtree
        else:
            # max is in right subtree,
recursively find and delete it
            maxVal, root.right =
self._subtreeDelMax(root.right)
            return maxVal, root
```

Traversing a tree

- In-order traversal to create list of items in the BST in the proper order.
- Function asList

- Function visit that performs an in-order visit of the BST, performs some processing with function f in-order at each node
- Important thing to notice – a function can be passed as a parameter to another function. Functions are objects in Python

Run time analysis

- Traversal – visit each of n nodes one time, so $\Theta(n)$ where there are n nodes.
- For search, insert, delete, only visit some of the nodes. How many?
- For each operation, take a path from the root to a leaf. How long is this path? In the worst case, this is the longest path. So in the worst case the number of steps is the height of the tree.

- The problem is that the height of the tree could be n – largest case - or it could be $log_2(n)$.
- We would like to have a balanced tree, where about half of the items in any subtree are in the left side and half are in the right side. We can guarantee this with some extra conditions that we will see later.