Outline

1. Chapter 7: Trees
   - An Application: A Binary Search Tree
   - In-class work

Approaching BST

Making a decision

We discussed the trade-offs between linked and array-based implementations of sequences (back in Section 4.7).

Linked lists are efficient for insertion and deletion operations, while a sorted array allows for efficient searching (recall binary search algorithm), although requires $\Theta(n)$ for insertion and deletion operations.

Let's combine the best of both worlds!

Approaching BST

Making a decision

We discussed the trade-offs between linked and array-based implementations of sequences (back in Section 4.7).

Linked lists are efficient for insertion and deletion operations, while a sorted array allows for efficient searching (recall binary search algorithm), although requires $\Theta(n)$ for insertion and deletion operations.

Let's combine the best of both worlds!

Approaching BST

Making a decision

We discussed the trade-offs between linked and array-based implementations of sequences (back in Section 4.7).

Linked lists are efficient for insertion and deletion operations, while a sorted array allows for efficient searching (recall binary search algorithm), although requires $\Theta(n)$ for insertion and deletion operations.

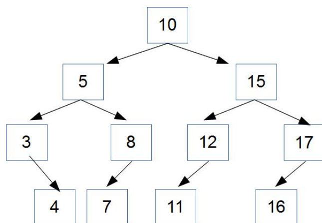
Let's combine the best of both worlds!

The Binary Search Property

A Binary Tree is "Sorted"

A Binary Search Tree, or BST, is a binary tree where every node has the following property:

- Each value in the left subtree is less than the value at the node.
- Each value in the right subtree is greater than the value at the node.



The Binary Search Property

Binary Search With A Binary Tree

- Start at the root
- If the value is there, we are done
- If the value is less than the node value, search the left subtree
- If the value is greater than the node value, search the right subtree

The Binary Search Property

Performance (Running Time) To Find A Value

- Average Performance is $\Theta(\log n)$.
  If the tree is not too unbalanced, then we divide the number of items to search in half at each node. This is actually a binary search.

- Worst-Case Performance is $\Theta(n)$.
  If the tree branches only to one side (left or right) this is the same as linear search.

Implementing a BST

__init__ (Constructor)

```
from TreeNode import TreeNode

class BST (object):

   def __init__(self):

      """ creates empty binary search tree """

      self.root = None
```

Implementing a BST

Trees are a naturally recursive data structure. Therefore, let's implement recursive insertion of an element into the BST.

```
def insert_rec(self, item):
    self.root = self._subtreeInsert(self.root, item)

def _subtreeInsert(self, root, item):  #recursive helper func.
    if root is None:
        return TreeNode(item)
    if item == root.item:
        raise ValueError("Inserting duplicate item")
    if item < root.item:
        root.left = self._subtreeInsert(root.left, item)
    else:
        root.right = self._subtreeInsert(root.right, item)
    return root #original root is root of modified tree
```

Implementing a BST

FIND

```
def find(self, item):

   """ post:  returns item from BST; 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
```

Implementing a BST

Removing nodes from the tree

Removing a specific item from a BST is a bit tricky. List of cases:

- the node to be removed is a leaf:
  then we can simply drop it off the tree
  (reference in its parent node is set to None)
- the node to be removed has a single child:
  then we can simply reset the reference from its parent to the reference to the node's child instead.
- the node to be removed has two children:
  leave the node in place, but replace its contents (i.e. value), i.e. find an easily deletable node whose contents can be transferred into the target node, while maintaining the tree's binary search property.
  (Two options there: rightmost node of the left subtree (our book), or leftmost node of the right subtree)

Implementing a BST

DELETE

```
def delete(self, item):
   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:  # overwrite root with max of left subtree
         root.item, root.left = self._subtreeDelMax(root.left)
   return root
```

Implementing a BST

_subtreeDelMax

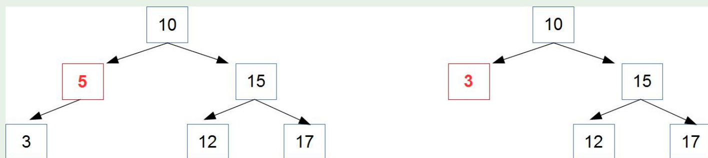
```
def _subtreeDelMax(self, root):

   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
```

Deletion from BST Examples

Example 1

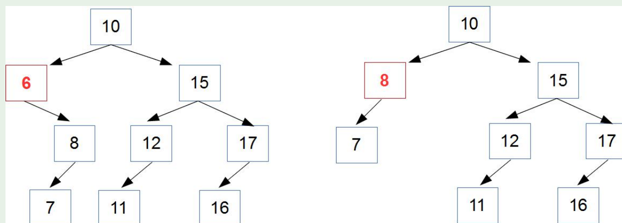
Let's delete 5 from a BST:



Deletion from BST Examples

Example 2

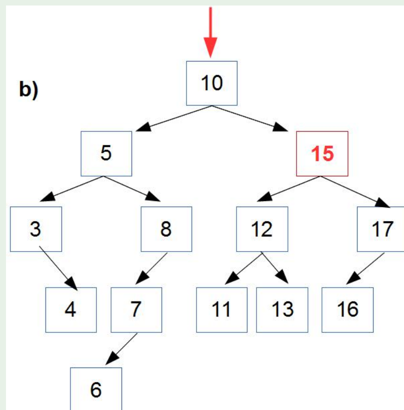
Let's delete 6 from a BST:



Deletion from BST Examples

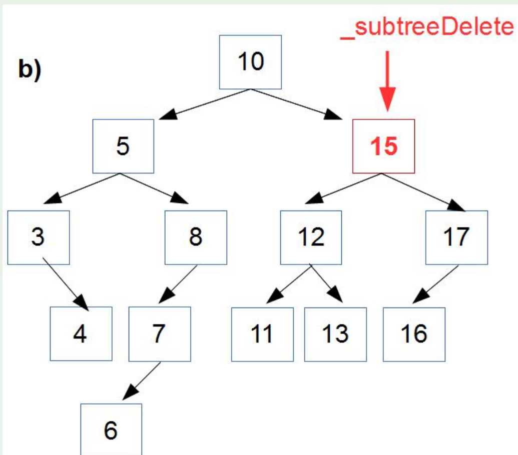
Example 3

Let's delete 15 from the given BST:



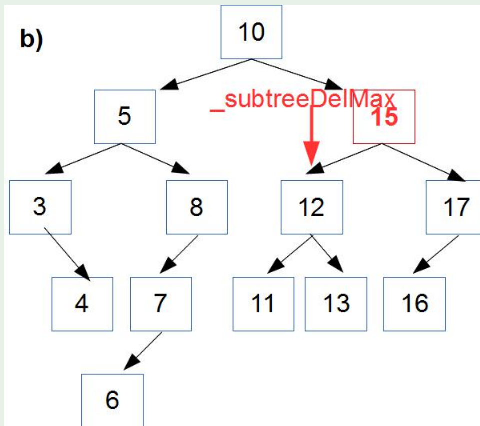
Deletion from BST Examples

Example 3



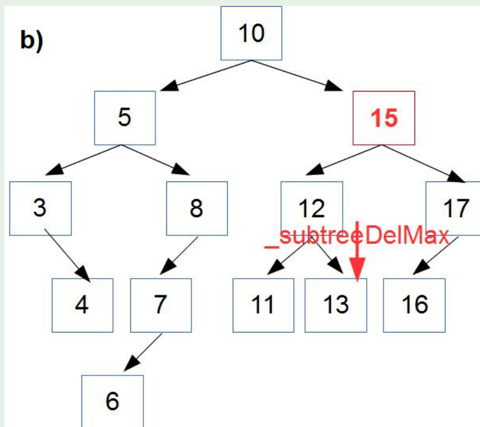
Deletion from BST Examples

Example 3



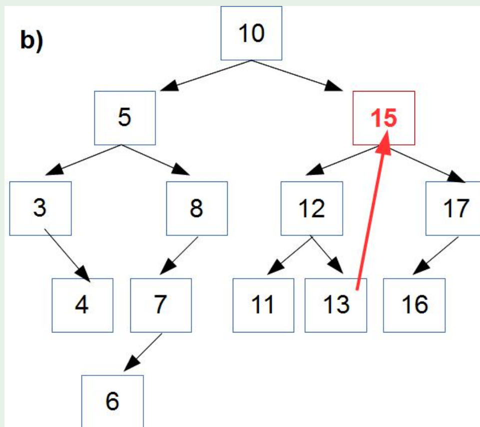
Deletion from BST Examples

Example 3



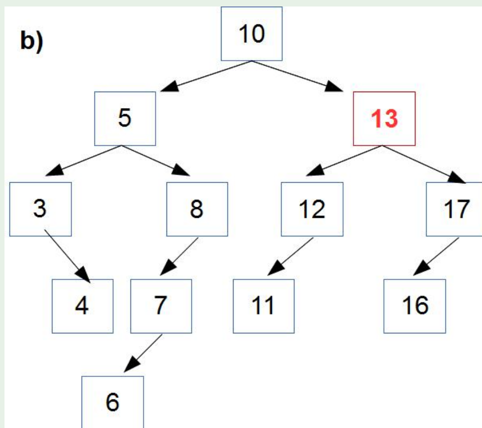
Deletion from BST Examples

Example 3



Deletion from BST Examples

Example 3



Deletion from BST Examples

Conclusion

As a conclusion, we can say that when deleting a value from BTS, which is represented by a node with two children, our book follows the following procedure:

step to the left sub-tree,
locate the rightmost node(or a leaf) in it,
and place its value in to the node with value to be deleted,
making necessary adjustments of the references.

Traversing a BST

Copy Data Into A List

As usual, we need to be able to iterate over the collection.

One approach is to assemble items from the tree into a sequential form, say a list or a queue.

- Use inorder traversal to keep items in order.
- Then process the list using Python's list methods.
- Disadvantage: uses extra memory for the list.

See the method `asList` in `BST.py`

Traversing a BST

Use Visitor Pattern

In case if we don't need all the elements from the tree, but just need to loop over the elements while doing something with them, then another design pattern, called the *visitor pattern* is useful.

- This is not the same as the iterator pattern—a separate class is not used.
- A visit method using inorder traversal can use a *function parameter*.
- The function is called for every node visited during traversal.

Traversing a BST

Visitor pattern

```
def visit(self, f):

    self._inorderVisit(self.root, f)


def _inorderVisit(self, root, f):

    if root is not None:
        self._inorderVisit(root.left, f)
        f(root.item)
        self._inorderVisit(root.right, f)
```

Traversing A BST

Use of a Visitor Pattern

Let's print all the elements of myBST:

```
def prnt(item):
   print(item)

myBST.visit(prnt)
```

Traversing a BST

Use Iterator Pattern

- Write a Python generator
- The yield statement will return the next node each time the generator is called.

Traversing a BST

Iterator

```
def __iter__(self):

   return self._inorderGen(self.root)

def _inorderGen(self, root):

   if root is not None:
      for item in self._inorderGen(root.left):
         yield item
      yield root.item
      for item in self._inorderGen(root.right):
         yield item
```

Run-Time Analysis Of BST Methods

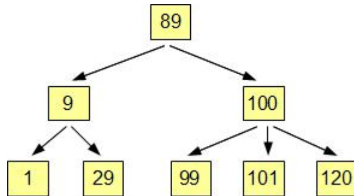
Methods

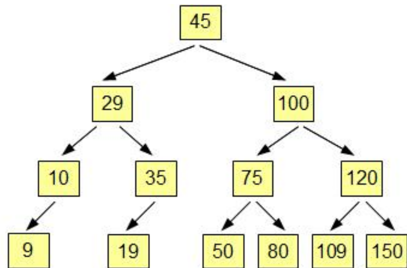
- visit is $\Theta(n)$.
- insert, delete, find have $\Theta(\log n)$ average behavior.
- insert, delete, find have $\Theta(n)$ worst-case behavior.

In-class work

In-class work - together

For the following, state weaver each of them is a binary tree, a binary search tree (BST), or just a tree.

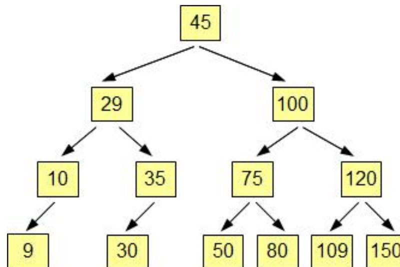




In-class work

In-class work - together

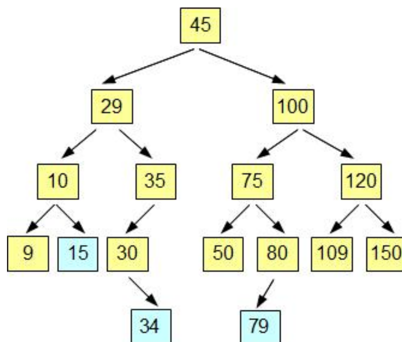
For the following BST, insert values 15, 34, and 79. Then delete values 30, 100, and 109.



In-class work

In-class work - together

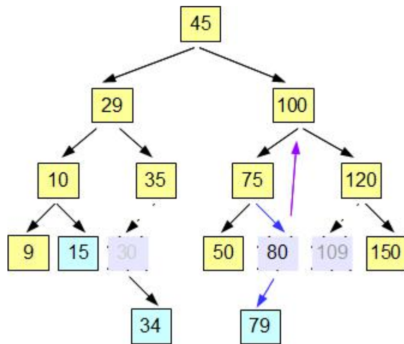
After insertion of 15, 34, and 79:

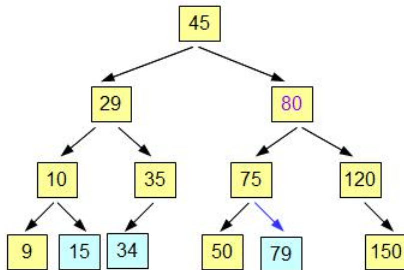


In-class work

In-class work - together

Deleting 30, 100, and 109:





In-class work

In-class work - on your own, part 1

Using class BST, insert the following numbers, one by one: 14, 10, 18, 25, 17, 7, 1, 12, 30.

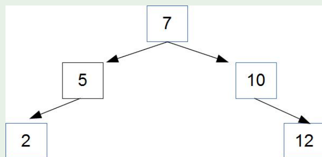
Draw this BST as you think it should look. Then use method `asList` to display the list as array.

In-class work

In-class work - on your own, part 2

Use TreeNode.py and BST.py to write the program that does the following:

1) Creates the following binary search tree:



2) then adds 6, 3, 11, and 14 to it,
3) prints the BST tree as an ordered list,
4) finds the maximum and minimum values, and
5) prints the product of all the numbers in the tree.