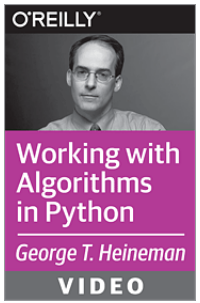

O'REILLY®

BINARYSEARCH
O (log n) Behavior



Problem Statement

- Does a collection contain a specific element
  - Assume collection is structured as a Python list
  - Python has **in** statement for this purpose
  - What is expected performance of **`in`**?

Does collection contain element

```
def contains(collection, target):
    return target in collection
```

Best/Average/Worst Case Analysis

- Best case is when 1st element is target
- Worst case you have to check each one
- Average case
  - If element is random one from list, then must check ½ of the elements; if not in list, must check each one
- Can you do better? Yes!
  - But only if you add structure the collection

Why Contains Functionality Is Important

- Fundamental functionality for a collection
- Precursor for insert request
  - In many situations you only want to insert a value if collection doesn't already contain it
  - Let's approach **in** with this concern in mind

```
if value not in collection:
 col.append(value)
```

BINARYARRAYSEARCH Algorithm

- A phone book with *n* entries is sorted by last name (and first name within last name)
  - Easy to locate a phone # for a given person
  - Hard to locate a person for a given phone #
- Observation on task difficulty
  - Searching through a phone book with 400 pages is not twice as hard as with a 200-page phone book

Algorithm Pseudocode

Assume A is sorted

To search through A, this algorithm cuts the problem size in half with each pass through the while loop

Divide problem into sub-problems half as large

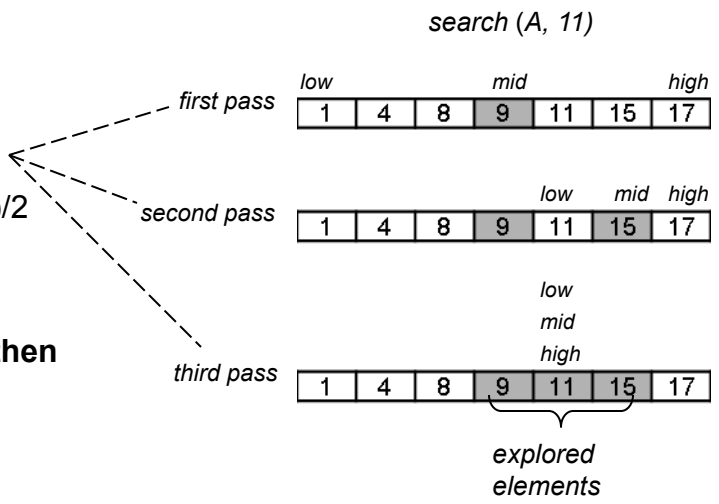
BINARYARRAYSEARCH

Best case	Average case	Worst case
O(*1*)	O(*log n*)	O(*log n*)

Array

Divide and Conquer

```
search (A, t)
  low = 0
  high = n–1
  while low ≤ high do
        mid = (low + high)/2
        if t = A[mid] then
              return true
        else if t < A[mid] then
              high = mid-1
        else low = mid+1
  return false
end
```



BINARYARRAYSEARCH Summary

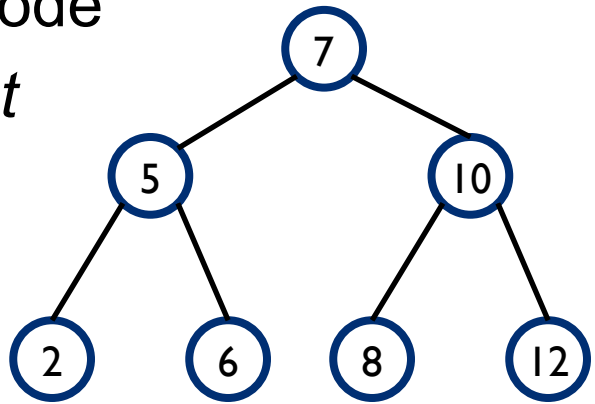
- Contiguous array storage inefficient
  - Block memory moves when inserting single values
- Consider using
  - You need to visit all elements in sorted order
  - Array doesn't change
- Evaluate each **in** use in your code
  - Easy performance gains

Binary Tree Data Structure

- Recursive data structure
  - Each **node** object has a value
  - A node may have a *left* or *right* child node
  - Topmost node in **tree** is called the *root*

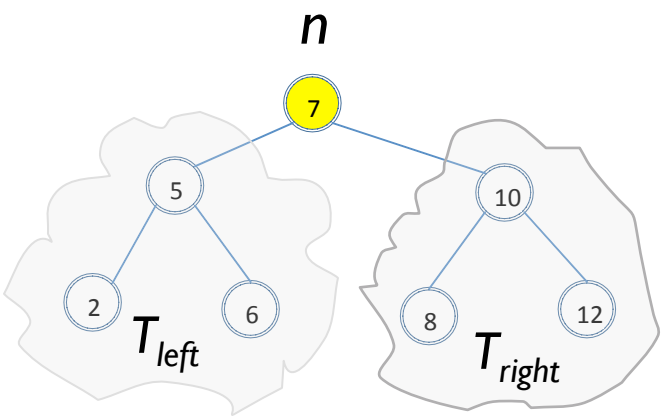
```
class BinaryNode {
    int             value
    BinaryNode      left
    BinaryNode      right
}
```

```
class BinaryTree {
    BinaryNode      root
}
```



Binary Search Tree Structure

- Binary Search Tree Property
  - All values in T_{left} for n are ≤ $n.value$
  - All values in T_{right} for n are ≥ $n.value$
- Observation
  - Each node is root of a BST
  - Same Hierarchical property as used by BINARYARRAYSEARCH

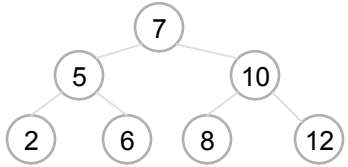


BST Issues

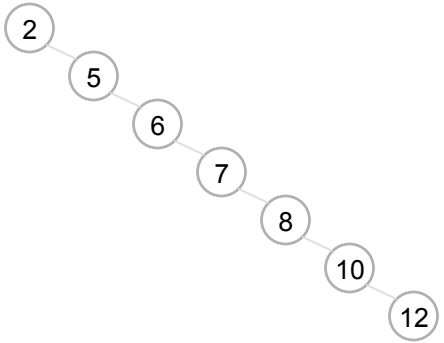
- Reasons to use Binary Search Tree
  - Input data size is unknown
  - Input data is highly dynamic, with significant number of insertions and deletions
- Problems that may arise
  - When a Binary Search Tree is constructed and modified, it may become *unbalanced*

BST Issues

- Use a BST implementation that self-balances
  - Otherwise…



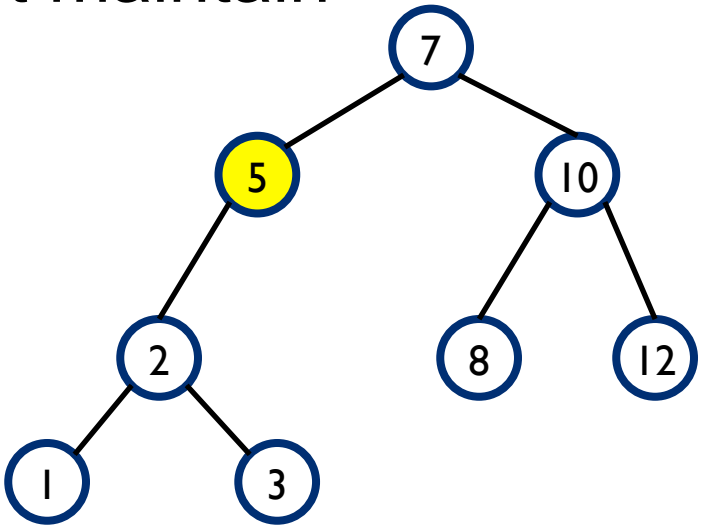
Tree is fully balanced for maximum efficiency



Tree looks more like a linked list, which leads to O(n) search performance

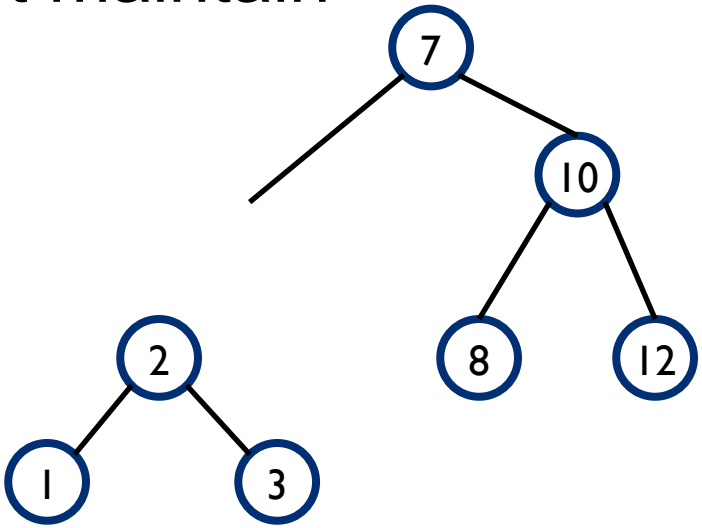
BST
Removing A Node

- To remove node from BST, must maintain Binary Search Tree Property
  - If leaf, just delete it
  - If has one child, replace self with sub-tree rooted at that child



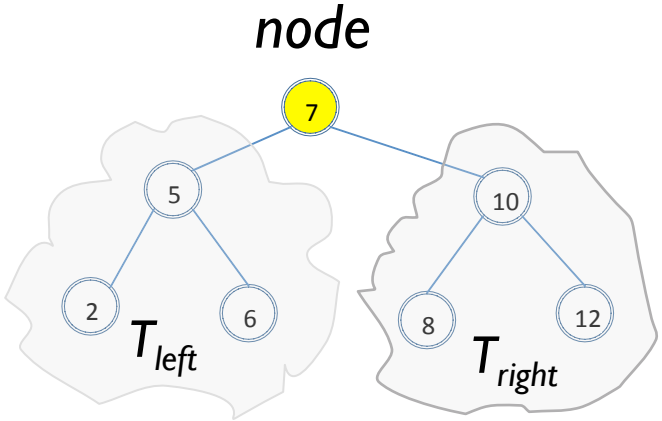
BST
Removing A Node

- To remove node from BST, must maintain Binary Search Tree Property
  - If leaf, just delete it
  - If has one child, replace self with sub-tree rooted at that child



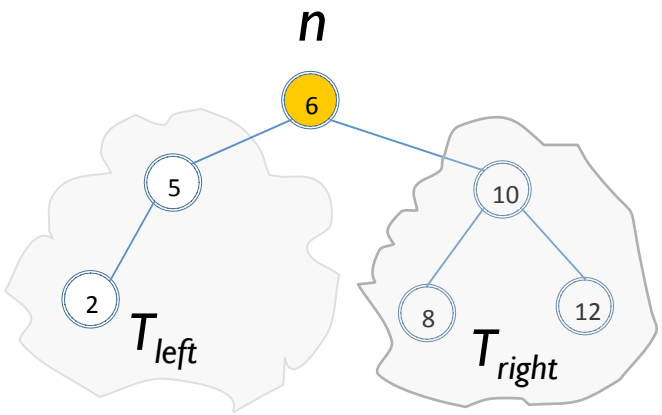
BST
Removing A Node

- If node has both children
  - Remove largest value M from T_{left}
  - Replace node's value with M (in this case 6)



BST
Removing A Node

- If node has both children
  - Replace with largest value in left sub-tree (in this case 6)
  - Code describing delete case is provided for your review



BST Balancing

- Should you choose to use Binary Search Trees
  - Choose a balanced tree implementation
- Several choices
  - Red/Black Trees
  - AVL Trees (discovered in 1962)
- (Re)balance Tree after insert/delete
  - Insertions and deletions may unbalance tree

BST Summary

- Efficient choice with dynamic behavior
  - Search, Insert, Delete all in O(log n) time
  - Iteration as traversal
- Balancing Required
  - Implementations exist (AVL and Red/Black trees)

BST Problem

- Create a balanced BST from a sorted collection?
  - Does the following image provide any suggestions?
  - Note how median is root?
  - And its two children are the medians of the left and right sub-arrays?