Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Trees II: Binary Search Trees

""

CS2013: Programming with Data Structures

BST Algorithms - find()

```
key: the data we are searching for
return: true or false depending on if key was
        found or not.

find(key):
  current = root
  while current != null:
    if key == current.data:
      return true
    else if key < current.data:
      current = current.left
    else if key > current.data:
      current = current.right
  return false
```

- What would be the runtime of this algorithm?

```
key:      the item we want to add

returns: the future parent Node of the item to add

assumptions: the tree is not empty, this case is dealt
             with in the insert() method.

insertionPoint(key):
  current = root
  parent = null

  while current != null:
    if key == current.data:
      throw DuplicateItemException
    else if key < current.data:
      parent = current
      current = current.left
    else if key > current.data:
      parent = current
      current = current.right

  return parent
```

```
key: the item to be inserted

returns: nothing

insert(key):
  Node child = new Node(key)

  if tree is empty:
    root = child
  else:
    try:
      parent = insertionPoint(key)
      if key < parent.data:
        parent.left = child
      else if key > parent.data:
        parent.right = child
    catch DuplicateItemException ex:
      throw ex
```

BST Algorithm - delete()

- Removing a node from a BST is a little more complicated, especially if the node we want to delete is not a leaf node.

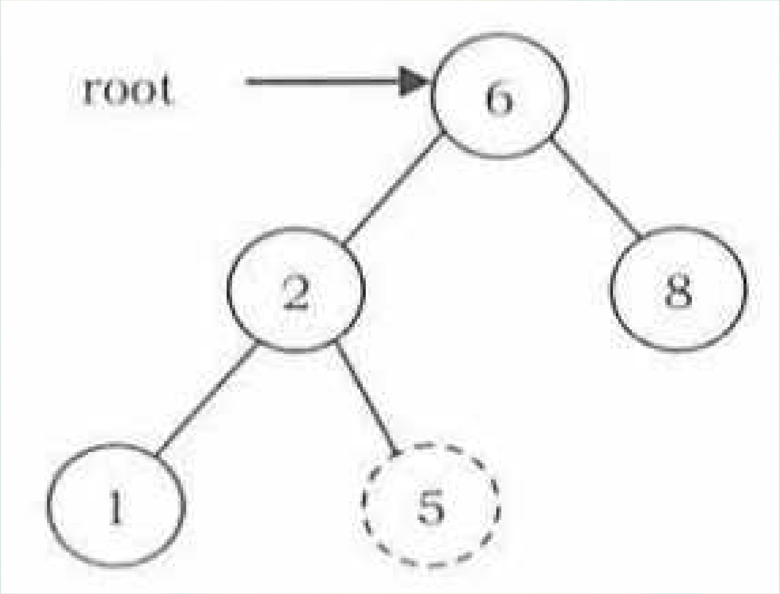
- First find the node you want to delete:
  - **`nodeToDelete()`**: We will just reimplement the find() method to return the actual node, instead of the data.

```
key: the data to delete

returns: The node to delete or
         null if the node was
         not found.

nodeToDelete(key):
  current = root
  while current != null:
    if key == current.data:
      return current
    else if key < current.data:
      current = current.left
    else if key > current.data:
      current = current.right

  return null
```

BST Algorithm - delete()

- **`delete()`** has three main cases to consider:

- **Case 1: The node we want to remove is a leaf node**
  - just delete it by setting its parent's left or right pointer to be null.
  - How can we tell if the node we want to delete is the left child or right child of its parent?



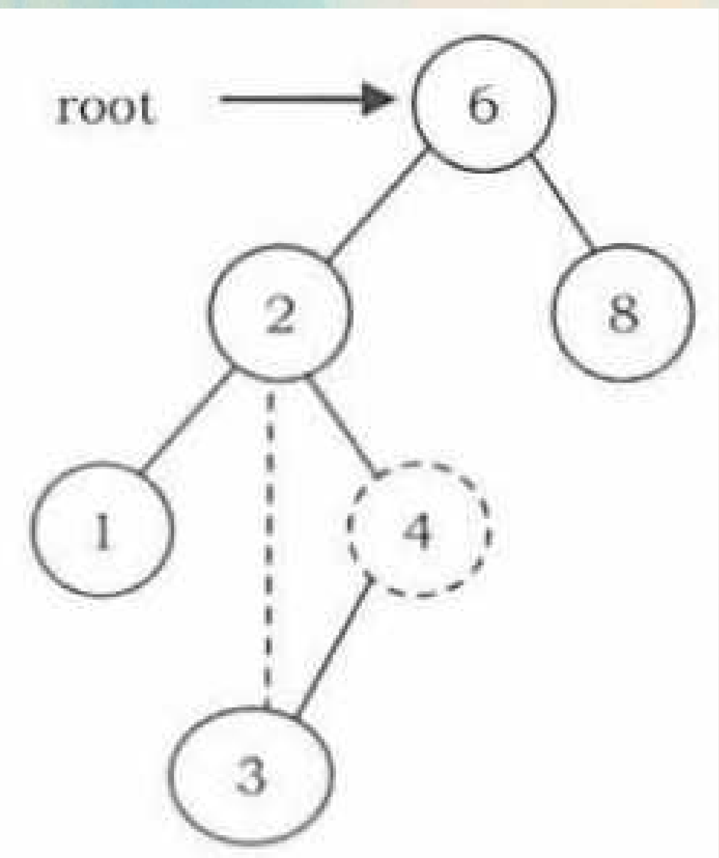
Left or Right Child?

```
isLeftChild(node):
  return node.parent.left.data ==  node.data
```

```
isRightChild(node):
  return node.parent.right.data ==  node.data
```

BST Algorithm - delete()

- **Case 2: The item you want to delete has one child.**
  - the node to delete's (*d*) child (*c*) must be connected to the parent of *d.*
    - Another way to say this is that (*c*) must be connected to its grandparent, bypassing *d*.
  - How can we tell how many children our node has?
  - How do we connect the child?

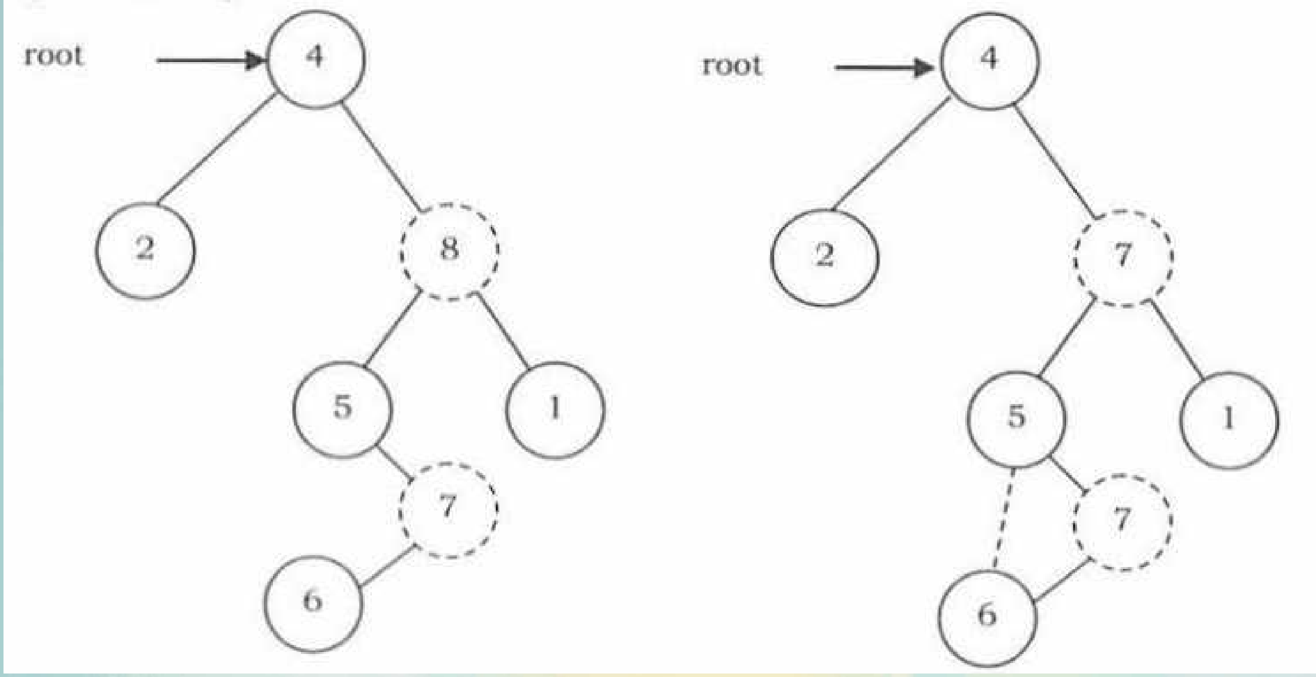


numChildren()

```
numChildren(node):
  count = 0

  if node.left != null:
    count++

  if node.right != null:
    count++

  return count
```

BST Algorithm - delete()

- **Case 3: The element to delete has both children:**
  - Assume you can find the node to delete (*d*).
  - Assume you can find the max element (*m*) in the left subtree of (*d*).
  - Copy the data from *m* to *d* and recursively delete() *m* from the tree.
  - How do we find the max element in the left subtree?



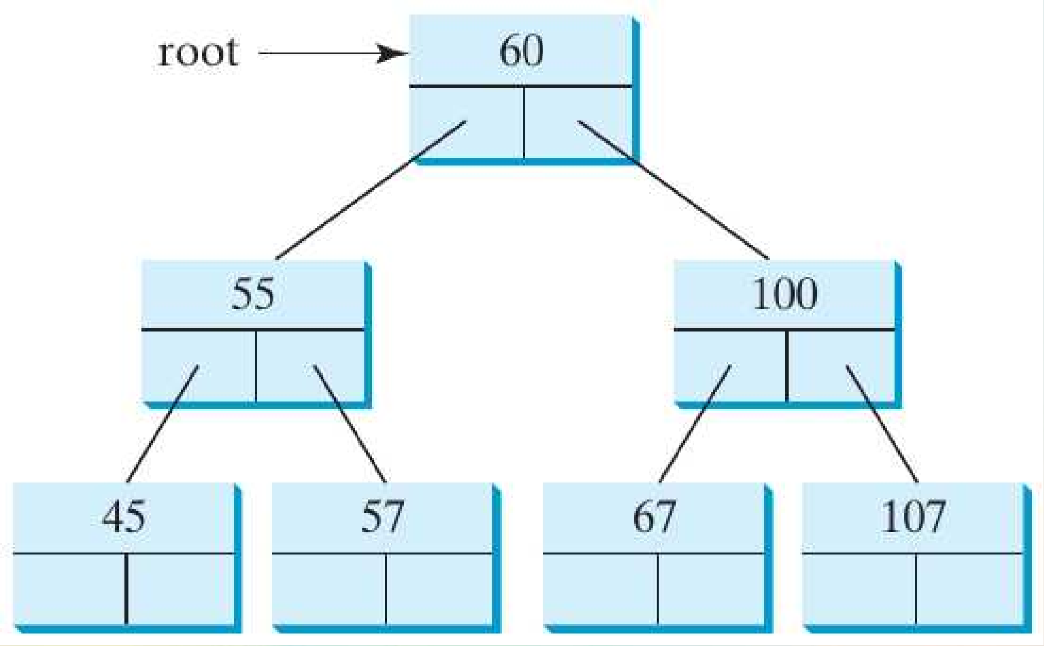
```
delete(key):
   delete(nodeToDelete(key))

delete(node):
   if node is leaf:
     if isLeftChild(node):
       node.parent.left = null
     else if isRightChild(node):
       node.parent.right = null
   else if numChildren(node) == 1:
     child = get the left or right child of node
     if isLeftChild(node):
        node.parent.left  = child
     else if isRightChild(node):
        node.parent.right = child
   else if numChildren(node) == 2:
     max = maxLeftSubtree(node)
     node.setItem(max.getItem())
     delete(max)
```

Tree Traversal Algorithms

Preorder Traversal - Binary Tree

- ***preorder traversal***: of a binary tree, means that you visit the root of a subtree first before visiting the left and right children
  - also known as Depth-First Search

```
preorder(node):
   if node is null: return
   visit node
   preorder(node.left)
   preorder(node.right)
```

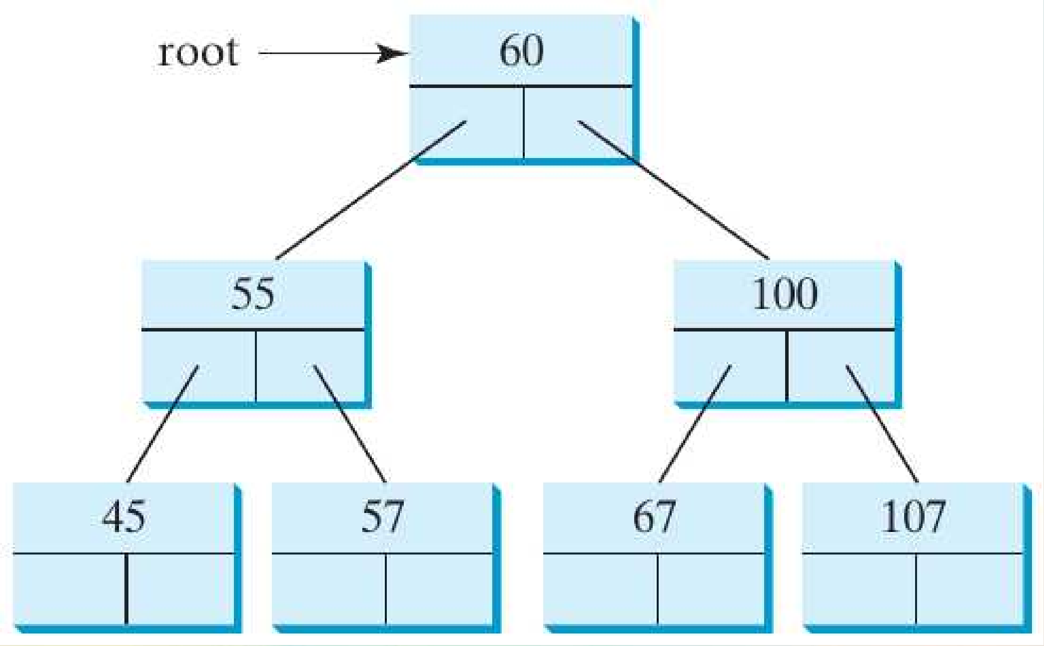


- In the example to the right the preorder traversal would be:

  60, 55, 45, 57, 100, 67, 107

Inorder Traversal - Binary Tree

- ***inorder traversal***: of a binary tree, means that you visit the left child first, then the root of the subtree, then the right child.
  - In a binary search tree, the nodes will be given in sorted order.




```
inorder(node):
  if node is null: return
  inorder(node.left)
  visit node
  inorder(node.right)
```

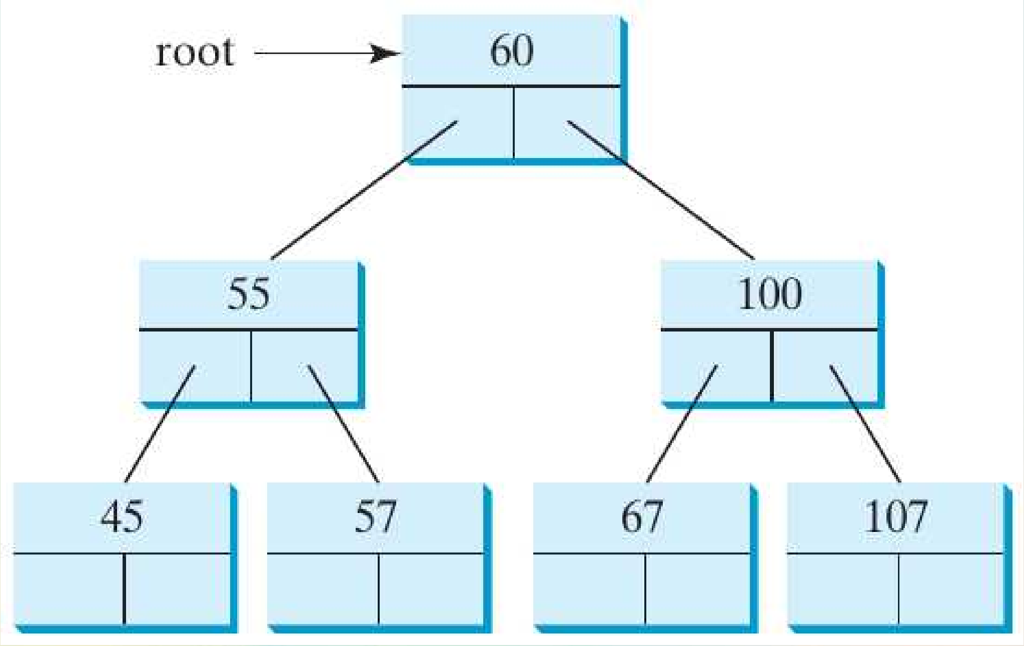
- In the example to the right the inorder traversal would be:

  45, 55, 57, 60, 67, 100, 107

Postorder Traversal - Binary Tree

- ***postorder traversal***: of a binary tree, means that you visit the left child first, then the right child, then the root of the subtree.
  - Finding the size of a directory uses post order.

```
postorder(node):
   if node is null: return
   postorder(node.left)
   postorder(node.right)
   visit node
```

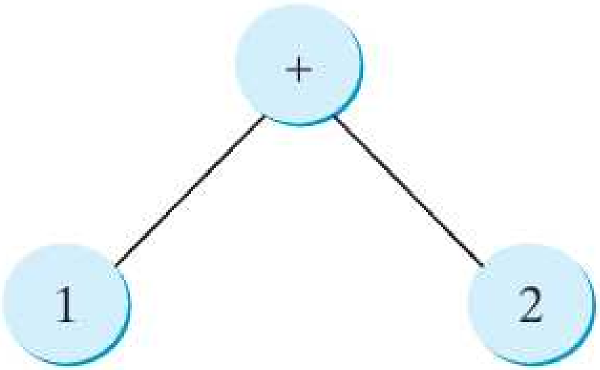


- In the example to the right the postorder traversal would be:

    45, 57, 55, 67, 107, 100, 60

Traversal Mnemonic

You can use the following tree to help remember inorder, postorder, and preorder.

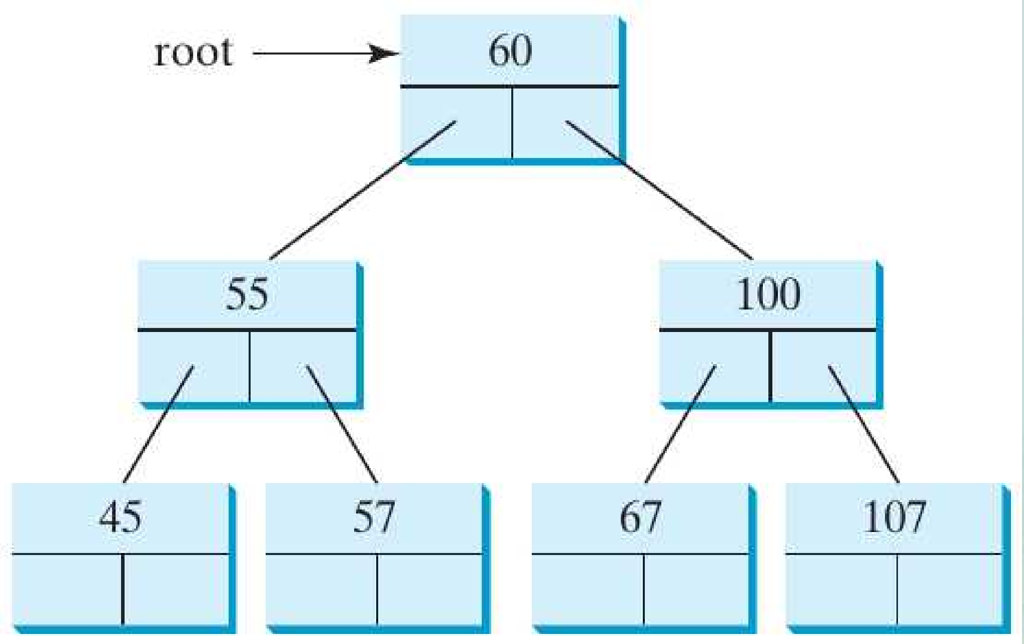


The inorder is 1 + 2, the postorder is 1 2 +, and the preorder is + 1 2.

Breadth-First Traversal - Binary Tree

- ***breadth-first traversal***: means that nodes are visited level by level.

```
breadthfirst(root):
  create a Queue Q a
  add root to Q
  while Q not empty:
    node = Q.dequeue()
    visit node
    Q.enqueue(node.left)
    Q.enqueue(node.right)
```



- In the example to the right the breadth-first traversal would be:

  60, 55, 100, 45, 57, 67, 107

Traversal Practice