Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Trees II: Binary Search Trees

""

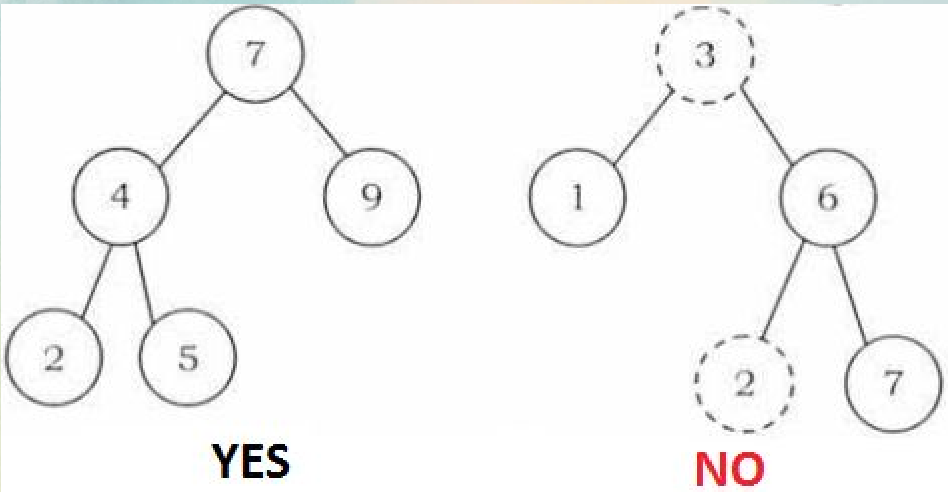
CS2013: Programming with Data Structures

Binary Search Trees

Binary Search Tree Basics

- Recall:
  - ***binary tree***: a tree where each node has zero, one, or two child nodes.

- A ***binary search tree*** adds the following properties:
  - the tree may not contain duplicate ***keys***
    - here a ***key*** is an item that you will be searching for.
    - binary search trees can be used to implement a **Map** or **Set** data structure.
  - the left subtree of a parent node only contains keys ***less than*** the parent node key.
  - the right subtree of a parent node only contains keys ***greater than*** the parent node key.



Binary Tree Node

```
public class BSTNode<E extends Comparable<E>> {
   protected E data;
   protected BSTNode<E> parent;
   protected BSTNode<E> left;
   protected BSTNode<E> right;

   public BSTNode(E data) {
      this.data = data;
   }
}
```

- BSTs can store key/value pairs if the tree is used to implement a map.
  - In this case the Node class for the tree would have an extra data field to store the key in addition to the data.
- BSTs can also store individual items if the tree is not used to implement a Map
  - In our example we do not store a separate key value, so the data is its own key.

Binary Search Tree Algorithms

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?

BST Algorithm - insert()

- Insertion into a BST is also very easy to implement.
- The first step is to find the insertion point using the same idea as the **`find()`** function discussed previously.
  - The insertion point will be the future parent of the data we want to add.
- Cases to consider:
  - Duplicate item already exists in the tree.
    - throw a DuplicateItemException.

```
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
```

BST Algorithm - insert()

- Now that we have the insertion point, it is very easy to add the new node to our tree.

- Cases to consider:
  - The tree may be empty.
  - A **`DuplicateItemException`** may have been thrown by the **`insertionPoint()`** algorithm.
    - In this case we don't want to deal with the exception in the **`insert()`** method.
    - We want to allow someone using our method to deal with it in their own way.
    - Just catch and rethrow the exception.

```
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.

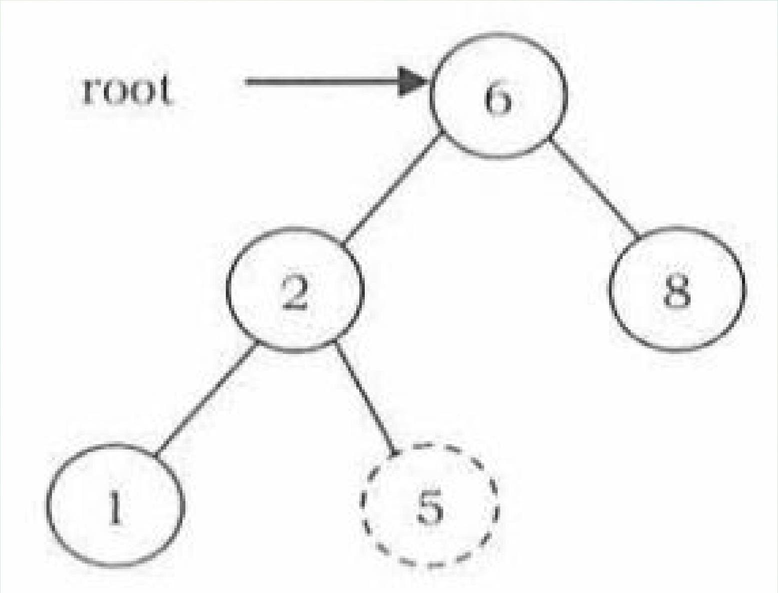
- First find the node you want to delete:
  - **`nodeToDelete()`**: We will just reimplement the find() method to return the actual node, instead of the data.

```
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?



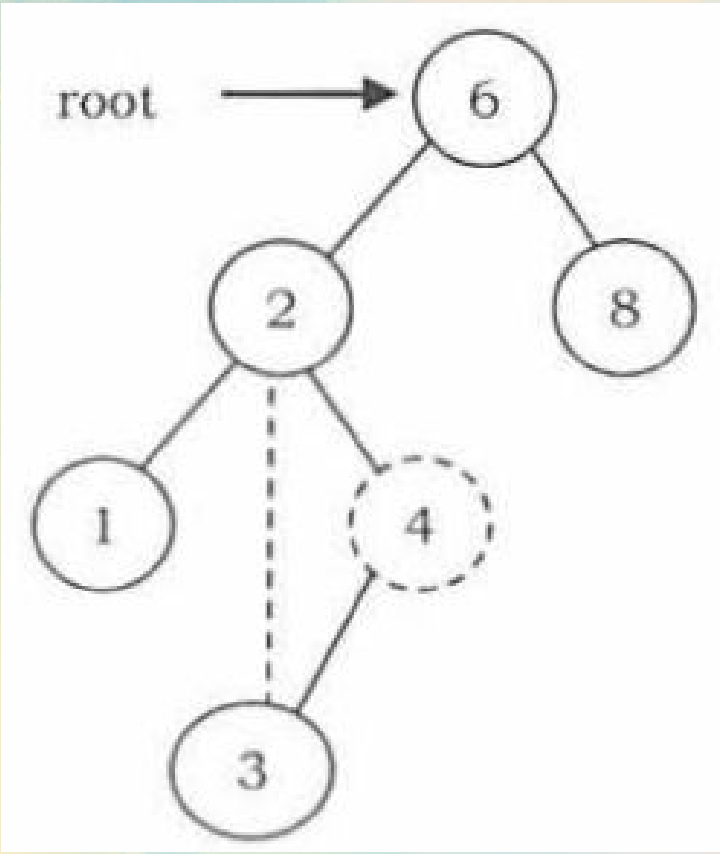
Left or Right Child?

```
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?

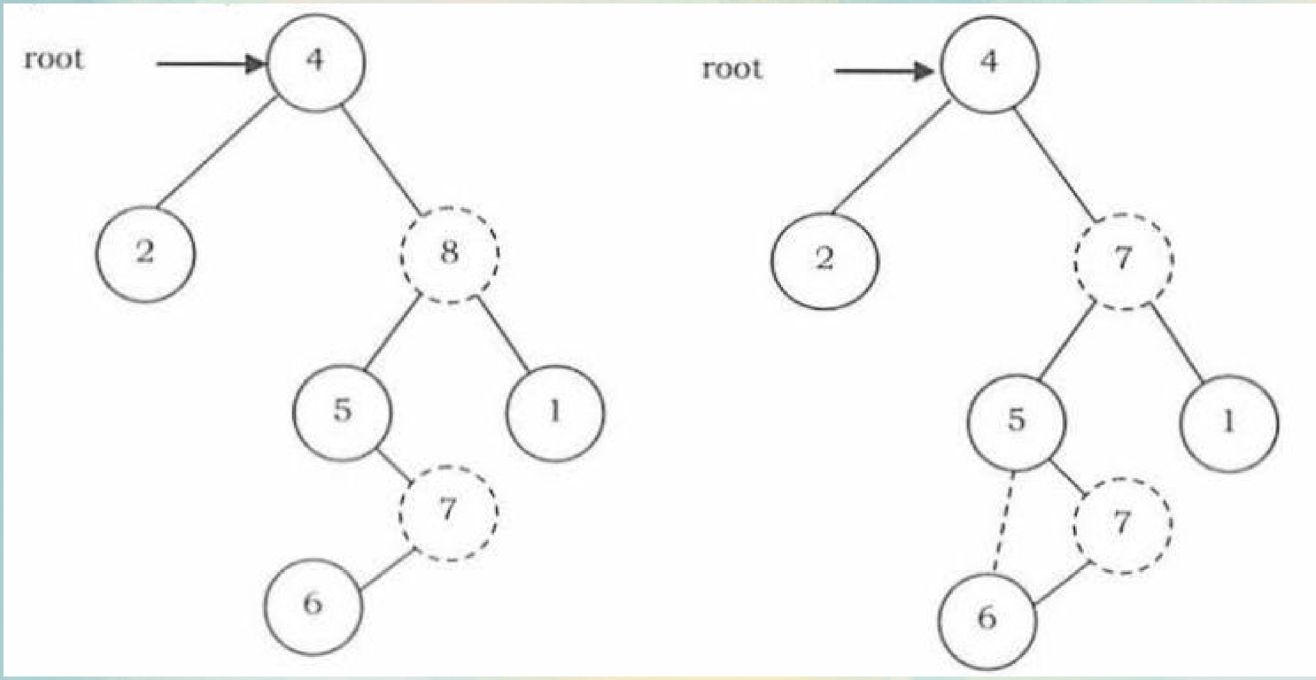


numChildren()

```
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 with Stacks

Preorder Traversal

```
preorder(node):
  if tree is empty:
    ERROR: Tree is empty.
    return

  S = create a stack
  S.push(node)

  while !S.empty():
    current = S.pop()
    visit current
    if current.right != null:
      S.push(current.right)
    if current.left != null:
      S.push(current.left)
```

Inorder Traversal

```
inorder(node):
  if tree is empty:
    ERROR: Tree is empty.
    return

  S = create a Stack
  current = node

  while !S.isEmpty() || current != null:
    if(current != null):
      S.push(current)
      current = current.left
    else:
      current = S.pop()
      visit current
      current = current.right
```

Postorder Traversal

```
postorder(node) {
  if tree is empty:
    ERROR: Tree is empty.
    return

  S1 = create the first Stack
  S2 = create the second stack

  S1.push(node)

  while !S1.isEmpty():
    current = S1.pop()
    S2.push(current)

    if current.left != null:
      S1.push(current.left);
    if current.right != null:
      S1.push(current.right);

  while !S2.isEmpty():
    current = S2.pop()
    visit current
```

Title

Title

Title

Title