CSC148 Intro. to Computer Science
Lecture 9: BST (insert, delete)

Amir H. Chinaei, Summer 2016
Office Hours: R 10-12 BA4222
achchinaei@cs.toronto.edu
http://www.cs.toronto.edu/~achchinaei/

Course page:
http://www.cs.toronto.edu/~achchinaei/teaching/20164/csc148/

Last week
- Binary trees (branch factor =2)
- Depth-first traversal
  - inorder, preorder, and postorder
- Breadth-first traversal
  - level-order
- Binary Search Trees

Today
- More on BST
  - insert
  - delete

Binary Search Trees
- Add ordering conditions to a binary tree:
  - data are comparable
  - data in left subtree are less than node.data
  - data in right subtree are more than node.data

Binary Search Trees
- A BST with 1 node has height 1
- A BST with 3 nodes may have height 2
- A BST with 7 nodes may have height 3
- A BST with 15 nodes may have height 4
- A BST with \( n \) nodes may have height \( \lceil \lg n \rceil \)

- If the BST is "balanced", then we can check whether an element is present in about \( \lg n \) node accesses
  - This is significantly faster than a linear search: \( O(n) \)

```python
def bst_contains(node, value):
    # Return whether tree rooted at node contains value.
    Assume node is the root of a Binary Search Tree
    :param node: node of a Binary Search Tree
    :param value: value to search for
    :param node: BinaryTree|None
    :param value: object
    :type: bool
    >>> bst_contains(None, 5)
    False
    >>> bst_contains(BinaryTree(7, BinaryTree(5), BinaryTree(9)), 5)
    True
```

**Binary Trees**

### bst_contains

```python
def bst_contains(node, value):
    """Return whether tree rooted at node contains value.
    Assume node is the root of a Binary Search Tree:
    param node: node of a Binary Search Tree
    :type node: BinaryTree|None
    param value: value to search for
    :type value: object
    rtype: bool
    ""
    if node is None:
        return False
    elif value < node.data:
        return bst_contains(node.left, value)
    elif value > node.data:
        return bst_contains(node.right, value)
    else:
        return True
```

- Test examples:

  ```python
  bst_contains(None, 5)  # False
  bst_contains(BinaryTree(7, BinaryTree(5), BinaryTree(9)), 5)  # True
  ```

### bst_insert

```python
def insert(node, data):
    """Insert data in BST rooted at node if necessary, and return new root.
    Assume node is the root of a Binary Search Tree.
    :param node: root of a binary search tree.
    :type node: BinaryTree
    :param data: data to insert into BST, if necessary.
    :type data: object
    >>> b = BinaryTree(5)
    >>> b1 = insert(b, 3)
    >>> print(b1)
    5
    3
    ""
    return_node = node
    if not node:
        return_node = BinaryTree(data)
    elif data < node.data:
        node.left = insert(node.left, data)
    elif data > node.data:
        node.right = insert(node.right, data)
    else:
        pass
    return return_node
```

- Let's trace it for a few examples:

### bst_delete

- First locate the node that contains the element and also its parent node.
- Let `current` point to the node that contains the element in the tree and `parent` point to the parent of the `current` node.
- There are two cases to consider …

**Case 1: The current node has no left child**

- Simply connect the `parent` with the right child of the `current` node.
Example for Case 1: Deleting node 10

Connect the parent of node 10 with the right child of node 10.

Case 2: The current node has a left child.

- Let right_most point to the node that contains the largest element in the left subtree of the current node.
- Let parent_of_right_most point to the parent node of the right_most node.

Then:
1. Replace the element value in the current node with the one in the right_most node.
2. Connect the parent_of_right_most node with the left child of the right_most node.

Case 2 (diagram)

Example for Case 2. Deleting node 20

More Examples

Case 1 or 2?

More Examples

Case 1 or 2?
**More Examples**

**Case 1 or 2?**

**bst_delete**

- First locate the nodes that contain the element and its parent. Call them current and parent.

```python
parent = None
current = root
while current is not None and current.data != data:
    if data < current.data:
        parent = current
        current = current.left
    elif data > current.data:
        parent = current
        current = current.right
    else:
        pass  # Element is in the tree pointed at by current
if current is None:
    return False  # Element is not in the tree
```

**Case I:** bst_delete

# Case I: current has no left child
if current.left is None:
    # Connect the parent with the right child of the current node
    # Special case, assume the node being deleted is at root
    if parent is None:
        current = current.right
    else:
        # Identify if parent left or parent right should be connected
        if data < parent.data:
            parent.left = current.right
        else:
            parent.right = current.right
    # Case 2: The current node has a left child
else:
    # Case 2: current has a left child and also its parent
    parent_of_right_most = current
    right_most = current.left
    while right_most.right is not None:
        parent_of_right_most = right_most
        right_most = right_most.right
    # Replace the element in current by the element in rightMost
    current.element = right_most.element
    # Eliminate rightMost node
    if parent_of_right_most.right == right_most:
        parent_of_right_most.right = right_most.left
    else:
        # Special case: parent_of_right_most == current
        parent_of_right_most.left = right_most.left
    return True  # Element deleted successfully
```

**Summary**

- **Homework:**
  - In Slides 12 and 14,
    - replace every left with right, every right with left, and also largest with smallest.
  - And, implement the method.

- **Next Week:**
  - How bst_delete can be written recursively?