

CSCI48 Intro. to Computer Science

Lecture 9: BST (insert, delete)

Amir H. Chinatei, Winter 2016

Office Hours: W 16:00–17:45 BA4222

ahchinatei@cs.toronto.edu
<http://www.cs.toronto.edu/~ahchinatei/>

Course webpage:
<http://www.cdf.toronto.edu/~csci48h/winter>

Binary Trees 4-1

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 Trees 4-2

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 Trees 4-3

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: $O(n)$

Binary Trees 4-4

bst_contains

```
def bst_contains(node, value):
    """
    Return whether tree rooted at node contains value.

    Assume node is the root of a Binary Search Tree

    @param BinaryTree|None node: node of a Binary Search Tree
    @param object value: value to search for
    @rtype: bool

    >>> bst_contains(None, 5)
    False
    >>> bst_contains(BinaryTree(7, BinaryTree(5), BinaryTree(9)), 5)
    True
    """

    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
```

Binary Trees 4-5

bst_insert

```
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 BinaryTree node: root of a binary search tree.
    @param object data: data to insert into BST, if necessary.

    >>> b = BinaryTree(5)
    >>> b1 = insert(b, 3)
    >>> print(b1)
    3
    <BLANKLINE>

    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:
        # nothing to do
        pass
    return return_node
```

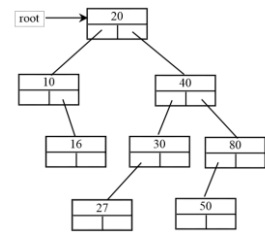
Binary Trees 4-6

bst_insert

- ❖ Let's trace it for a few examples:

Binary Trees 4-7

bst_delete



Binary Trees 4-8

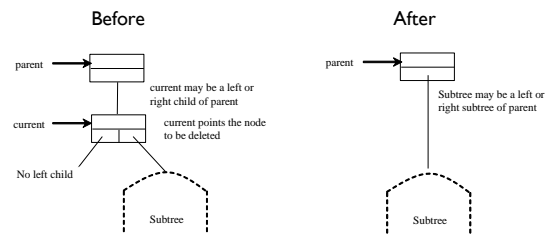
bst_delete

- ❖ 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 ...

Binary Trees 4-9

Case 1: The current node has no left child

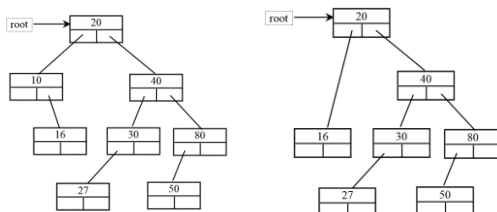
- ❖ Simply connect the parent with the right child of the current node.



Binary Trees 4-10

Example for Case 1. Deleting node 10

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



11

Case 2: The current node has a left child.

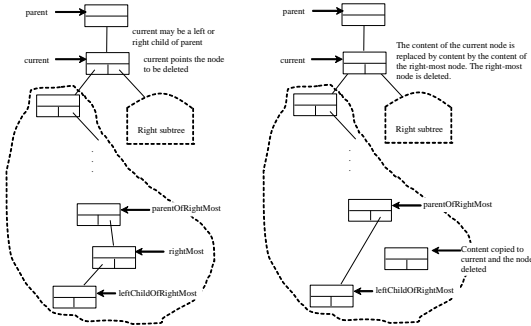
- ❖ Let right_most point to the node that contains the largest element in the left subtree of the current node.
- ❖ Let parent_of_right_most point to the parent node of the right_most node.

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.

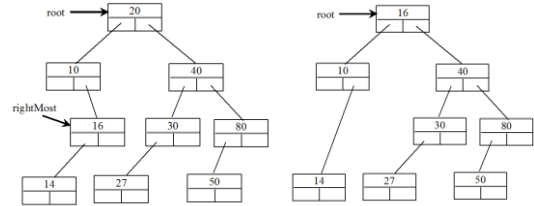
12

Case 2 (diagram)



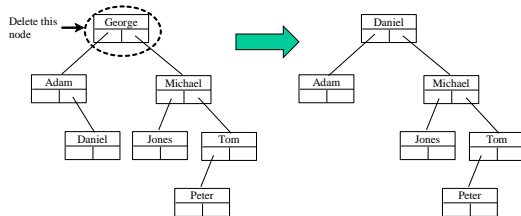
13

Example for Case 2. Deleting node 20



14

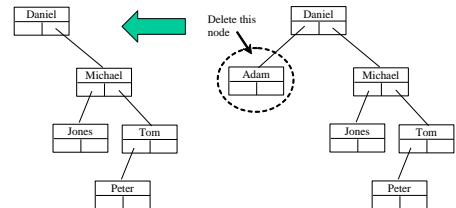
More Examples



Case 1 or 2? 2

15

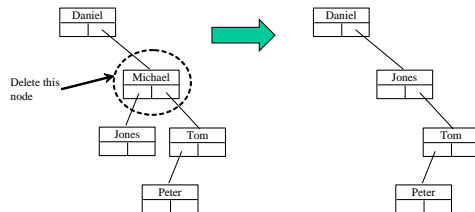
More Examples



Case 1 or 2? 1

16

More Examples



Case 1 or 2? 2

17

bst_delete

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

```
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
```

18

Case I: bst_delete

```
# Case 1: 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
else:
    # Case 2: The current node has a left child
```

19

Case II: bst_delete

```
# Locate the rightmost node in the left subtree of
# the current node 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 # Keep going to the 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
```

20

Summary

- ❖ Homework:
 - In Slides 10 and 12,
 - 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?

21