

CSCI 48 *Intro. to Computer Science*

Lecture 10: *BST Recursive Delete, Efficiency of Algorithms*

Amir H. Chanaei, Winter 2016

Office Hours: W 16:00–17:45 BA4222

ahchinaei@cs.toronto.edu

<http://www.cs.toronto.edu/~ahchinaei/>

Course webpage:

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

Last week

❖ BST

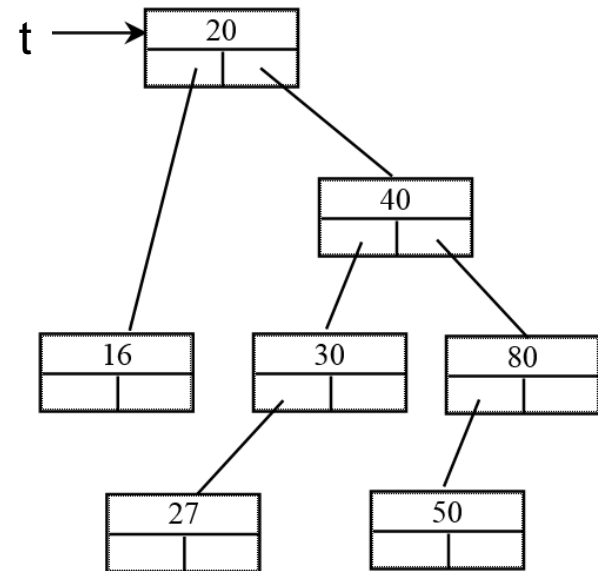
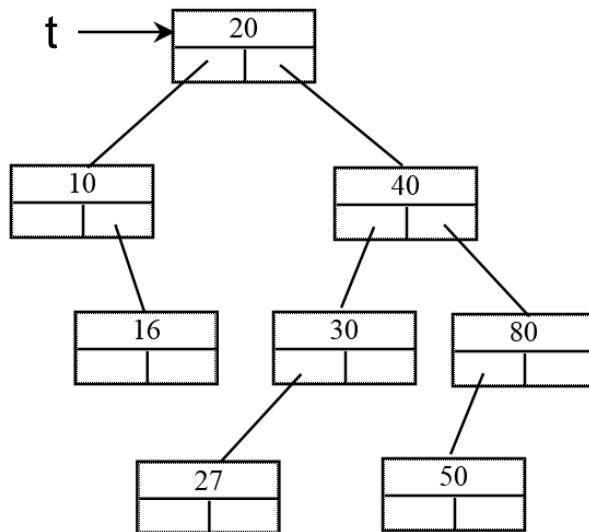
- Insert (and trace)
- Iterative delete

❖ Today

- More on BST
 - Recursive delete
- Efficiency

bst_del_rec

- ❖ Let's define it as deleting a node (if exists) from the BST and returning the resulting BST
- ❖ Example:
 - `t = bst_del_rec(t, 10)`
 - deletes 10 from BST t and returns the reference to the tree



bst_del_rec(tree, data)

❖ Base case

- If the tree is none return none

```
if not tree:  
    return None
```

❖ Recursive case I

- If data is less than tree data, delete it from left child

```
if data < tree.data:  
    tree.left = bst_del_rec(tree.left, data)
```

❖ Recursive case II

```
if data > tree.data:  
    tree.right = bst_del_rec(tree.right, data)
```

bst_del_rec(tree, data)

- ❖ What does it mean if none of the above if's have been true?
 - **We have located the tree node to be deleted**
- ❖ What next?
- ❖ **There are two cases to consider ...**
- ❖ **Case I:**
 - If the tree node does not have a left child,
 - return the right child

```
if tree.left is None:  
    return tree.right
```

bst_del_rec

- ❖ Recall examples for case I:

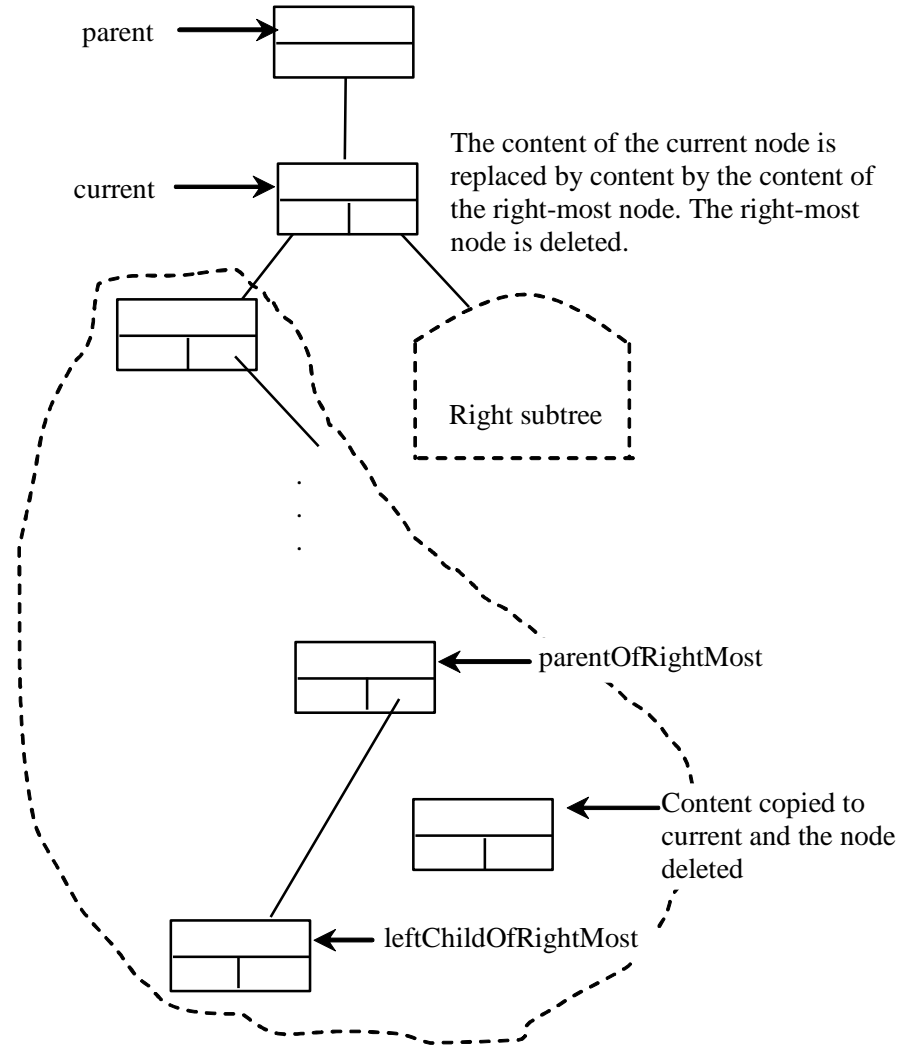
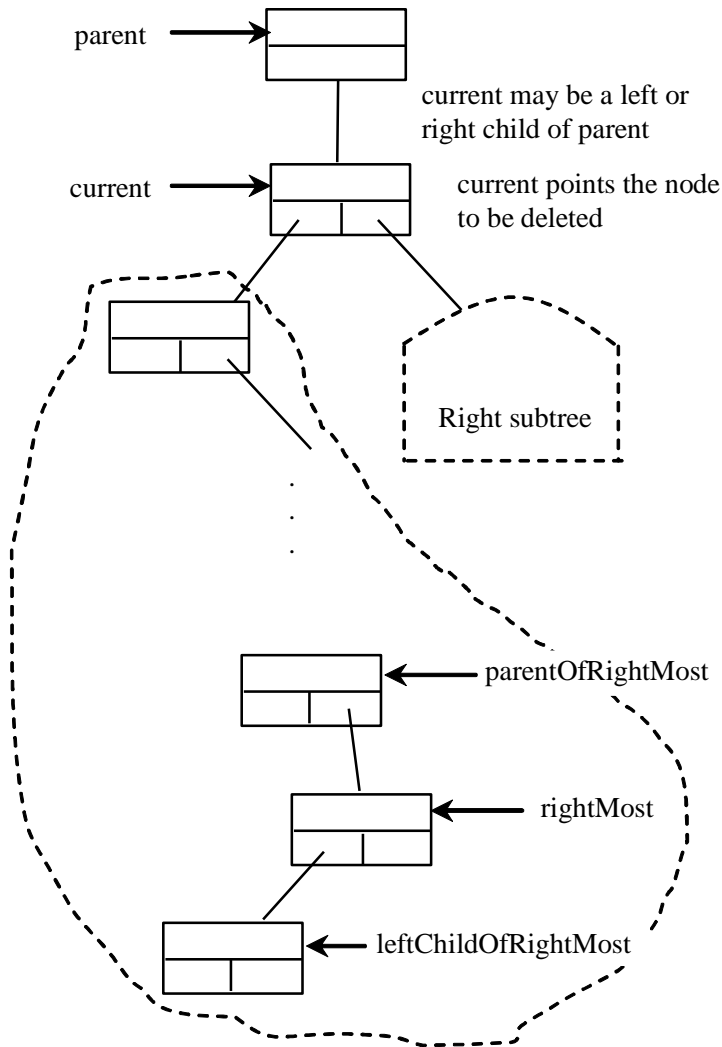
bst_del_rec(tree, data)

❖ Case II:

- If the tree node does have a left child,
 - find the largest node of the left child
 - replace the tree node data with the largest just found
 - delete the largest

```
if tree.left is not None:  
    largest = findmax(tree.left)  
    tree.data = largest.data  
    tree.left = bst_del_rec(tree.left, largest.data)  
return tree
```

Case 2 (diagram)



bst_del_rec

- ❖ Recall examples for case II:

bst_del_rec(tree, data)

#putting everything together

base case

```
if not tree:  
    return None
```

recursive case I

```
elif data < tree.data:  
    tree.left = bst_del_rec(tree.left, data)
```

recursive case II

```
elif data > tree.data:  
    tree.right = bst_del_rec(tree.right, data)
```

left child is empty

```
elif tree.left is None:  
    return tree.right
```

left child is not empty

```
else:
```

```
    largest = findmax(tree.left)  
    tree.data = largest.data  
    tree.left = bst_del_rec(tree.left, largest.data)  
    return tree
```

helper

```
def findmax(tree):  
    return tree if not tree.right else findmax(tree.right)
```

Efficiency of algorithms

- ❖ BST: iterative delete vs. recursive delete?
 - Extra memory?
 - Constant vs. in order of height of tree
 - $O(1)$ vs. $O(\lg n)$ if balanced or $O(n)$ otherwise
 - Time?
 - Although both in order of height of tree, the latter requires more work
- ❖ **Fibonacci:** iteration vs. recursion?
 - Extra memory?
 - $O(1)$ vs. $O(n)$
 - Time?
 - $O(n)$ vs. $O(2^n)$!!

Efficiency of algorithms

n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	$3 \times 10^{-11} \text{ s}$	10^{-10} s	$3 \times 10^{-10} \text{ s}$	10^{-9} s	10^{-8} s	$3 \times 10^{-7} \text{ s}$
10^2	$7 \times 10^{-11} \text{ s}$	10^{-9} s	$7 \times 10^{-9} \text{ s}$	10^{-7} s	$4 \times 10^{11} \text{ yr}$	*
10^3	$1.0 \times 10^{-10} \text{ s}$	10^{-8} s	$1 \times 10^{-7} \text{ s}$	10^{-5} s	*	*
10^4	$1.3 \times 10^{-10} \text{ s}$	10^{-7} s	$1 \times 10^{-6} \text{ s}$	10^{-3} s	*	*
10^5	$1.7 \times 10^{-10} \text{ s}$	10^{-6} s	$2 \times 10^{-5} \text{ s}$	0.1 s	*	*
10^6	$2 \times 10^{-10} \text{ s}$	10^{-5} s	$2 \times 10^{-4} \text{ s}$	0.17 min	*	*

Recursive vs iterative

- ❖ Recursive functions impose a loop
 - ❖ The loop is implicit and the compiler/interpreter (here, Python) takes care of it
 - ❖ This comes at a price: time & memory
 - ❖ The price may be negligible in many cases
-
- ❖ After all, no recursive function is more efficient than its iterative equivalent

Recursive vs iterative cont'ed

- ❖ Every recursive function can be written iteratively (by explicit loops)
 - may require stacks too
- ❖ yet, when the nature of a problem is recursive, writing it iteratively can be
 - time consuming, and
 - less readable
- ❖ So, recursion is a very powerful technique for problems that are naturally recursive