# Analysis of Algorithms

## Binary Search Trees

Andres Mendez-Vazquez

September 30, 2018



Notes

# Outline





Notes

## Why Binary Search Trees?

**Compared them with an array representation**

Ouch!!! Insertion, Search and Deletion are quite expensive with the $O(n)$.

**Instead Binary Search Trees**

Since they are node based the cost of moving an element either into the collection or out of the collection is faster.



Notes

## Binary Search Tree Concepts

**Definition**

A binary search tree (BST) is a data structure where each node posses three fields $left$, $right$ and $p$.

- They represent its left child, right child and parent.
- In addition, each node has the field key.

**Property**

- Let $x$ be a node in a binary search tree. If $y$ is a node in the left subtree of $x$, then $key[y] \leq key[x]$.
- Similarly, if $y$ is a node in the right subree of $x$, then $key[x] \leq key[y]$.



Notes

## In Order Walk

**This walk allows to print the keys in sorted order!**

Inorder-tree-walk(x)

1. if $x \neq$ NIL
2. Inorder-tree-walk($x$.left)
3. print $x$.key
4. Inorder-tree-walk($x$.right)



Notes

## Cost of inorder walk

**Theorem 12.1**

If $x$ is the root of an n-node subtree, then the call Inorder-tree-walk$(x)$ takes $\Theta(n)$ time.

**Proof:**

Let $T(n)$ denote the time taken by Inorder-tree-walk$(x)$ when called at the root.

**First**

- Since Inorder-tree-walk$(x)$ visit all the nodes then we have that $T(n) = \Omega(n)$.
- Thus, you need to prove $T(n) = O(n)$?



Notes

## Proof of inorder walk, $T(n) = O(n)$

### First

For $n = 0$, the method takes a constant time $T(0) = c$ for some $c > 0$.

### Now for $n > 0$

We have the following situation:

1. Left subtree has $k$ nodes
2. Right subtree has $n - k - 1$ nodes



Notes

## Substitution Method

### We have finally

$$T(n) = T(k) + T(n - k - 1) + d$$

1. $T(k)$ is the amount of work done in the left
2. $T(n - k - 1)$ is the amount of work done in the right
3. $d > 0$ reflects an upper bound for the in-between work done for the print.

### We use the substitution method to prove that $T(n) = O(n)$

This can be done if we can bound $T(n)$ by bounding it by

$$(c + d)n + c \tag{1}$$



Notes

## Thus

**For $n = 0$**

$$T(0) = c = (c + d) \times 0 + c \tag{2}$$



Notes

## Now, By Substitution Method

**For $n > 0$**

$$\begin{aligned} T(n) &\leq T(k) + T(n-k-1) + d \\ &= ((c+d)\,k + c) + ((c+d)\,(n-k-1) + c) + d \\ &= (c+d)\,n + c - (c+d) + c + d \\ &= (c+d)\,n + c \end{aligned}$$

**Thus**

$$T(n) = \Theta(n) \tag{3}$$



Notes

## What may we use for a search?

### Given a key $k$, we have the following Trichotomy Law

1. $x.key == k$
2. $x.key > k$
3. $x.key < k$

### This allows us to take decisions

Go to the left or go to the right down the tree!!!

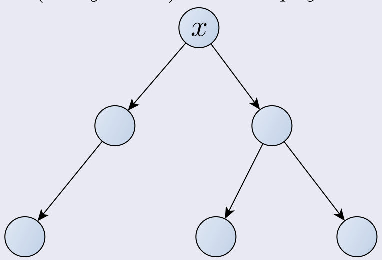


Notes

## Case 1

### Return Payload

if $(x.key == k)$ return $x.payload$






Notes

## Searching

### Searching

Tree-search$(x, k)$

1. if $x == $ NIL or $k == x.key$
2. return x
3. if $k < x.key$
4. return Tree-search$(x.left, k)$
5. else return Tree-search$(x.right, k)$

### Complexity

$$O(h) \tag{4}$$

where $h$ is the height of the tree $\Rightarrow$ **we look for well balanced trees.**



Notes

## Minimum and Maximum

### Minimum and Maximum

Tree-minimum$(x)$

1. while $x.left \neq$NIL
2. $x = x.left$
3. return $x$

### Complexity

$$O(h) \tag{5}$$

where $h$ is the height of the tree $\Rightarrow$ **we look for well balanced trees.**



Notes

# Ouch!!!

## At the End We Delete

- Thus, we have a problem!!!
- We need to maintain the Binary Search Property.

## A simple idea

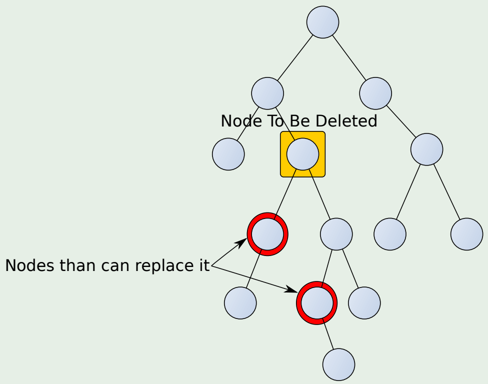
Move the previous or next element to the deleted position!!!



Notes

# We want to do the following

## We have then






Notes

## Tree-Delete

### TREE-DELETE$(T, z)$

```
if z.left == NIL
    Transplant(T, z, z.right)
elseif z.right == NIL
    Transplant(T, z, z.left)
else
    y=Tree-minimum(z.right)
    if y.p ≠ z
```

**Case 1**

- Basically if the element $z$ to be deleted has a NIL left child simply replace $z$ with that child!!!

**Case 2**

- Basically if the element $z$ to be deleted has a NIL right child simply replace $z$ with that child!!!

**Case 3**

- The $z$ element has not empty children you need to find the successor of it.



Notes

## Support Operations: Transplant

### Transplant$(T, u, v)$

```
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠NIL
    v.p = u.p
```

**Case 1**

- If $u$ is the root then make the root equal to $v$

**Case 2**

- if $u$ is the left child make the left child of the parent of $u$ equal to $v$

**Case 3**

- Similar to the second case, but for right child

**Case 4**

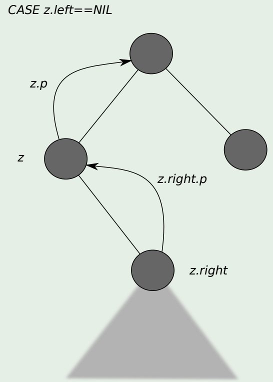
- If $v \neq$ NIL then make the parent of $v$ the parent of $u$



Notes

## Example: Deletion in BST

### Case $z.left == NIL$



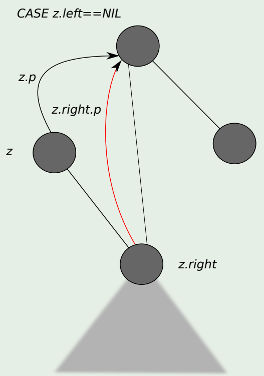

- if $z.left ==$ NIL
- Transplant$(T, z, z.right)$...



Notes

## Example: Deletion in BST

### Case $z.left == NIL$




Transplant$(T, z, z.right)$

- elseif $z == z.p.left$
- $z.p.left = z.right$
- if $z.right \neq$NIL
- $z.right.p = z.p$

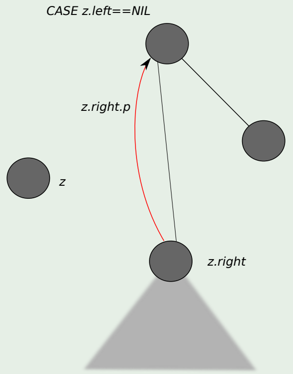


Notes

## Example: Deletion in BST

**Case $z.left == NIL$**

Remove the node z once you get out of the procedure



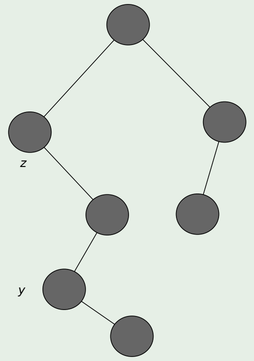



Notes

## Another Example: Deletion in BST

**Case $z.left \neq NIL$ and $z.right \neq NIL$**

- $y$=Tree-minimum($z.right$)



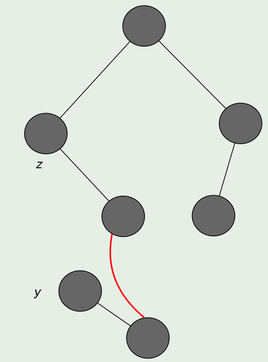



Notes

## Another Example: Deletion in BST

### Case $z.left \neq NIL$ and $z.right \neq NIL$

- if $y.p \neq z$
- Transplant$(T, y, y.right)$



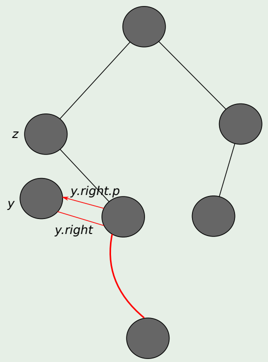



Notes

## Another Example: Deletion in BST

### Case $z.left \neq NIL$ and $z.right \neq NIL$

- $y.right = z.right$
- $y.right.p = y$



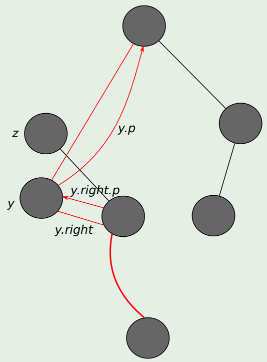



Notes

## Another Example: Deletion in BST

### Case $z.left \neq NIL$ and $z.right \neq NIL$

- Transplant$(T, z, y)$
- $y.left = z.left$
- $y.left.p = y$



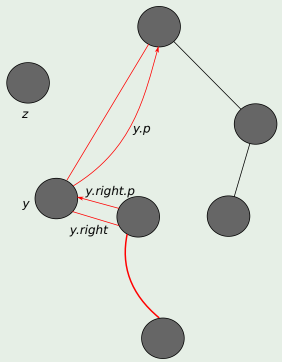



Notes

## Another Example: Deletion in BST

### Case $z.left \neq NIL$ and $z.right \neq NIL$

- Transplant$(T, z, y)$
- $y.left = z.left$
- $y.left.p = y$






Notes

## What do we need?

**Tree Height**

To describe AVL trees we need the concept of tree height

**Definition**

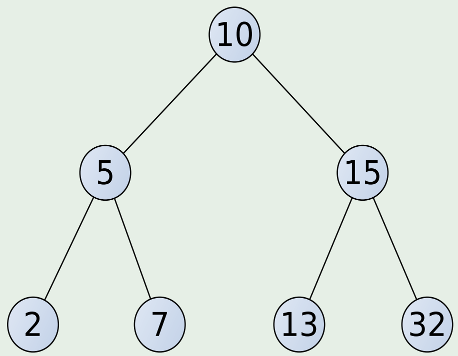
The maximal length of a path from the root to a leaf.



Notes

## Example

**Height = 3**






Notes

## We want the following

**Height Invariant**

**At any node in the tree, the heights of the left and right sub-trees differs by at most 1.**



Notes

## Thus, it is necessary to add an extra field to the Node Structure

**The Code**

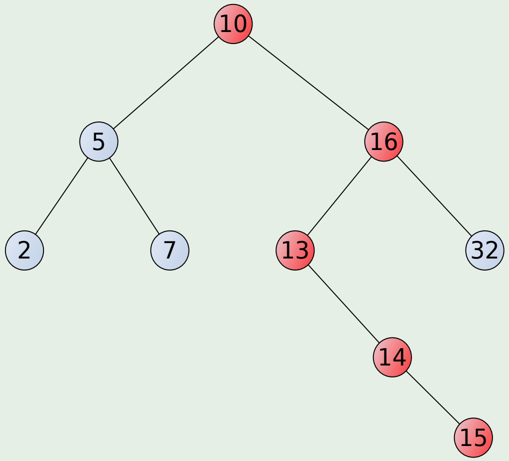
```
class Node():
  def __init__():
    self.key = None
    self.height = 0
    self.Val = None
    self.left  = None
    self.right = None
```



Notes

## Example

### Violation of the Height Property






Notes

## Insertion

### Similar to the Insertion in a BST

With a Fix-up at the end of the insertion

### We have the following cases

1. Right Subtree is of height $h+1$ and the left subtree is of height $h$
2. Right Subtree is of height $h$ and the left subtree is of height $h+1$



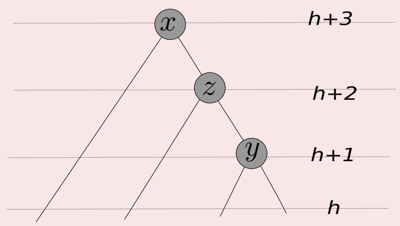
Notes

## Right Subtree is of height $h+1$ and the left subtree is of height $h$

**Now, if we are unlucky**

- Now, we insert in the **right subtree** of the right subtree.
- The result of inserting into the **right subtree** will give us a new right subtree of height $h+2$.

**This is how the tree looks like**






Notes

## Then

**This**

Which raises the height of the overall tree to $h+3$

**In addition**

In the new right subtree has height $h+2$

- Either its right or the left subtree must be of height h+1



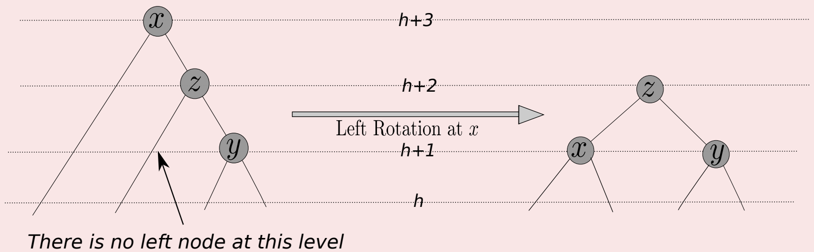
Notes

## Thus, we have

**This Violates the height invariance**

How we solve this?

**We can do the following**





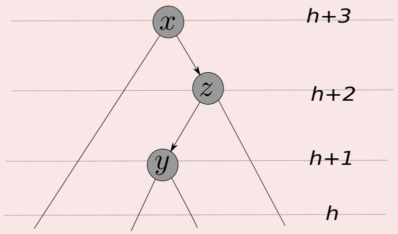

Notes

## Now, The second case

**We insert into the right subtree**

But now the left subtree of the right subtree has height $h+1$.

**Example**

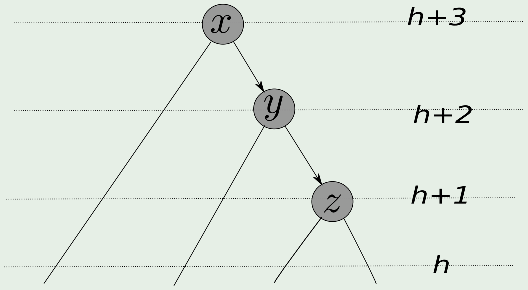





Notes

## We fix the problem by

**First a right rotation with respect to the $z$**

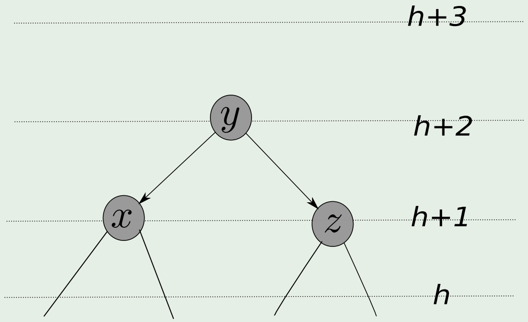





Notes

## We fix the problem by

**Now a left rotation with respect to the $x$**






Notes

## Excercises

### From Cormen's book, chapters 11 and 12

- 11.1-2
- 11.2-1
- 11.2-2
- 11.2-3
- 11.3-1
- 11.3-3
- 12.1-3
- 12.1-5
- 12.2-5
- 12.2-7
- 12.2-9
- 12.3-3



Notes

Notes