Analysis of Algorithms

Binary Search Trees

Andres Mendez-Vazquez

September 30, 2018

Outline

Outline

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.

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.

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.

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 $key[x] \leq key[y]$.

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 $key[x] \leq key[y]$.

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 $key[x] \leq key[y]$.

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 $key[x] \leq key[y]$.

Outline

In Order Walk

This walk allows to print the keys in sorted order!

Inorder-tree-walk(x)

```
if $x \neq$ NIL
    Inorder-tree-walk($x$.left)
    print $x$.key
    Inorder-tree-walk($x$.right)
```

In Order Walk

This walk allows to print the keys in sorted order!

Inorder-tree-walk(x)

```
if $x \neq$ NIL
    Inorder-tree-walk($x$.left)
    print $x$.key
    Inorder-tree-walk($x$.right)
```

In Order Walk

This walk allows to print the keys in sorted order!

Inorder-tree-walk(x)

```
if $x \neq$ NIL
    Inorder-tree-walk($x$.left)
    print $x$.key
    Inorder-tree-walk($x$.right)
```

In Order Walk

This walk allows to print the keys in sorted order!

```
Inorder-tree-walk(x)
if $x \neq$ NIL
        Inorder-tree-walk($x$.left)
        print $x$.key
        Inorder-tree-walk($x$.right)
```

In Order Walk

This walk allows to print the keys in sorted order!

```
Inorder-tree-walk(x)
  if $x \neq$ NIL
      Inorder-tree-walk($x$.left)
      print $x$.key
      Inorder-tree-walk($x$.right)
```

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.

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.

- Thus, you need to prove $T(n) = \Theta(n)$?

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 $T(n) = O(n)$?

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 $T(n) = O(n)$?

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

First

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

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

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

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

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.

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

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

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

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

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

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

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

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

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

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

Thus

For $n = 0$

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

Now, By Substitution Method

For $n > 0$

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

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

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

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

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

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

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

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

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

Outline

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!!!

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. $x.key < k$

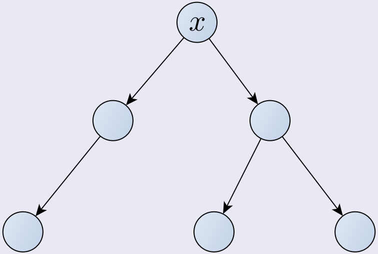
This allows us to take decisions

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

Case 1

Return Payload

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



Searching

Searching

Tree-search(x, k)

```
if $x$ == NIL or $k$ == $x.key$
    return x
if $k < x.key$
    return Tree-search($x.left, k$)
else return Tree-search($x.right, k$)
```

Searching

Searching

Tree-search(x, k)

```
if $x == \text{NIL}$ or $k == x.key$
        return x
if $k < x.key$
        return Tree-search$(x.left, k)$
else return Tree-search$(x.right, k)$
```

Complexity

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

where h is the height of the tree ⇒ **we look for well balanced trees.**

Searching

Searching

Tree-search(x, k)

```
if $x == $ NIL or $k == x.key$
        return x
if $k < x.key$
        return Tree-search$(x.left, k)$
else return Tree-search$(x.right, k)$
```

Complexity

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

where h is the height of the tree ⇒ **we look for well balanced trees.**

Searching

Searching

Tree-search(x, k)

```
if $x == $ NIL or $k == x.key$
    return x
if $k < x.key$
    return Tree-search$(x.left, k)$
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.**

Searching

Searching

Tree-search(x, k)

```
if $x == $ NIL or $k == x.key$
        return x
if $k < x.key$
        return Tree-search$(x.left, k)$
else return Tree-search$(x.right, k)$
```

Complexity

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

where h is the height of the tree ⇒ **we look for well balanced trees.**

Searching

Searching

Tree-search(x, k)

```
if $x == $ NIL or $k == x.key$
        return x
if $k < x.key$
        return Tree-search$(x.left, k)$
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.**

Searching

Searching

```
Tree-search($x, k$)
    if $x == $ NIL or $k == x.key$
        return x
    if $k < x.key$
        return Tree-search($x.left, k$)
    else return Tree-search($x.right, k$)
```

Complexity

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

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

Searching

Searching

```
Tree-search$(x, k)$
if $x == $ NIL or $k == x.key$
    return x
if $k < x.key$
    return Tree-search$(x.left, k)$
else return Tree-search$(x.right, k)$
```

Complexity

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

where h is the height of the tree ⇒ **we look for well balanced trees.**

Outline

Minimum and Maximum

Minimum and Maximum

Tree-minimum(x)

```
while $x.left \neq$NIL
        $x = x.left$
return $x$
```

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.**

Minimum and Maximum

Minimum and Maximum

Tree-minimum(x)

```
while $x.left \neq$NIL
        $x = x.left$
return $x$
```

Complexity

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

where h is the height of the tree ⇒ **we look for well balanced trees.**

Minimum and Maximum

Minimum and Maximum

Tree-minimum(x)

```
while $x.left \neq$ NIL
        $x = x.left$
return $x$
```

Complexity

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

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

Minimum and Maximum

Minimum and Maximum

Tree-minimum(x)

```
while $x.left \neq$NIL
        $x = x.left$
return $x$
```

Complexity

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

where h is the height of the tree ⇒ we look for well balanced trees.

Minimum and Maximum

Minimum and Maximum

Tree-minimum(x)

```
1. while $x.left$ ≠NIL
2.         $x = x.left$
3. return $x$
```

Complexity

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

where h is the height of the tree ⇒ **we look for well balanced trees.**

Outline

Ouch!!!

At the End We Delete

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

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!!!

Ouch!!!

At the End We Delete

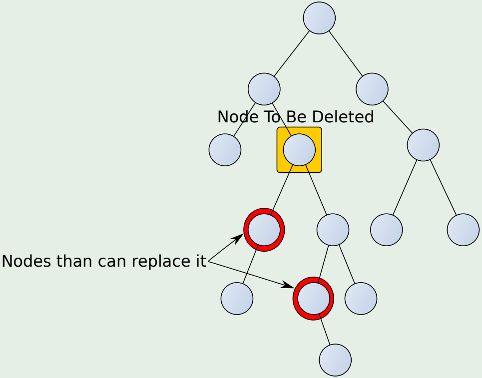
- Thus, we have a problem!!!
- We need to maintain the Binary Search Property.

A simple idea

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

We want to do the following

We have then



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 $y.p \neq z$
        Transplant$(T, y, y.right)$
        $y.right = z.right$
        $y.right.p = y$
    Transplant$(T, z, y)$
    $y.left = z.left$
    $y.left.p = y$
```

Case 1

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

Tree-Delete

TREE-DELETE(T, z)

```
if $z.left == \text{NIL}$
    Transplant$(T, z, z.right)$
elseif $z.right == \text{NIL}$
    Transplant$(T, z, z.left)$
else
    $y$=Tree-minimum$(z.right)$
    if $y.p \neq z$
        Transplant$(T, y, y.right)$
        $y.right = z.right$
        $y.right.p = y$
    Transplant$(T, z, y)$
    $y.left = z.left$
    $y.left.p = y$
```

Case 2

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

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 $y.p \neq z$
        Transplant$(T, y, y.right)$
        $y.right = z.right$
        $y.right.p = y$
    Transplant$(T, z, y)$
    $y.left = z.left$
    $y.left.p = y$
```

Case 3

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

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 $y.p \neq z$
        Transplant($T, y, y.right$)
        $y.right = z.right$
        $y.right.p = y$
    Transplant($T, z, y$)
    $y.left = z.left$
    $y.left.p = y$
```

Case 4

- if $y.p \neq z$ then $y.right$ takes the position of y after all $y.left ==$ NIL
    - take $z.right$ and make it the new $right$ of y
    - make the $(y.right == z.right).p$ equal to y

Tree-Delete

TREE-DELETE(T, z)

```
if $z.left == \text{NIL}$
    Transplant$(T, z, z.right)$
elseif $z.right == \text{NIL}$
    Transplant$(T, z, z.left)$
else
    $y$=Tree-minimum$(z.right)$
    if $y.p \neq z$
        Transplant$(T, y, y.right)$
        $y.right = z.right$
        $y.right.p = y$
    Transplant$(T, z, y)$
    $y.left = z.left$
    $y.left.p = y$
```

Case 4

- put y in the position of z
- make $y.left$ equal to $z.left$
- make the $(y.left == z.left).p$ equal to y

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 $v \neq$NIL
    $v.p = u.p$
```

Case 1

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

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 $v \neq$NIL
    $v.p = u.p$
```

Case 2

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

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 $v \neq$NIL
    $v.p = u.p$
```

Case 3

- Similar to the second case, but for right child

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 $v \neq$NIL
    $v.p = u.p$
```

Case 4

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

Outline

1. Binary Search Trees Concepts
   - Introduction

2. Binary Search Tree Operations
   - Walking on a Tree
   - Searching
   - Minimum and Maximum
   - Deletion in Binary Search Trees
   - Examples of Deletion

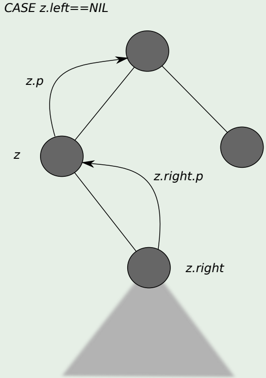
3. Balancing a Tree, AVL Trees
   - Adding a Height
   - The Height Problem
   - Insertions in AVL-Trees

4. Exercises
   - Some Excercises

Example: Deletion in BST

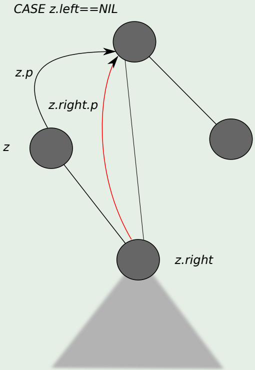
Case $z.left == NIL$

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



Example: Deletion in BST

Case $z.left == NIL$



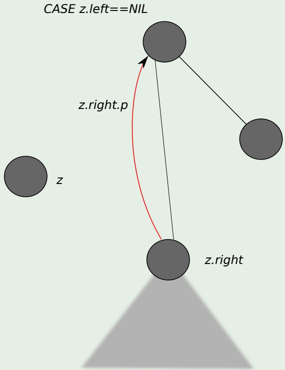
Transplant$(T, z, z.right)$

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

Example: Deletion in BST

Case $z.left == NIL$

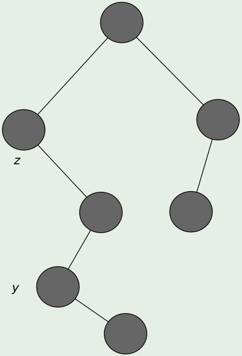
Remove the node z once you get out of the procedure



Another Example: Deletion in BST

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

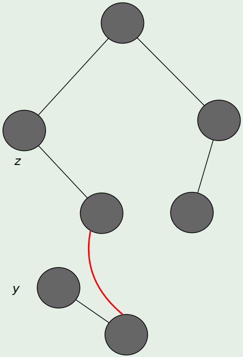
- y=Tree-minimum$(z.right)$



Another Example: Deletion in BST

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

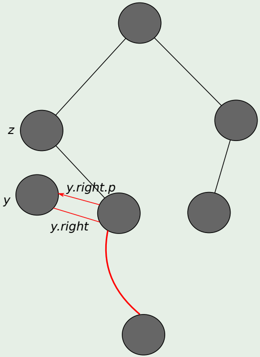
- if $y.p \neq z$
- Transplant$(T, y, y.right)$



Another Example: Deletion in BST

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

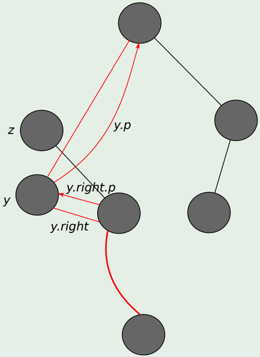
- $y.right = z.right$
- $y.right.p = y$



Another Example: Deletion in BST

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

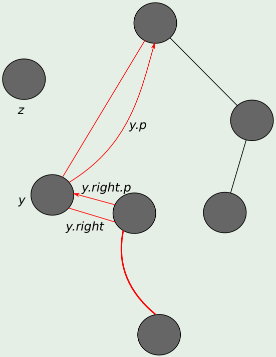
- Transplant(T, z, y)
- $y.left = z.left$
- $y.left.p = y$



Another Example: Deletion in BST

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

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



Outline

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.

What do we need?

Tree Height

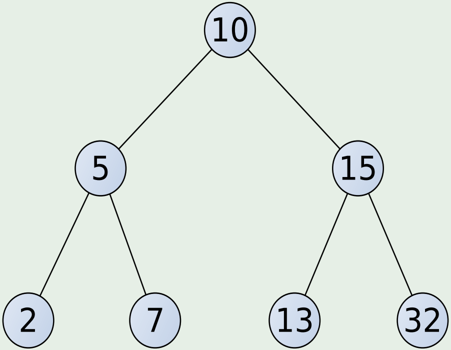
To describe AVL trees we need the concept of tree height

Definition

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

Example

Height = 3



Outline

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.

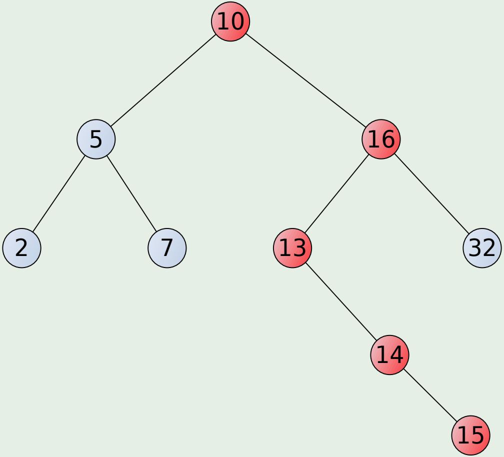
Thus, it is necessary to add an extra field to the Node Structure

The Code

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

Example

Violation of the Height Property



Outline

1 Binary Search Trees Concepts
- Introduction

2 Binary Search Tree Operations
- Walking on a Tree
- Searching
- Minimum and Maximum
- Deletion in Binary Search Trees
- Examples of Deletion

3 Balancing a Tree, AVL Trees
- Adding a Height
- The Height Problem
- Insertions in AVL-Trees

4 Exercises
- Some Excercises

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 $h+1$

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 $h+1$

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 $h+2$.

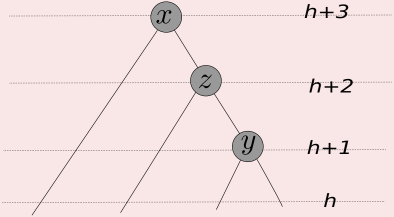
This is how the tree looks like

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



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

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

Thus, we have

This Violates the height invariance

How we solve this?

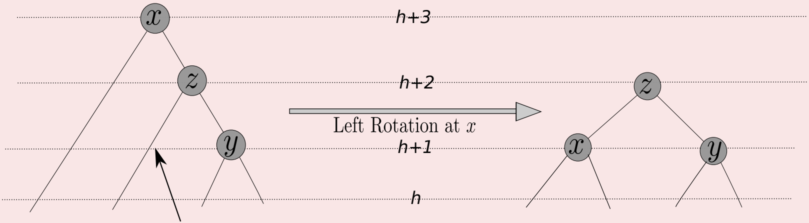
We can do the following

Thus, we have

This Violates the height invariance

How we solve this?

We can do the following



There is no left node at this level

Now, The second case

We insert into the right subtree

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

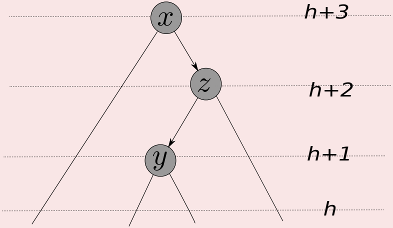
Example

Now, The second case

We insert into the right subtree

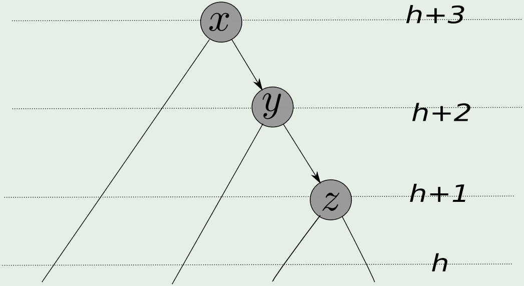
But now the left subtree of the right subtree has height $h+1$.

Example



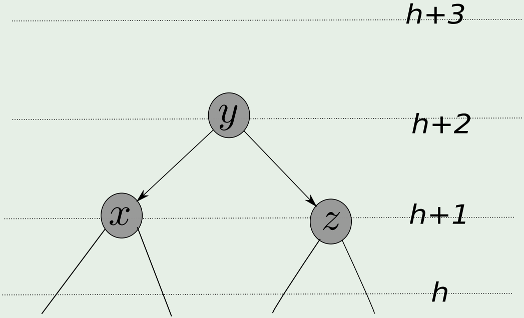
We fix the problem by

First a right rotation with respect to the z



We fix the problem by

Now a left rotation with respect to the x



Outline

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