# Balanced Search Trees

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 7

# Search trees

- `find()`, `insert()` and `delete()` all walk down a single path
- Worst-case: height of the tree
- An unbalanced tree with $n$ nodes may have height $O(n)$
- Balanced trees have height $O(\log n)$

# Search trees

- `find()`, `insert()` and `delete()` all walk down a single path
- Worst-case: height of the tree
- An unbalanced tree with $n$ nodes may have height $O(n)$
- Balanced trees have height $O(\log n)$
- How can we maintain balance as tree grows and shrinks

# Search trees

- `find()`, `insert()` and `delete()` all walk down a single path
- Worst-case: height of the tree
- An unbalanced tree with $n$ nodes may have height $O(n)$
- Balanced trees have height $O(\log n)$
- How can we maintain balance as tree grows and shrinks

## Defining balance

- Left and right subtrees should be "equal"
  - Two possible measures: `size` and `height`

# Search trees

- `find()`, `insert()` and `delete()` all walk down a single path
- Worst-case: height of the tree
- An unbalanced tree with $n$ nodes may have height $O(n)$
- Balanced trees have height $O(\log n)$
- How can we maintain balance as tree grows and shrinks

## Defining balance

- Left and right subtrees should be "equal"
  - Two possible measures: `size` and `height`
- `self.left.size()` and `self.right.size()` are equal?
  - Only possible for complete binary trees

# Search trees

- `find()`, `insert()` and `delete()` all walk down a single path
- Worst-case: height of the tree
- An unbalanced tree with $n$ nodes may have height $O(n)$
- Balanced trees have height $O(\log n)$
- How can we maintain balance as tree grows and shrinks

## Defining balance

- Left and right subtrees should be "equal"
  - Two possible measures: `size` and `height`
- `self.left.size()` and `self.right.size()` are equal?
  - Only possible for complete binary trees
- `self.left.size()` and `self.right.size()` differ by at most 1?
  - Plausible, but difficult to maintain

# Height balanced trees

- `self.height()` — number of nodes on longest path from root to leaf
  - $0$ for empty tree
  - $1$ for tree with only a root node
  - $1 + \max$ of heights of left and right subtrees, in general

# Height balanced trees

- `self.height()` — number of nodes on longest path from root to leaf
  - $0$ for empty tree
  - $1$ for tree with only a root node
  - $1 + \max$ of heights of left and right subtrees, in general
- Height balance
  - `self.left.height()` and `self.right.height()` differ by at most 1
  - AVL trees — Adelson-Velskii, Landis

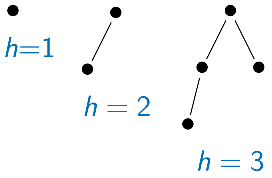
# Height balanced trees

- `self.height()` — number of nodes on longest path from root to leaf
  - $0$ for empty tree
  - $1$ for tree with only a root node
  - $1 + \max$ of heights of left and right subtrees, in general
- Height balance
  - `self.left.height()` and `self.right.height()` differ by at most 1
  - AVL trees — Adelson-Velskii, Landis
- Does height balance guarantee $O(\log n)$ height?

# Height balanced trees

- `self.height()` — number of nodes on longest path from root to leaf
  - 0 for empty tree
  - 1 for tree with only a root node
  - $1 + \max$ of heights of left and right subtrees, in general
- Height balance
  - `self.left.height()` and `self.right.height()` differ by at most 1
  - AVL trees — Adelson-Velskii, Landis
- Does height balance guarantee $O(\log n)$ height?

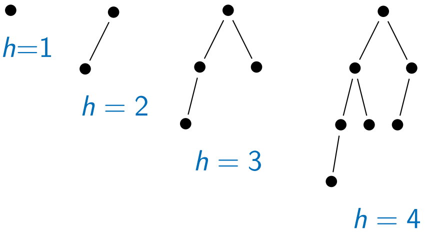
- Minimum size height-balanced trees



# Height balanced trees

- `self.height()` — number of nodes on longest path from root to leaf
  - 0 for empty tree
  - 1 for tree with only a root node
  - $1 + \max$ of heights of left and right subtrees, in general
- Height balance
  - `self.left.height()` and `self.right.height()` differ by at most 1
  - AVL trees — Adelson-Velskii, Landis
- Does height balance guarantee $O(\log n)$ height?

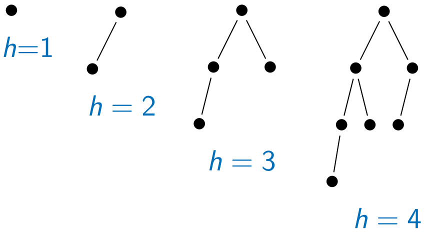
- Minimum size height-balanced trees




- General strategy to build a small balanced tree of height $h$
  - Smallest balanced tree of height $h - 1$ as left subtree
  - Smallest balanced tree of height $h - 2$ as right subtree

# Height balanced trees

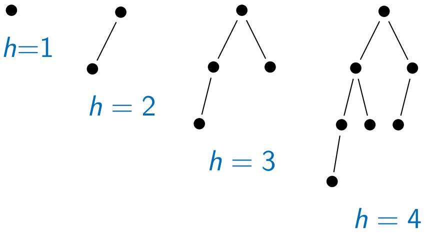
- Minimum size height-balanced trees




- General strategy to build a small balanced tree of height $h$
  - Smallest balanced tree of height $h - 1$ as left subtree
  - Smallest balanced tree of height $h - 2$ as right subtree

# Height balanced trees

- Minimum size height-balanced trees

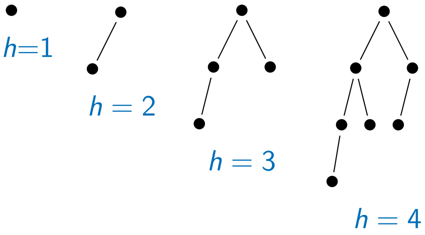



- General strategy to build a small balanced tree of height $h$
  - Smallest balanced tree of height $h-1$ as left subtree
  - Smallest balanced tree of height $h-2$ as right subtree

- $S(h)$, size of smallest height-balanced tree of height $h$

# Height balanced trees

- Minimum size height-balanced trees

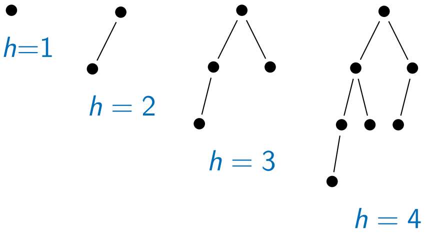



- General strategy to build a small balanced tree of height $h$
  - Smallest balanced tree of height $h - 1$ as left subtree
  - Smallest balanced tree of height $h - 2$ as right subtree

- $S(h)$, size of smallest height-balanced tree of height $h$
- Recurrence
  - $S(0) = 0$, $S(1) = 1$
  - $S(h) = 1 + S(h-1) + S(h-2)$

# Height balanced trees

- Minimum size height-balanced trees

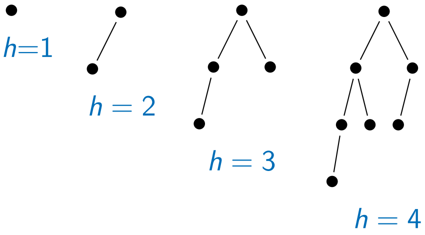



- General strategy to build a small balanced tree of height $h$
  - Smallest balanced tree of height $h - 1$ as left subtree
  - Smallest balanced tree of height $h - 2$ as right subtree

- $S(h)$, size of smallest height-balanced tree of height $h$
- Recurrence
  - $S(0) = 0$, $S(1) = 1$
  - $S(h) = 1 + S(h-1) + S(h-2)$
- Compare to Fibonacci sequence
  - $F(0) = 0$, $F(1) = 1$
  - $F(n) = F(n-1) + F(n-2)$

# Height balanced trees

- Minimum size height-balanced trees

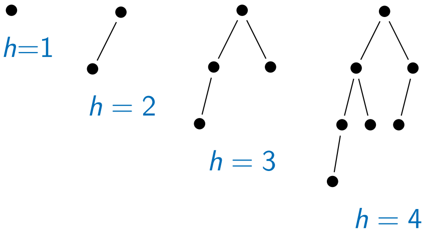



- General strategy to build a small balanced tree of height $h$
  - Smallest balanced tree of height $h - 1$ as left subtree
  - Smallest balanced tree of height $h - 2$ as right subtree

- $S(h)$, size of smallest height-balanced tree of height $h$
- Recurrence
  - $S(0) = 0$, $S(1) = 1$
  - $S(h) = 1 + S(h-1) + S(h-2)$
- Compare to Fibonacci sequence
  - $F(0) = 0$, $F(1) = 1$
  - $F(n) = F(n-1) + F(n-2)$
- $S(h)$ grows exponentially with $h$

# Height balanced trees

- Minimum size height-balanced trees




- General strategy to build a small balanced tree of height $h$
  - Smallest balanced tree of height $h-1$ as left subtree
  - Smallest balanced tree of height $h-2$ as right subtree

- $S(h)$, size of smallest height-balanced tree of height $h$
- Recurrence
  - $S(0) = 0$, $S(1) = 1$
  - $S(h) = 1 + S(h-1) + S(h-2)$
- Compare to Fibonacci sequence
  - $F(0) = 0$, $F(1) = 1$
  - $F(n) = F(n-1) + F(n-2)$
- $S(h)$ grows exponentially with $h$
- For size $n$, $h$ is $O(\log n)$

# Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`

# Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$

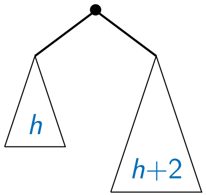
# Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$
- `t.insert(v)`, `t.delete(v)` can alter slope to $-2$ or $+2$

# Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$
- `t.insert(v)`, `t.delete(v)` can alter slope to $-2$ or $+2$

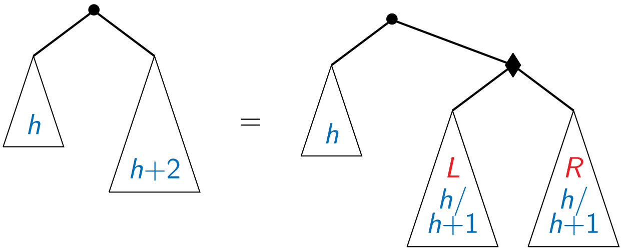
## Left rotation



# Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$
- `t.insert(v)`, `t.delete(v)` can alter slope to $-2$ or $+2$

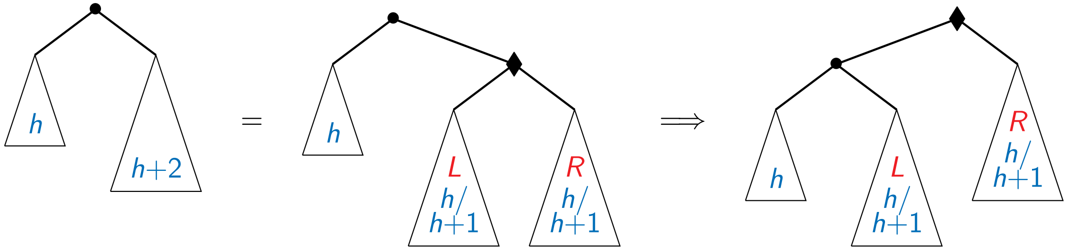
## Left rotation



# Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$
- `t.insert(v)`, `t.delete(v)` can alter slope to $-2$ or $+2$

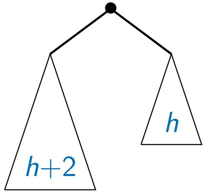
Left rotation — converts slope $-2$ to $\{0, 1, 2\}$



# Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$
- `t.insert(v)`, `t.delete(v)` can alter slope to $-2$ or $+2$

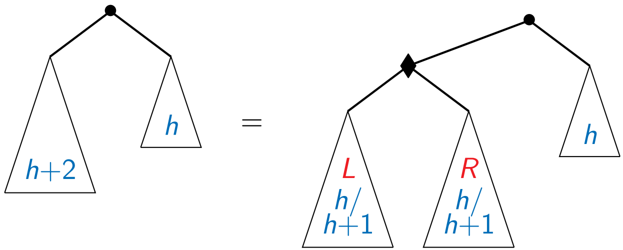
Right rotation



# Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$
- `t.insert(v)`, `t.delete(v)` can alter slope to $-2$ or $+2$

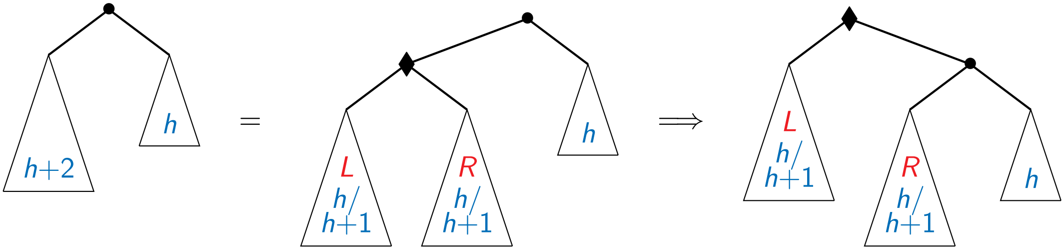
## Right rotation



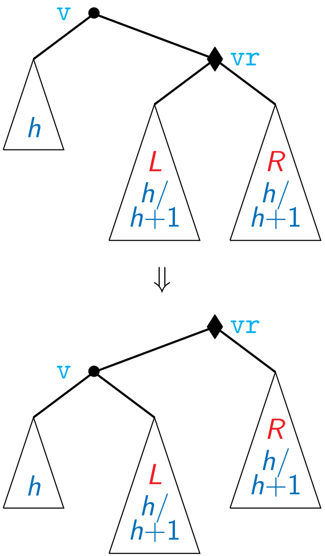
# Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$
- `t.insert(v)`, `t.delete(v)` can alter slope to $-2$ or $+2$

Right rotation — converts slope $+2$ to $\{-2, -1, 0\}$

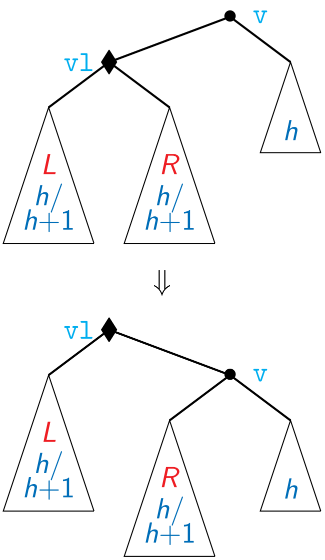


# Implementing rotations




```
class Tree:
  ...

  def leftrotate(self):
    v = self.value
    vr = self.right.value
    tl = self.left
    trl = self.right.left
    trr = self.right.right

    newleft = Tree(v)
    newleft.left = tl
    newleft.right = trl

    self.value = vr
    self.right = trr
    self.left = newleft
    return
```

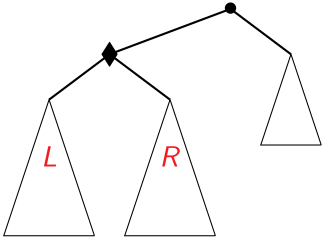
# Implementing rotations




```
class Tree:
  ...

  def rightrotate(self):
    v = self.value
    vl = self.left.value
    tll = self.left.left
    tlr = self.left.right
    tr = self.right

    newright = Tree(v)
    newright.left = tlr
    newright.right = tr

    self.value = vl
    self.left = tll
    self.right = newright
    return
```

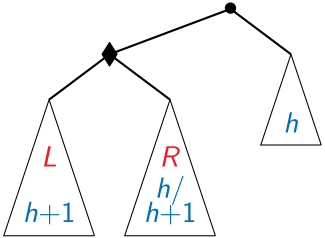
# Rebalancing, root has slope $+2$

- Rebalance bottom-up, assume subtrees are balanced



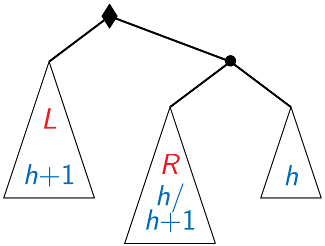
# Rebalancing, root has slope $+2$

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at ♦ is in $\{0, 1\}$



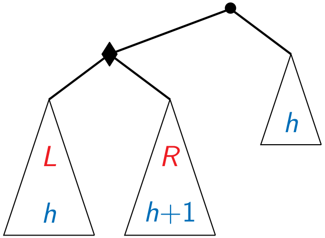
# Rebalancing, root has slope $+2$

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at ⧫ is in $\{0, 1\}$
  - Rotate right at •
  - All nodes are balanced



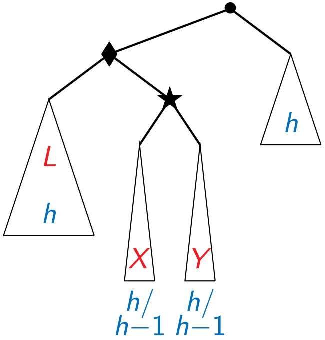
# Rebalancing, root has slope $+2$

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at ⧫ is in $\{0, 1\}$
  - Rotate right at •
  - All nodes are balanced
- Case 2: Slope at ⧫ is $-1$



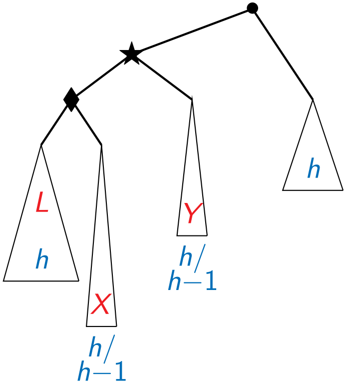
# Rebalancing, root has slope $+2$

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at ♦ is in $\{0, 1\}$
  - Rotate right at •
  - All nodes are balanced
- Case 2: Slope at ♦ is $-1$
  - Expand *R*



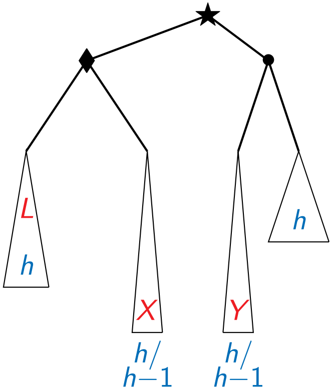
# Rebalancing, root has slope $+2$

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at ♦ is in $\{0, 1\}$
  - Rotate right at •
  - All nodes are balanced
- Case 2: Slope at ♦ is $-1$
  - Expand *R*
  - Rotate left at ♦



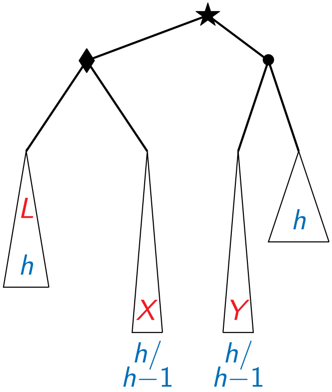
# Rebalancing, root has slope $+2$

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at ♦ is in $\{0, 1\}$
  - Rotate right at •
  - All nodes are balanced
- Case 2: Slope at ♦ is $-1$
  - Expand $R$
  - Rotate left at ♦
  - Rotate left at •



# Rebalancing, root has slope $+2$

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at ⧫ is in $\{0, 1\}$
  - Rotate right at •
  - All nodes are balanced
- Case 2: Slope at ⧫ is $-1$
  - Expand *R*
  - Rotate left at ⧫
  - Rotate left at •
- Rebalance with root slope $-2$ is symmetric



# Update insert() and delete()

- Use the rebalancing strategy to define a function `rebalance()`
- Rebalance each time the tree is modified
- Automatically rebalances bottom up

```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

        if v < self.value:
            self.left.insert(v)
            self.left.rebalance()
            return

        if v > self.value:
            self.right.insert(v)
            self.right.rebalance()
            return
```

# Update insert() and delete()

- Use the rebalancing strategy to define a function `rebalance()`
- Rebalance each time the tree is modified
- Automatically rebalances bottom up

```
class Tree:
    ...
    def delete(self,v):
        ...
        if v < self.value:
            self.left.delete(v)
            self.left.rebalance()
            return
        if v > self.value:
            self.right.delete(v)
            self.right.rebalance()
            return
        if v == self.value:
            if self.isleaf():
                self.makeempty()
            elif self.left.isempty():
                self.copyright()
            elif self.right.isempty():
                self.copyleft()
            else:
                self.value = self.left.maxval()
                self.left.delete(self.left.maxval())
            return
```

# Computing slope

- To compute the slope we need heights of subtrees
- But, computing height is $O(n)$

```
class Tree:
    ...
    def height(self):
        if self.isempty():
            return(0)
        else:
            return(1 +
                   max(self.left.height(),
                       self.right.height()))
```

# Computing slope

- To compute the slope we need heights of subtrees
- But, computing height is $O(n)$
- Instead, maintain a field `self.height`

```
class Tree:
    ...
    def height(self):
        if self.isempty():
            return(0)
        else:
            return(1 +
                   max(self.left.height(),
                       self.right.height())
```

# Computing slope

- To compute the slope we need heights of subtrees
- But, computing height is $O(n)$
- Instead, maintain a field `self.height`
- After each modification, update `self.height` based on `self.left.height`, `self.right.height`

```
class Tree:
    ...
    def insert(self,v):
        ...
        if v < self.value:
            self.left.insert(v)
            self.left.rebalance()
            self.height = 1 +
                          max(self.left.height,
                              self.right.height)
            return

        if v > self.value:
            self.right.insert(v)
            self.right.rebalance()
            self.height = 1 +
                          max(self.left.height,
                              self.right.height)
            return
```

# Summary

- Using rotations, we can maintain height balance
- Height balanced trees have height $O(\log n)$
- `find()`, `insert()` and `delete()` all walk down a single path, take time $O(\log n)$