Lecture 22, 7 November 2024

Madhavan Mukund

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

Programming and Data Structures with Python

Lecture 22, 07 Nov 2024

Dynamic sorted data

- Sorting is useful for efficient searching
- What if the data is changing dynamically?
  - Items are periodically inserted and deleted
- Insert/delete in a sorted list takes time $O(n)$

find(v) – sorted ⇒ log n
insert(v)
delete(v)

Heap – binary tree
– Filled level by level, L to R
– Node ≥ children

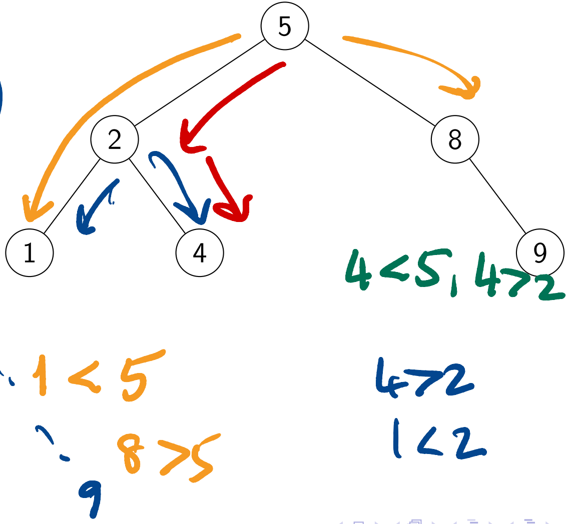
insert()
delete_max()

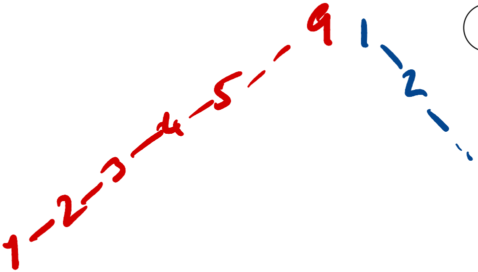
Dynamic sorted data

- Sorting is useful for efficient searching
- What if the data is changing dynamically?
  - Items are periodically inserted and deleted
- Insert/delete in a sorted list takes time $O(n)$
- Move to a tree structure, like heaps for priority queues

Binary search tree

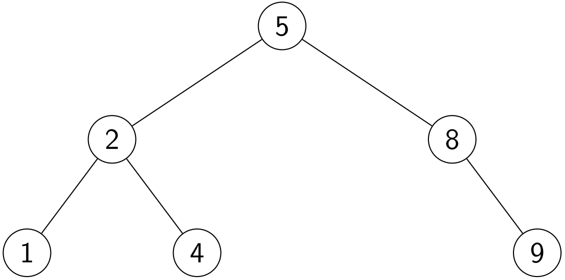
- For each node with value v
  - All values in the left subtree are $< v$
  - All values in the right subtree are $> v$



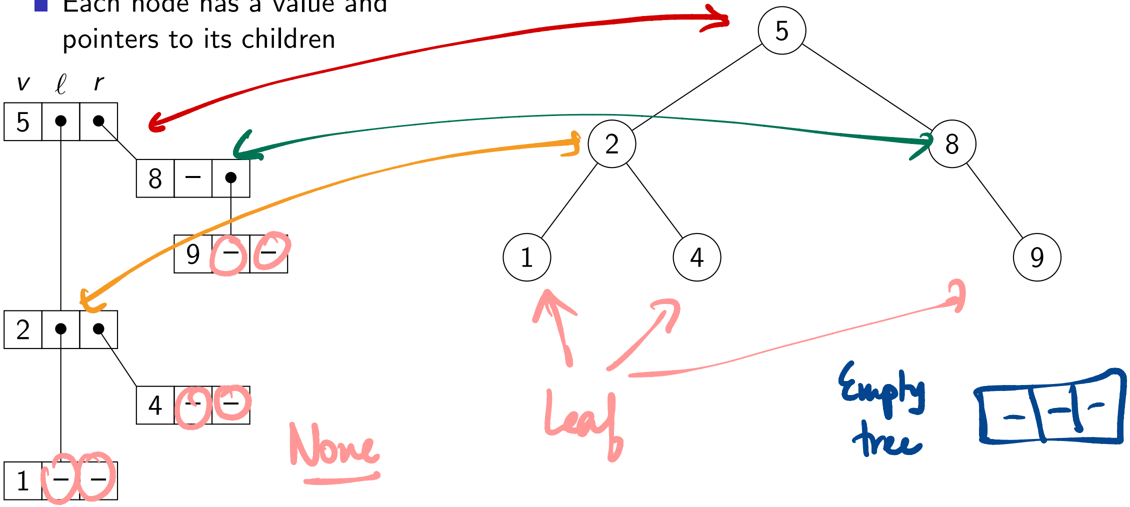


Binary search tree

- For each node with value v
  - All values in the left subtree are $< v$
  - All values in the right subtree are $> v$
- No duplicate values

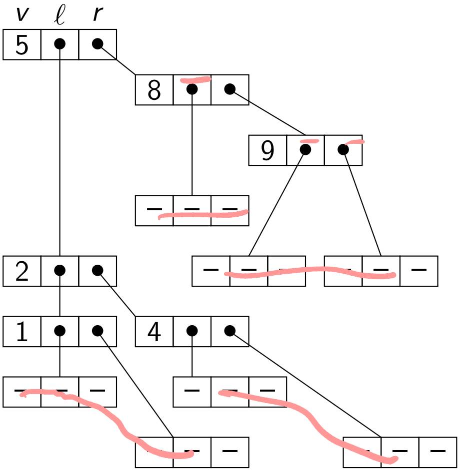


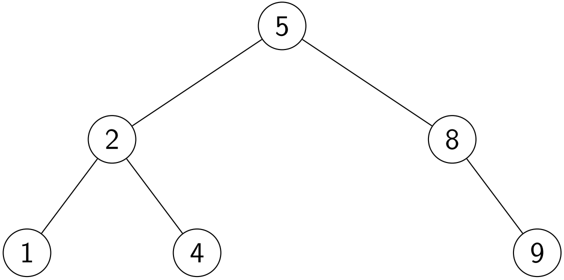
- Each node has a value and pointers to its children



Implementing a binary search tree

- Each node has a value and pointers to its children



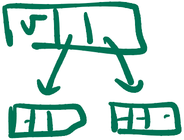


- Add a frontier with empty nodes, all fields –
  - Empty tree is single empty node
  - Leaf node points to empty nodes
- Easier to implement operations recursively

The class Tree

- Three local fields, `value`, `left`, `right`
- Value `None` for empty value –
- Empty true has all fields `None`
- Leaf has a nonempty `value` and empty `left` and `right`

```
class Tree:

    # Constructor:
    def __init__(self,initval=None):
        self.value = initval
        if self.value != None:
            self.left = Tree()
            self.right = Tree()
        else:
            self.left = None
            self.right = None
        return

    # Only empty node has value None
    def isempty(self):
        return (self.value == None)

    # Leaf nodes have both children empty
    def isleaf(self):
        return (self.value != None and
                self.left.isempty() and
                self.right.isempty())
```



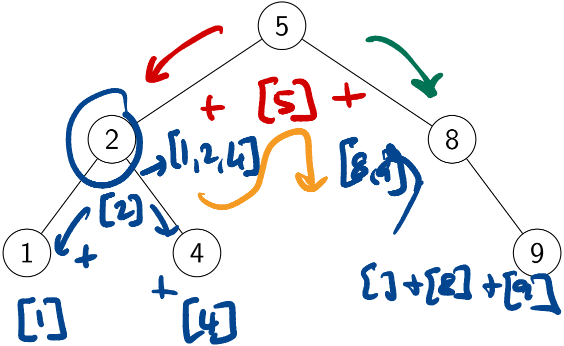
Inorder traversal

- List the left subtree, then the current node, then the right subtree
- Lists values in sorted order
- Use to print the tree

```
class Tree:
    ...
    # Inorder traversal
    def inorder(self):
        if self.isempty():
            return([])
        else:
            return(self.left.inorder()+
                   [self.value]+
                   self.right.inorder())

    # Display Tree as a string
    def __str__(self):
        return(str(self.inorder()))
```

Inorder traversal

- List the left subtree, then the current node, then the right subtree
- Lists values in sorted order
- Use to print the tree




```
class Tree:
  ...
  # Inorder traversal
  def inorder(self):
    if self.isempty():
      return([])
    else:
      return(self.left.inorder()+
             [self.value]+
             self.right.inorder())

  # Display Tree as a string
  def __str__(self):
    return(str(self.inorder()))
```

[1,2,4]+[5]+[8,9]

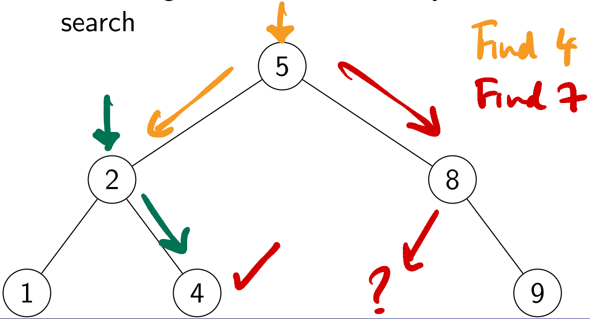
Find a value v

- Check value at current node
- If v smaller than current node, go left
- If v smaller than current node, go right
- Natural generalization of binary search

```
class Tree:
    ...
    # Check if value v occurs in tree
    def find(self,v):
        if self.isempty():
            return(False)

        if self.value == v:
            return(True)

        if v < self.value:
            return(self.left.find(v))

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

Find a value v

- Check value at current node
- If v smaller than current node, go left
- If v smaller than current node, go right
- Natural generalization of binary search



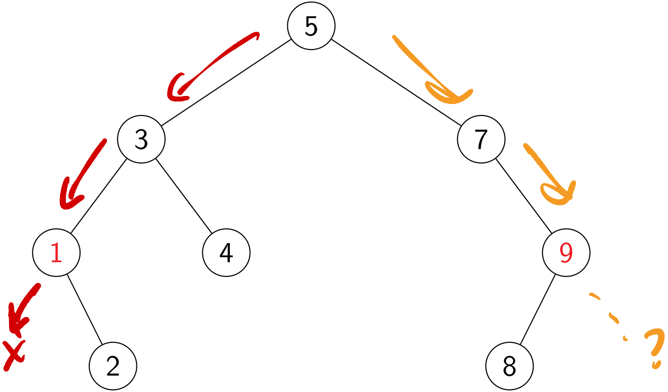

```
class Tree:
    ...
    # Check if value v occurs in tree
    def find(self,v):
        if self.isempty():
            return(False)

        if self.value == v:
            return(True)

        if v < self.value:
            return(self.left.find(v))

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

Minimum and maximum

- Minimum is left most node in the tree
- Maximum is right most node in the tree

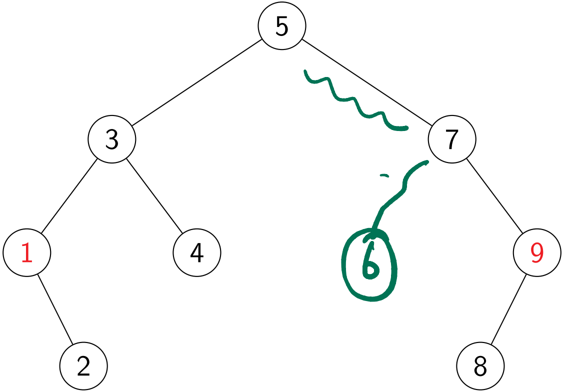
Minimum and maximum

- Minimum is left most node in the tree
- Maximum is right most node in the tree



Minimum and maximum

- Minimum is left most node in the tree
- Maximum is right most node in the tree




```
class Tree:
    ...
    def minval(self):
        if self.left.isempty():
            return(self.value)
        else:
            return(self.left.minval())


    def maxval(self):
        if self.right.isempty():
            return(self.value)
        else:
            return(self.right.maxval())
```

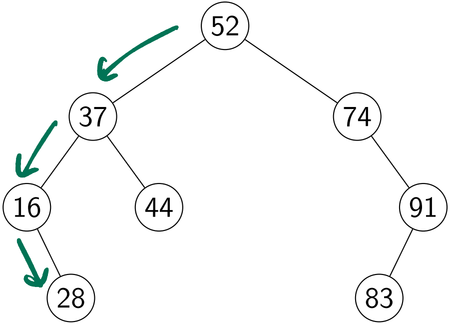
Insert a value `v`

- Try to find `v`
- Insert at the position where `find` fails

Insert a value v

- Try to find `v`
- Insert at the position where `find` fails

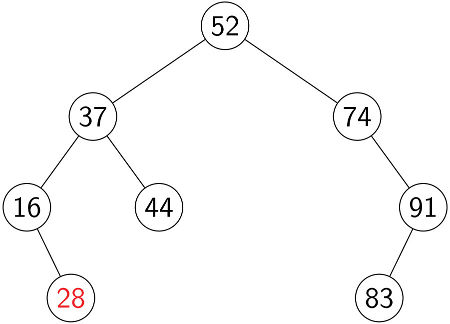
Insert 21



Insert a value v

- Try to find `v`
- Insert at the position where `find` fails

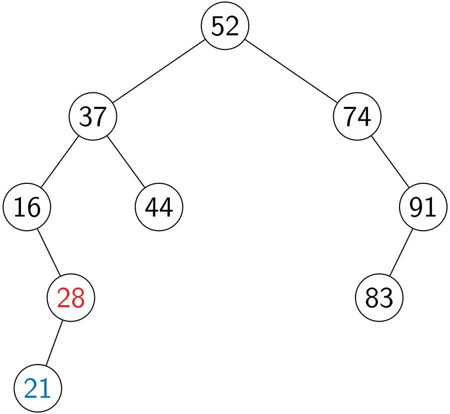
Insert 21



Insert a value v

- Try to find v
- Insert at the position where find fails

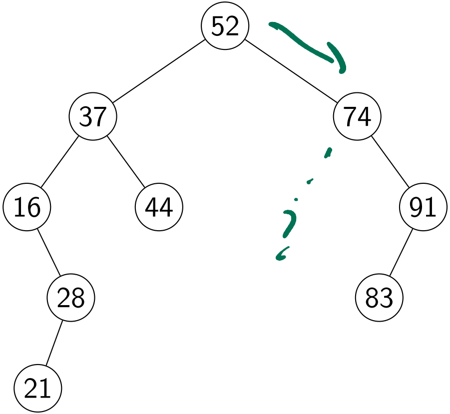
Insert 21



Insert a value v

- Try to find v
- Insert at the position where find fails

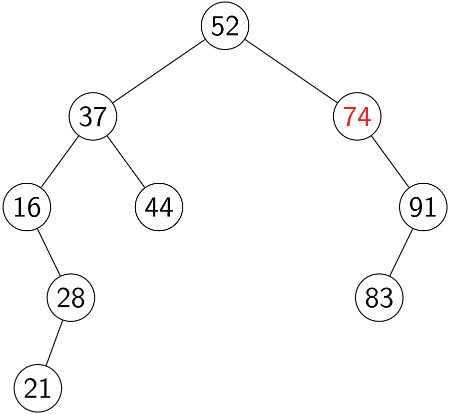
Insert 65



Insert a value v

- Try to find v
- Insert at the position where find fails

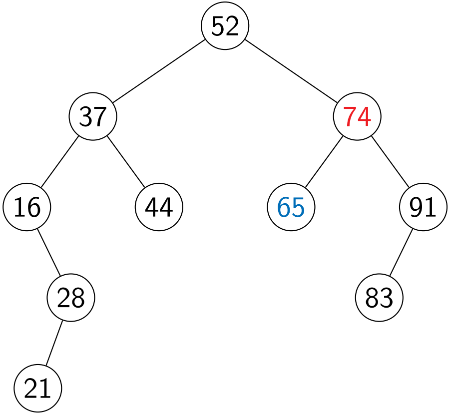
Insert 65



Insert a value v

- Try to find v
- Insert at the position where find fails

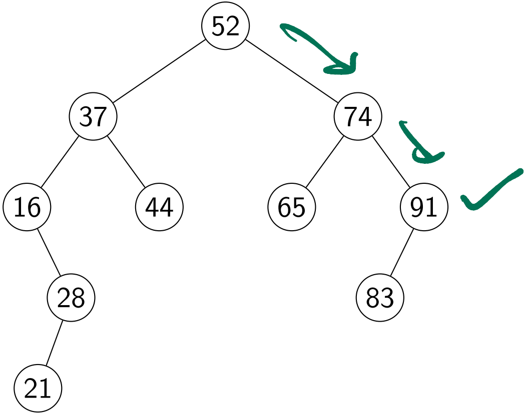
Insert 65



Insert a value v

- Try to find v
- Insert at the position where find fails

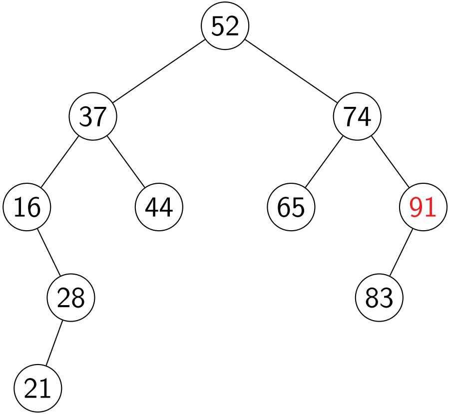
Insert 91



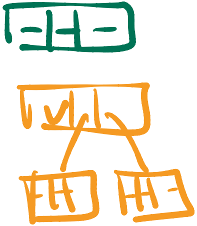
Insert a value v

- Try to find `v`
- Insert at the position where `find` fails

Insert 91

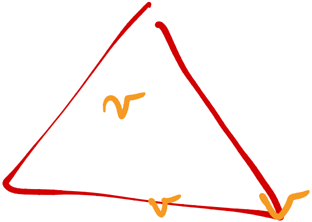



```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

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

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



Delete a value v

- If v is present, delete

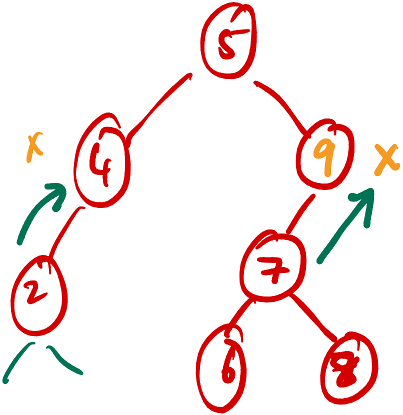


Delete a value `v`

- If `v` is present, delete
- Leaf node? No problem

Delete a value v

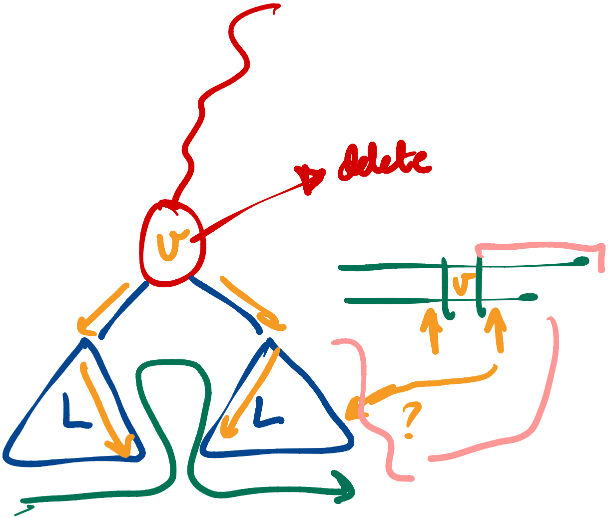
- If v is present, delete
- Leaf node? No problem
- If only one child, promote that subtree



Delete a value v

- If v is present, delete
- Leaf node? No problem
- If only one child, promote that subtree

Otherwise?



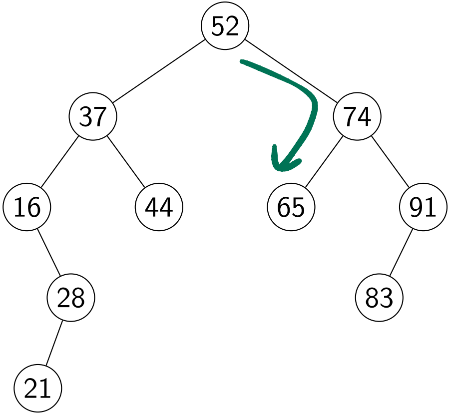
Delete a value v

- If v is present, delete
- Leaf node? No problem
- If only one child, promote that subtree
- Otherwise, replace v with `self.left.maxval()` and delete `self.left.maxval()`
  - `self.left.maxval()` has no right child

```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            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
```

Delete a value v

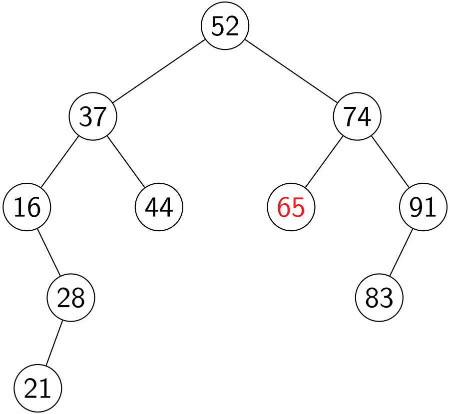
Delete 65




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            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
```

Delete a value v

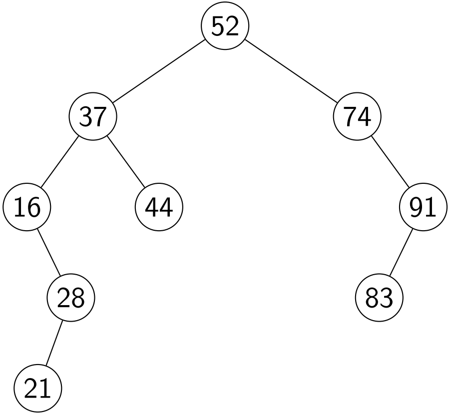
Delete 65




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            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
```

Delete a value v

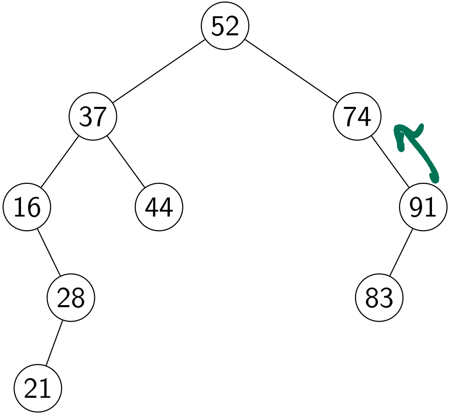
Delete 65




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            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
```

Delete a value v

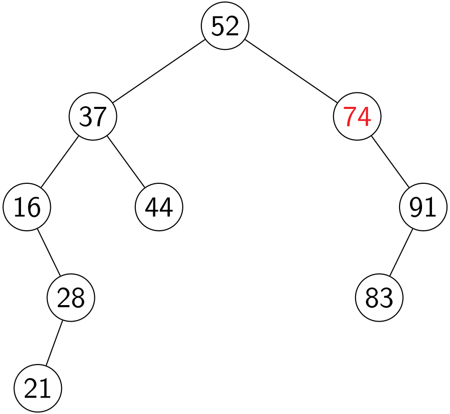
Delete 74




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            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
```

Delete a value v

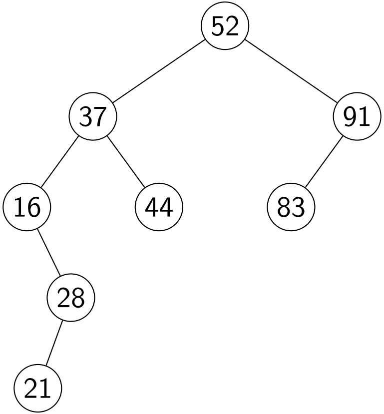
Delete 74




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            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
```

Delete a value v

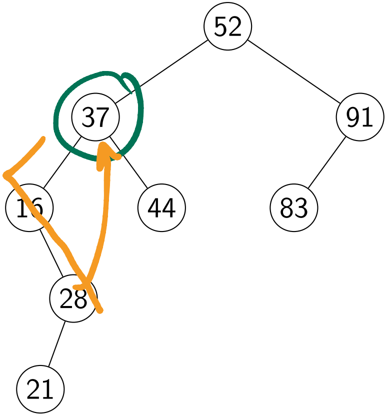
Delete 74




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            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
```

Delete a value v

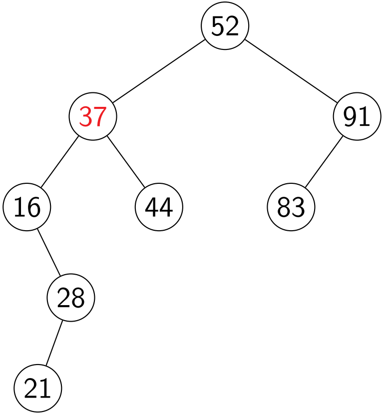
Delete 37




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            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
```

Delete a value v

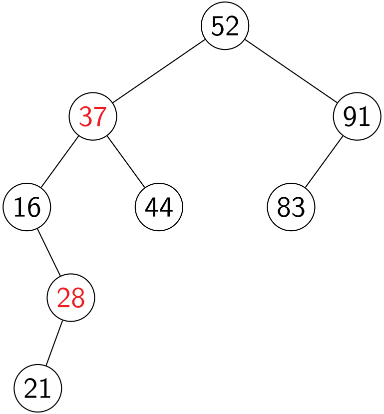
Delete 37




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            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
```

Delete a value v

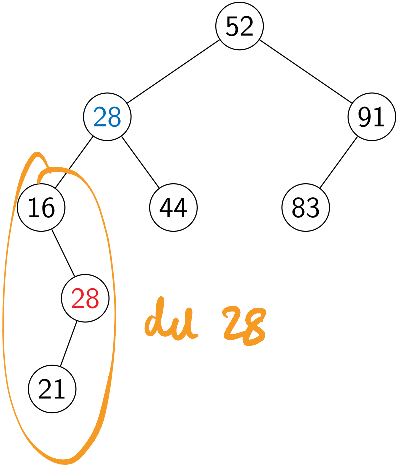
Delete 37




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            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
```

Delete a value v

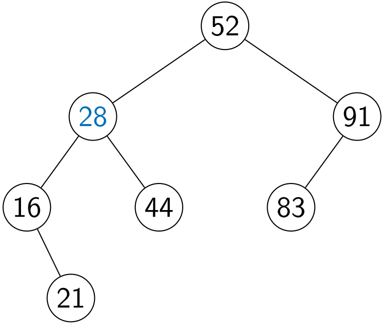
Delete 37




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            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
```

Delete a value v

Delete 37




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            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
```

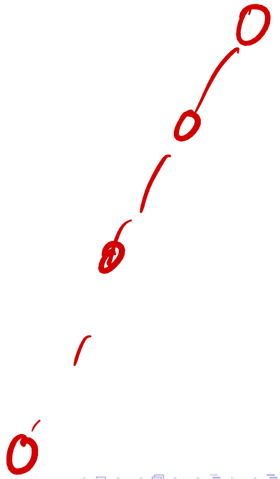
Delete a value v

```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            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
```

```
# Convert leaf node to empty node
def makeempty(self):
    self.value = None
    self.left = None
    self.right = None
    return

# Promote left child
def copyleft(self):
    self.value = self.left.value
    self.right = self.left.right
    self.left = self.left.left
    return

# Promote right child
def copyright(self):
    self.value = self.right.value
    self.left = self.right.left
    self.right = self.right.right
    return
```

Complexity

- `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?



Operations on search trees

Defining balance

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

Operations on search trees

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

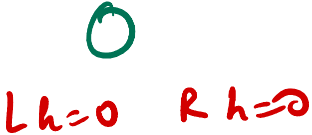
Operations on search trees

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

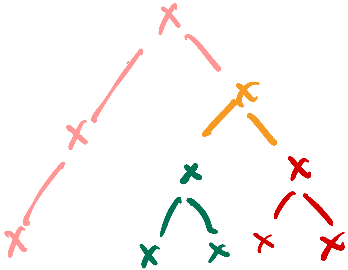
Height balanced trees

- `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



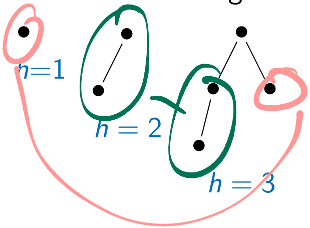
Height balanced trees

- `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?

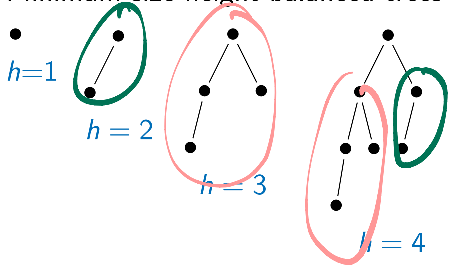
- Minimum size height-balanced trees



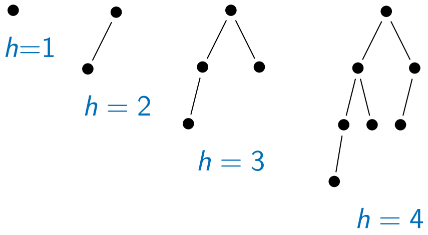
Height balanced trees

- `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



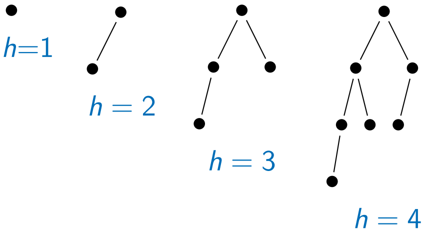
- `self.height()` — number of nodes on longest path from root to leaf
  - 0 for empty tree
  - 1 for tree with only a root node
  - $1 + \max$ of heights of left and right subtrees, in general
- Height balance
  - `self.left.height()` and `self.right.height()` differ by at most 1
  - AVL trees — Adelson-Velskii,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

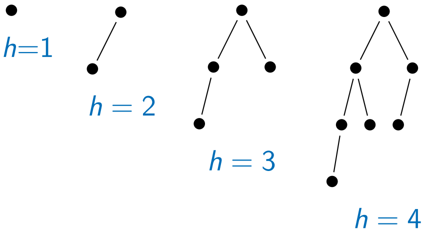
- Minimum size height-balanced trees



- 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

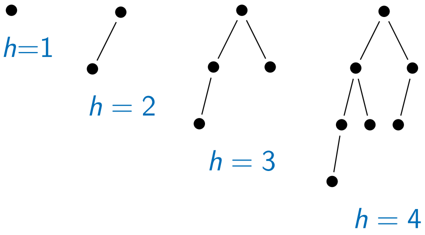


- General strategy to build a small balanced tree of height h
  - Smallest balanced tree of height $h-1$ as left subtree
  - Smallest balanced tree of height $h-2$ as right subtree

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

Height balanced trees

- Minimum size height-balanced trees

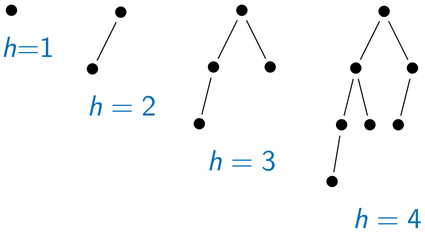


- General strategy to build a small balanced tree of height h
  - Smallest balanced tree of height $h-1$ as left subtree
  - Smallest balanced tree of height $h-2$ as right subtree

- $S(h)$, 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

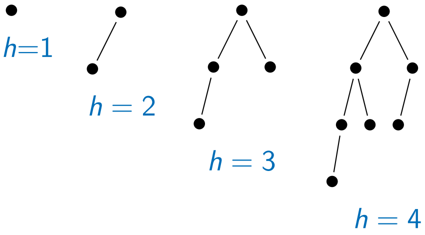


- General strategy to build a small balanced tree of height h
  - Smallest balanced tree of height $h-1$ as left subtree
  - Smallest balanced tree of height $h-2$ as right subtree

- $S(h)$, 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

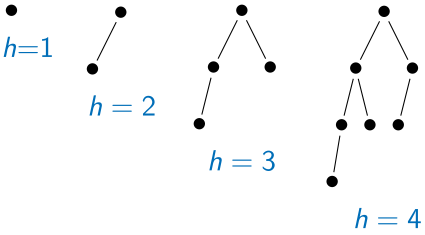


- General strategy to build a small balanced tree of height h
  - Smallest balanced tree of height $h-1$ as left subtree
  - Smallest balanced tree of height $h-2$ as right subtree

- $S(h)$, 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

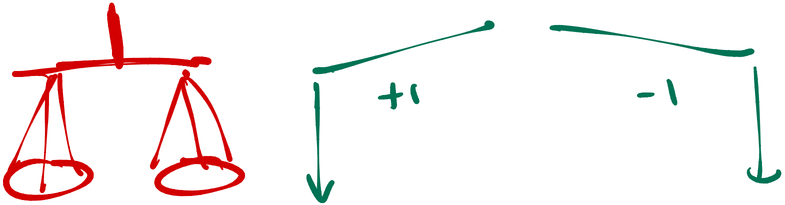


- General strategy to build a small balanced tree of height h
  - Smallest balanced tree of height $h-1$ as left subtree
  - Smallest balanced tree of height $h-2$ as right subtree

- $S(h)$, 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 $\{-1, 0, 1\}$

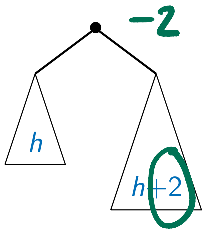
Correcting imbalance

- 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 $+2$

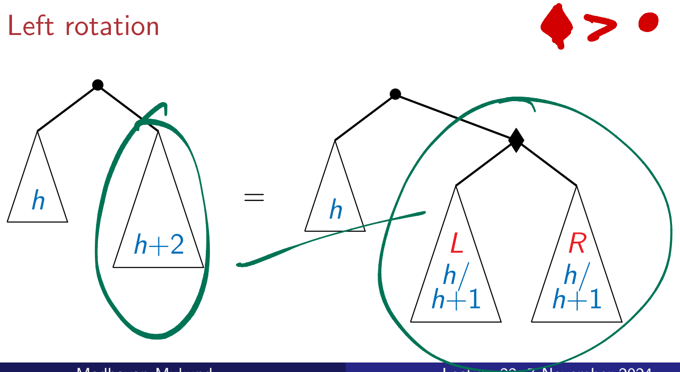
Left rotation



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 $+2$

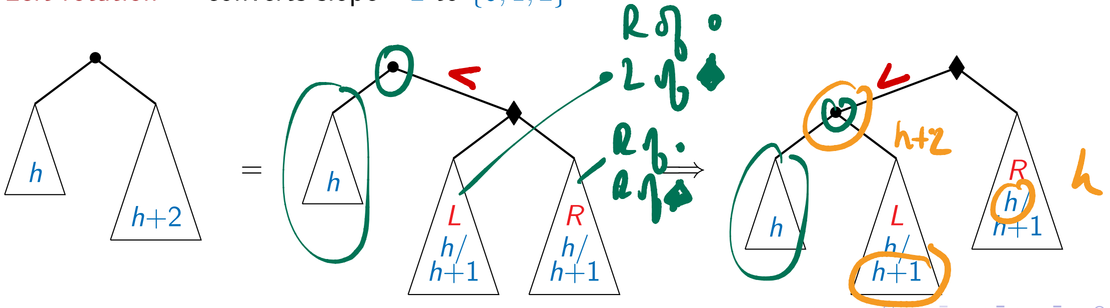
Left rotation



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 $+2$

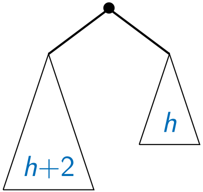
Left rotation — converts slope -2 to $\{0, 1, 2\}$



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 $+2$

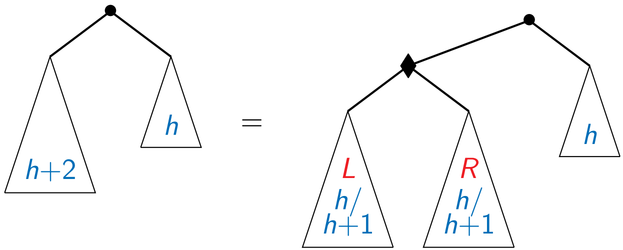
Right rotation



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 $+2$

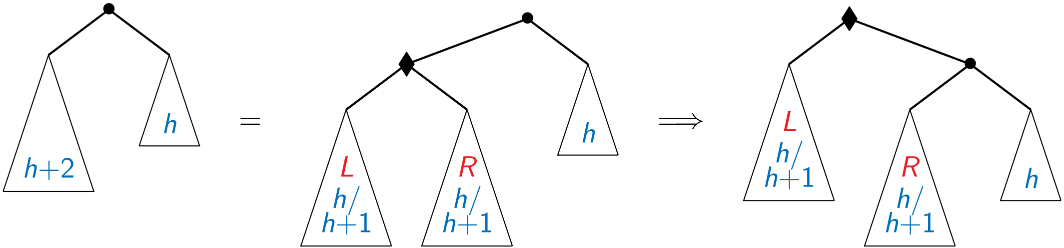
Right rotation



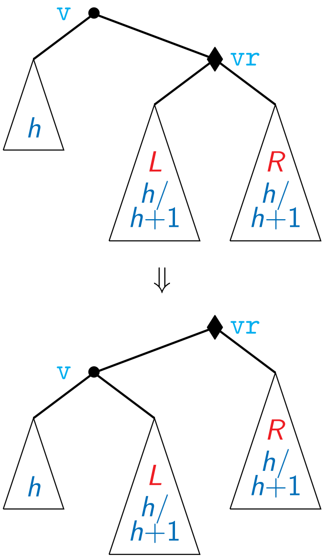
Correcting imbalance

- 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 $\{-2, -1, 0\}$

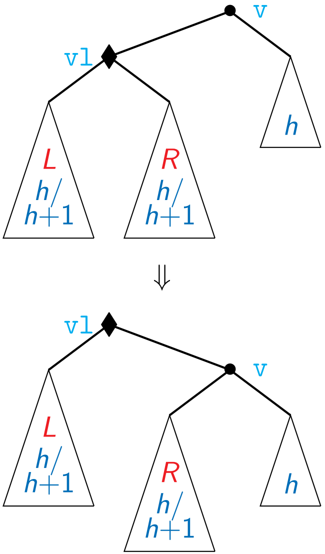


Implementing rotations




```
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.left = newleft
    self.right = trr

    return
```

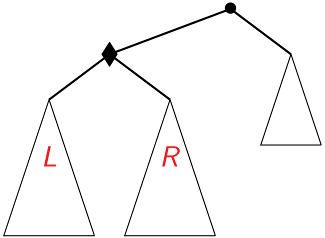
Implementing rotations




```
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
```

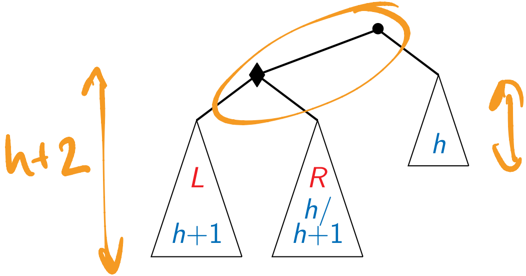
Rebalancing, root has slope +2

- Rebalance bottom-up, assume subtrees are balanced



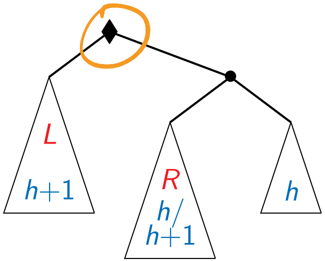
Rebalancing, root has slope +2

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



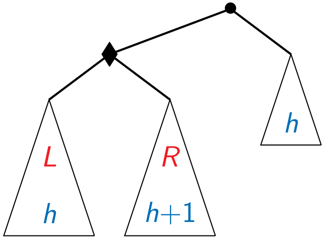
Rebalancing, root has slope $+2$

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



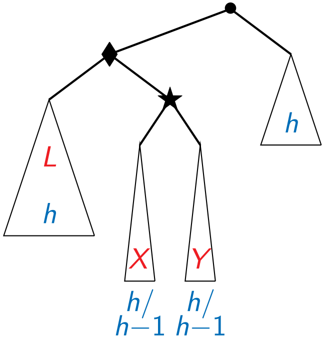
Rebalancing, root has slope $+2$

- 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



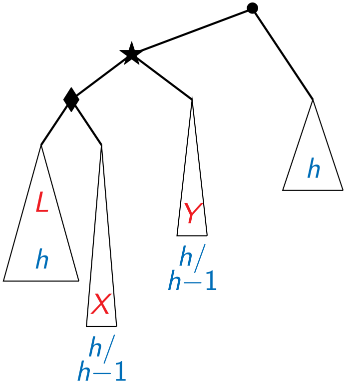
Rebalancing, root has slope $+2$

- 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*



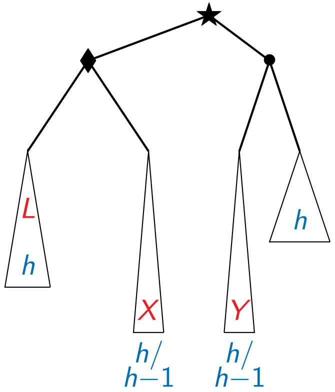
Rebalancing, root has slope $+2$

- 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 ♦



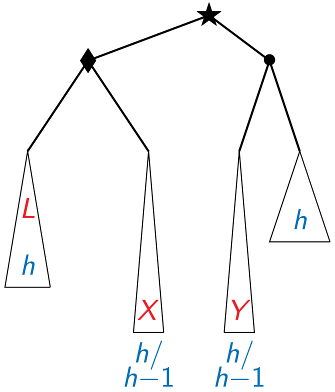
Rebalancing, root has slope $+2$

- 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 $O(\log n)$