Search Trees

Madhavan Mukund

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

13 December, 2021

Dynamic sorted data

- Sorting is useful for efficient searching

Dynamic sorted data

- Sorting is useful for efficient searching
- What if the data is changing dynamically?
  - Items are periodically inserted and deleted

Dynamic sorted data

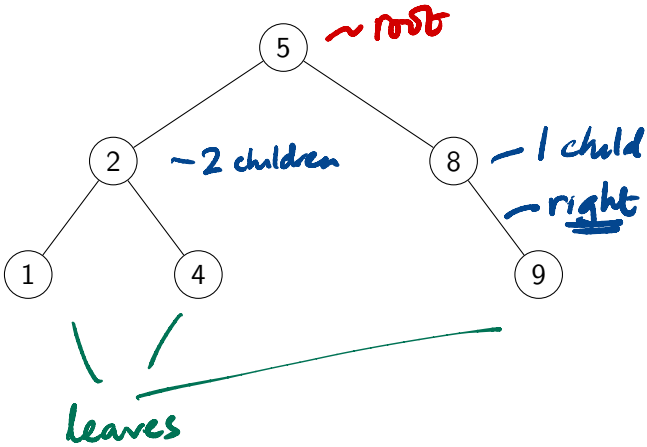
- Sorting is useful for efficient searching
- What if the data is changing dynamically?
  - Items are periodically inserted and deleted
- Insert/delete in a sorted list takes time $O(n)$

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

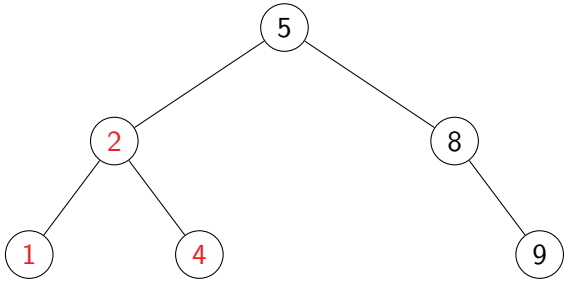
Binary search tree

- For each node with value v



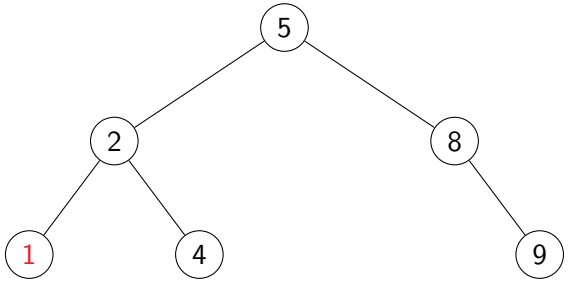
Binary search tree

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



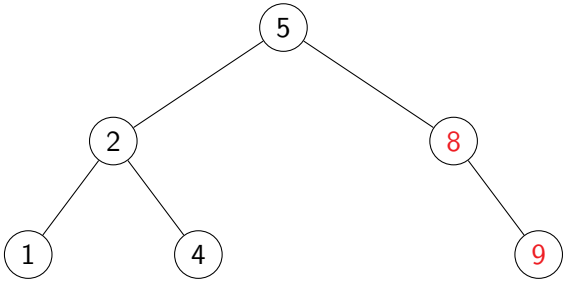
Binary search tree

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



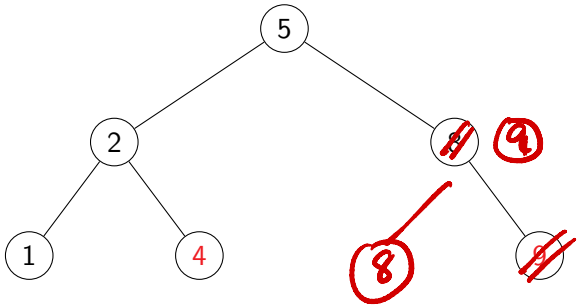
Binary search tree

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



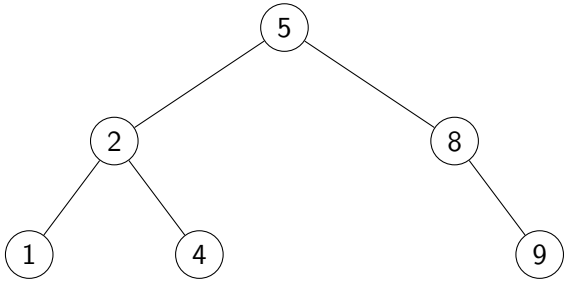
Binary search tree

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



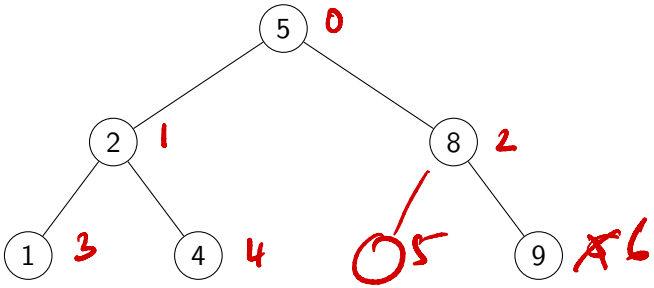
Binary search tree

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



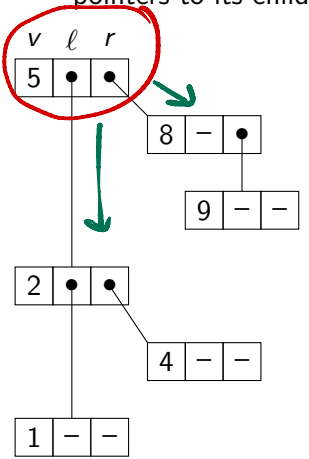
Implementing a binary search tree

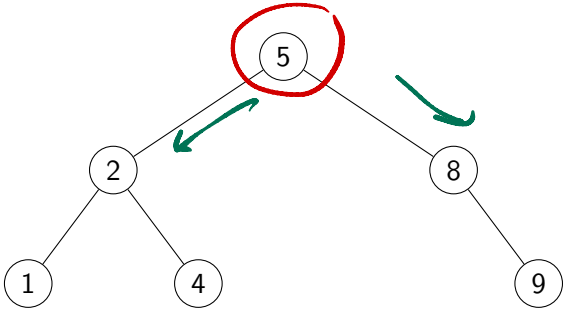
- Each node has a value and pointers to its children



Implementing a binary search tree

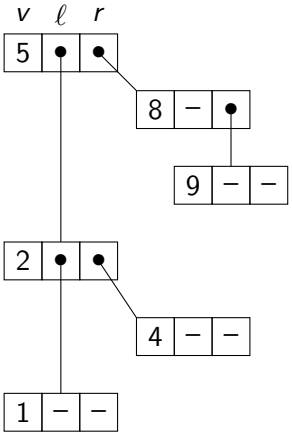
- Each node has a value and pointers to its children



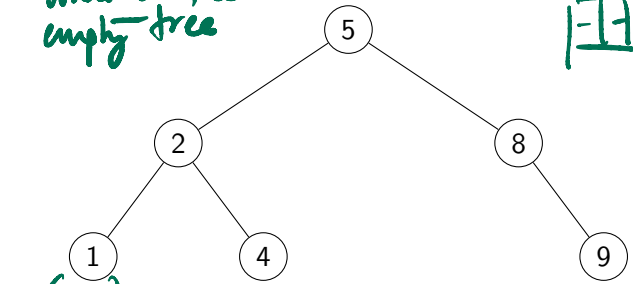


Implementing a binary search tree

- Each node has a value and pointers to its children



What is the empty tree

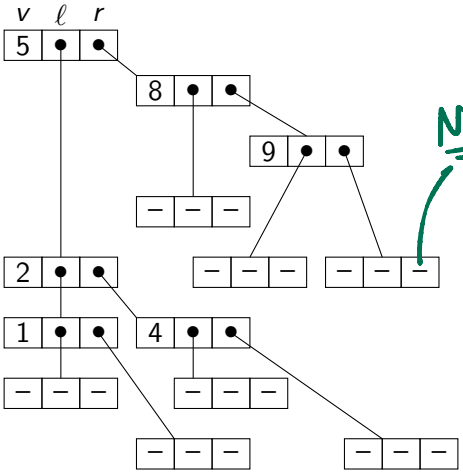


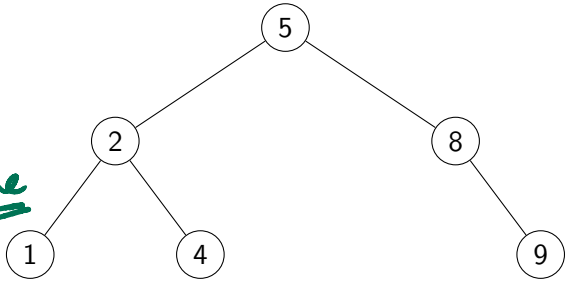
Empty Empty

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

Implementing a binary search tree

- Each node has a value and pointers to its children

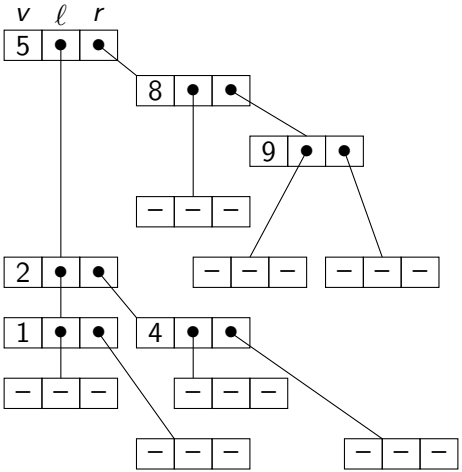


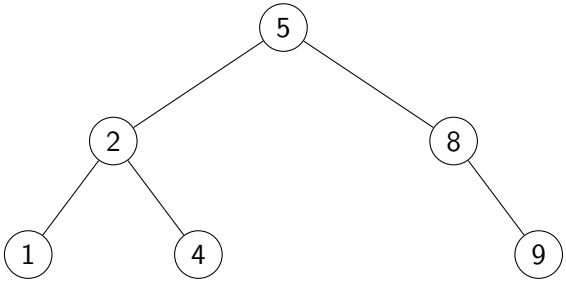


- Add a frontier with empty nodes, all fields –
  - Empty tree is single empty node
  - Leaf node points to empty nodes

Implementing a binary search tree

- Each node has a value and pointers to its children



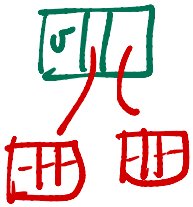


- Add a frontier with empty nodes, all fields –
  - Empty tree is single empty node
  - Leaf node points to empty nodes
- Easier to implement operations recursively

The class Tree

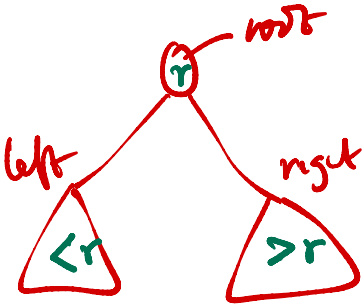
- Three local fields, `value`, `left`, `right`
- Value `None` for empty value –
- Empty true has all fields `None`
- Leaf has a nonempty `value` and empty `left` and `right`

```
class Tree:

    # Constructor:
    def __init__(self,initval=None):
        self.value = initval
        if self.value:
            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()))
```

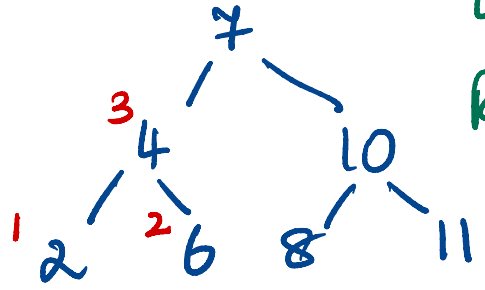
Other traversals

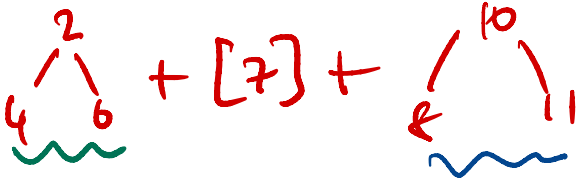
Post-order

Left subtree
Right subtree
Root

Pre Order

Root
Left
Right



Inorder  + [7] + ⇝ (2) + [4] + (6) + [7] (8) + [10] + (11)

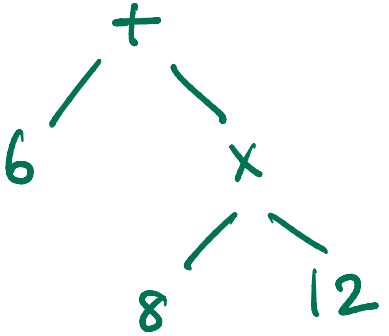
Postorder [2, 6, 4] + [8, 11, 10] + [7]

Preorder [7] + [4, 2, 6] + [10, 8, 11]

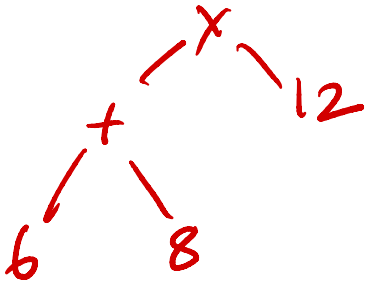
Arithmetic Expressions

$6 + 8 \times 12 =$

$6 + (8 \times 12)$



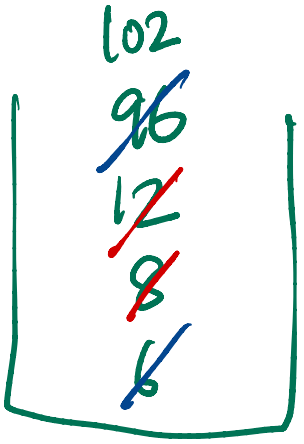
The "wrong" expr is



6 8 + 12 x

↳ Postfix 6 8 12 x +

6 8 12 × + → Evaluate



Each no. goes on stack

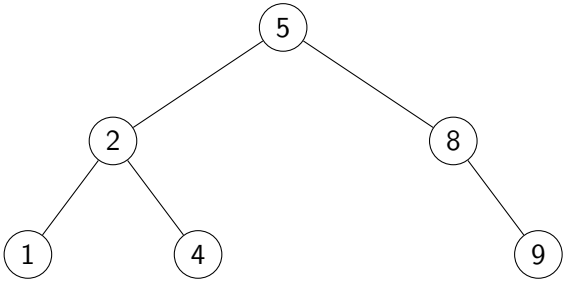
Each operation pops its operands & pushes result

6 8 + 12 ×

168
12
14
8
6

Inorder traversal

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



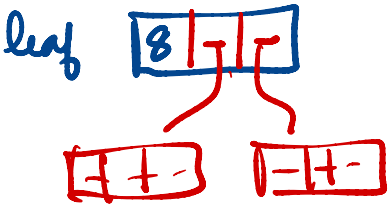

```
class Tree:
    ...
    # Inorder traversal
    def inorder(self):
        if self.isempty():
            return([])
        else:
            return(self.left.inorder()+
                   [self.value]+
                   self.right.inorder())

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

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

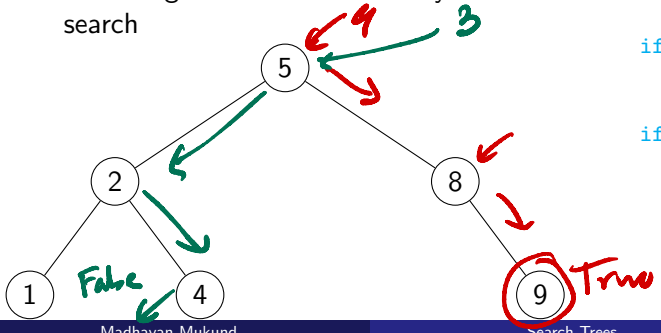
```
class Tree:
    ...
    # Check if value v occurs in tree
    def find(self,v):
        if self.isempty():
            return(False)

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

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

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

recursive calls



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))
```

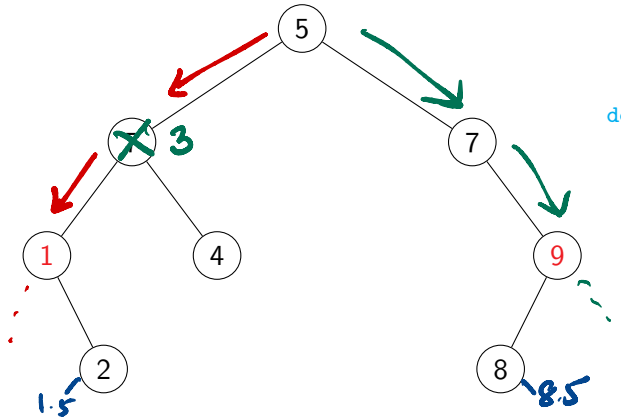
Minimum and maximum

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

```
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())
```

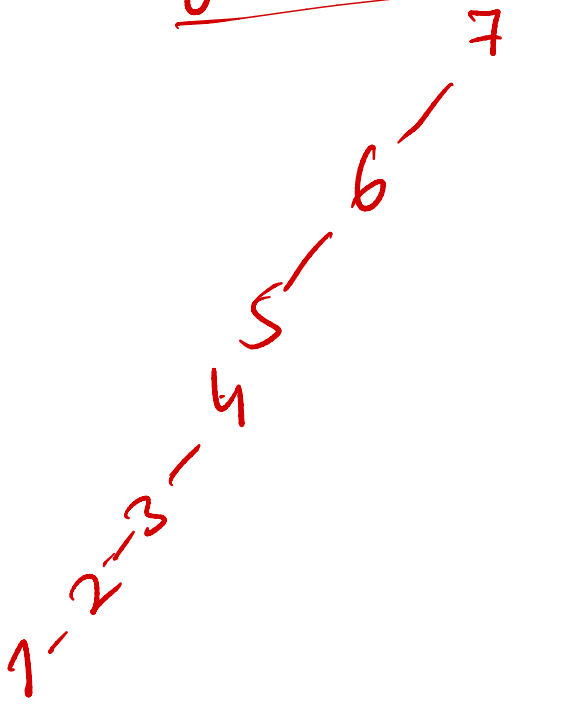
Minimum and maximum

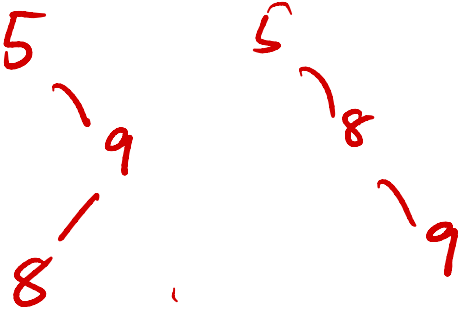
- Minimum is left most node in the tree
- Maximum is right most node in the tree

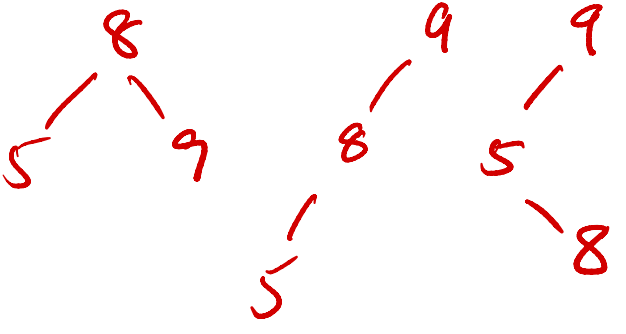



```
class Tree:
    ...
    def minval(self):
        if self.left.isempty():
            return(self.value)
        else:
            return(self.left.minval())


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

Degenerate trees







Insert a value v

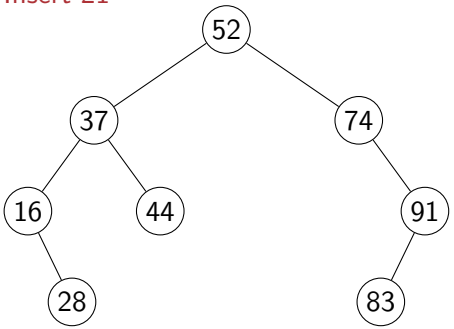
- Try to find `v`
- Insert at the position where `find` fails

```
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
```

Insert a value v

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

Insert 21

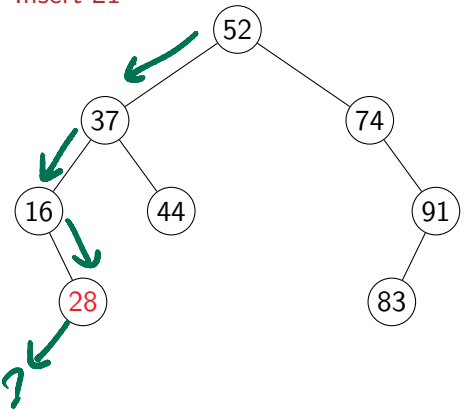



```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

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

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

Insert a value v

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

Insert 21

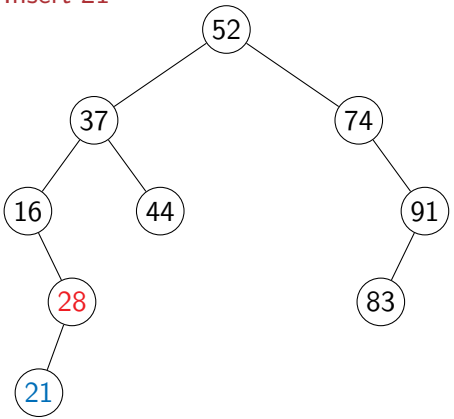



```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

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

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

Insert a value v

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

Insert 21




```
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
```

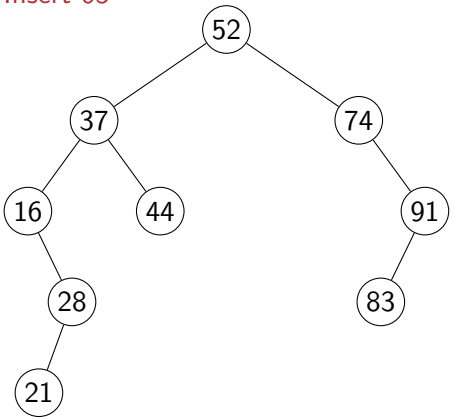
find

ret(False)

Insert a value v

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

Insert 65

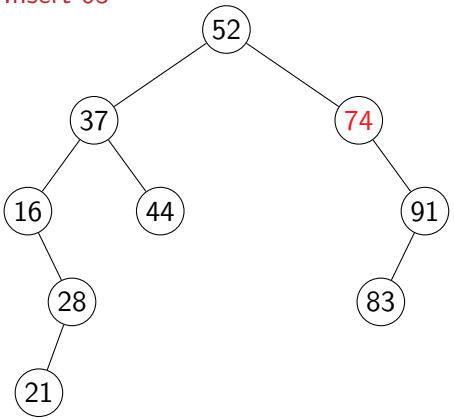



```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

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

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

Insert a value v

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

Insert 65

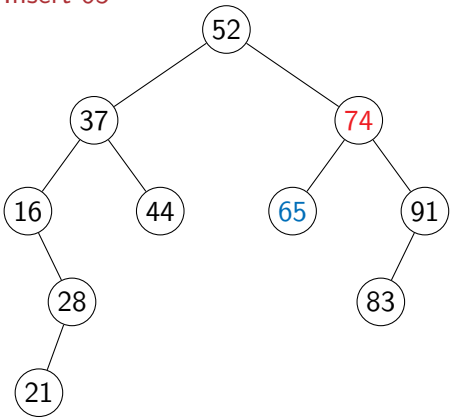



```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

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

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

Insert a value v

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

Insert 65

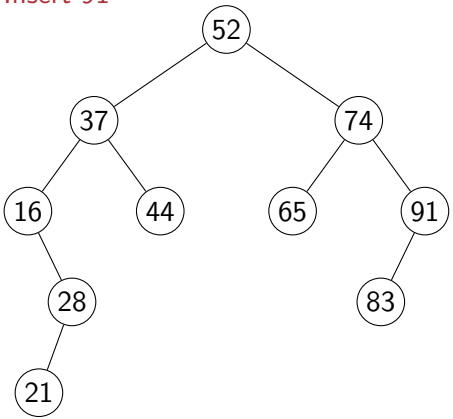



```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

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

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

Insert a value v

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

Insert 91

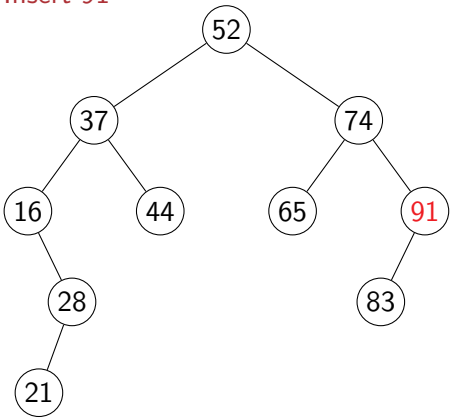



```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

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

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

Insert a value v

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

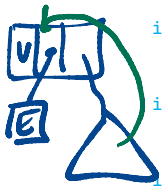
Insert 91



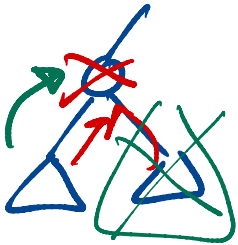

```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        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

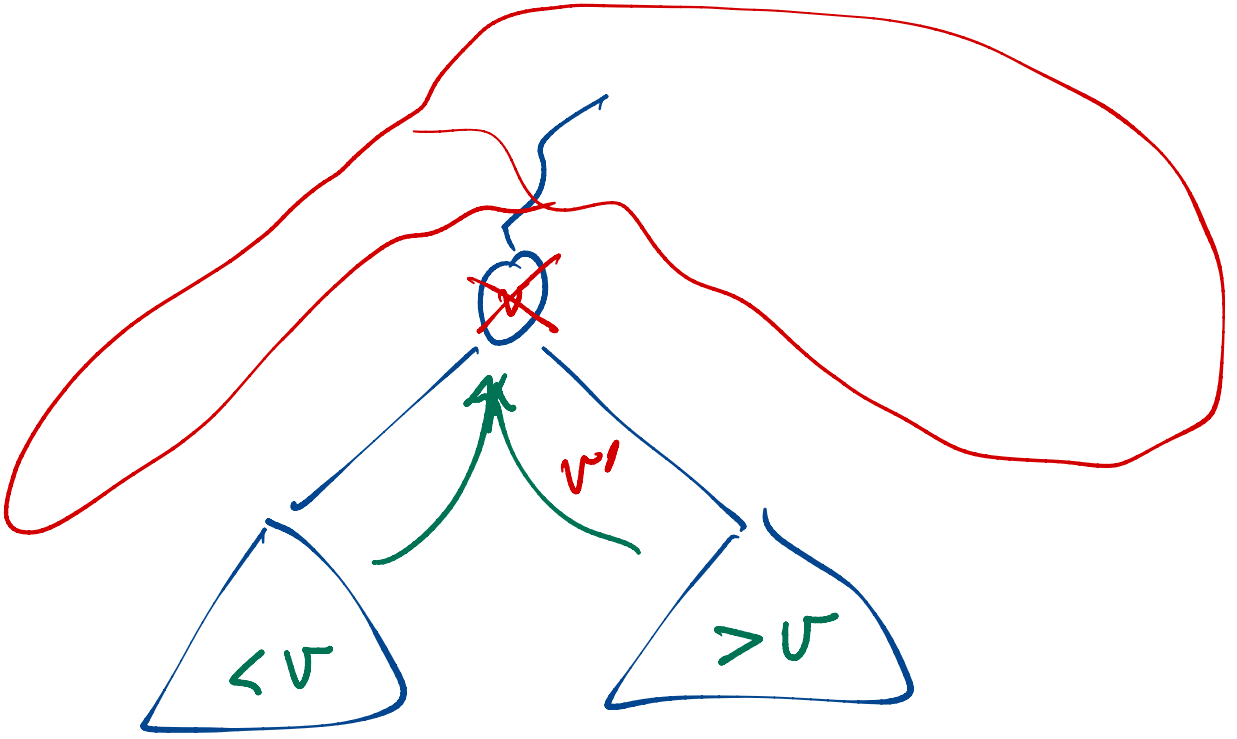
- If `v` is present, delete
- Leaf node? No problem
- If only one child, promote that subtree
- Otherwise, replace `v` with `self.left.maxval()` and delete `self.left.maxval()`
  - `self.left.maxval()` has no right child



```
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
```



Only one child



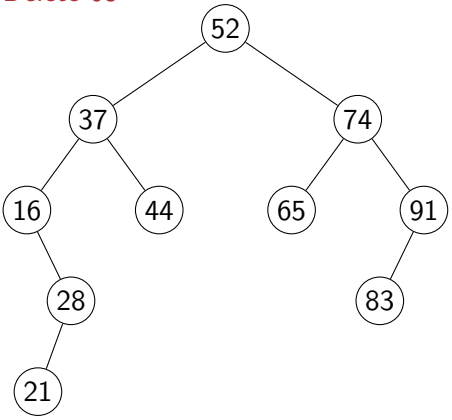
Want v', s.t $< v'$ $> v'$

If v' from left,
max of left subtree

If v' from right
min of right tree

Delete a value v

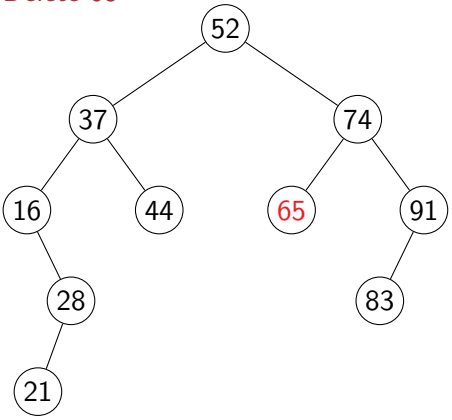
Delete 65




```
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

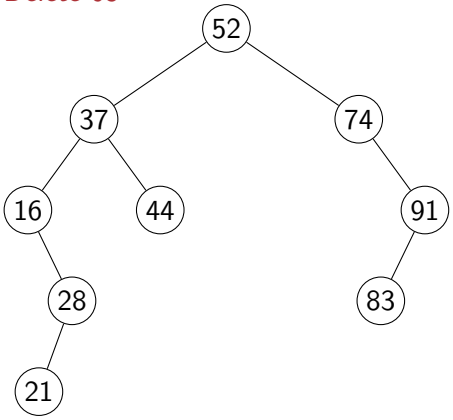
Delete 65




```
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

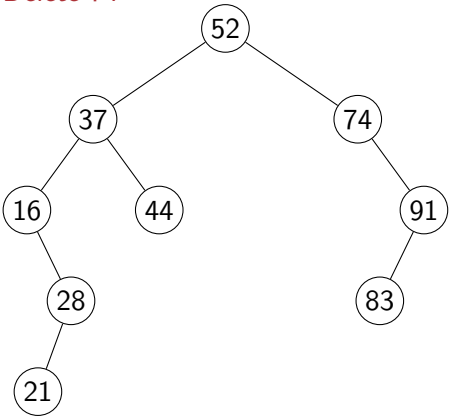
Delete 65




```
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

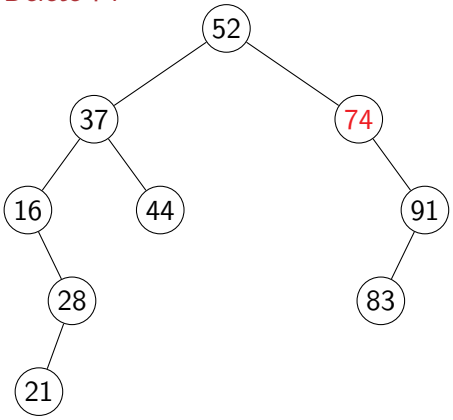
Delete 74




```
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

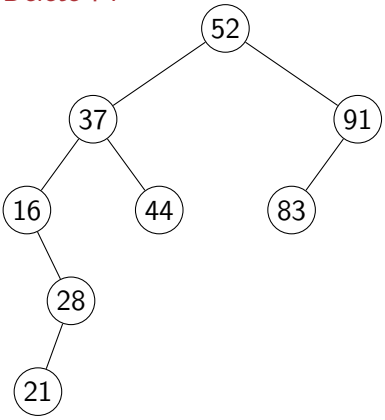
Delete 74




```
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

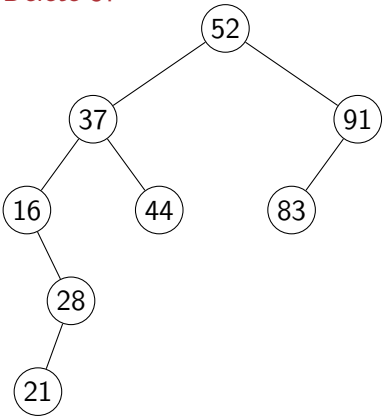
Delete 74




```
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

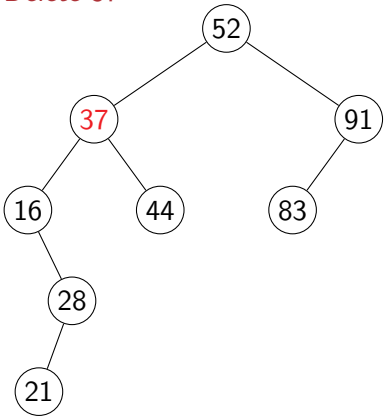
Delete 37




```
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

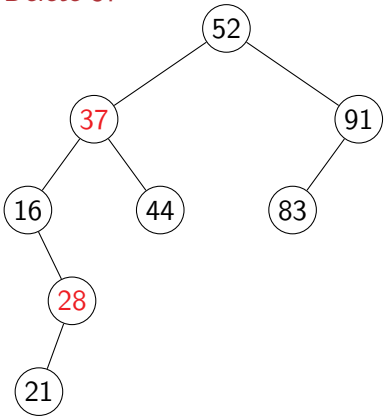
Delete 37




```
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

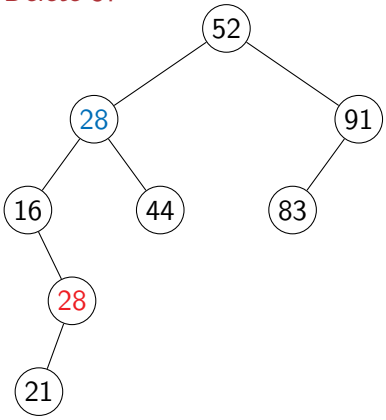
Delete 37




```
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

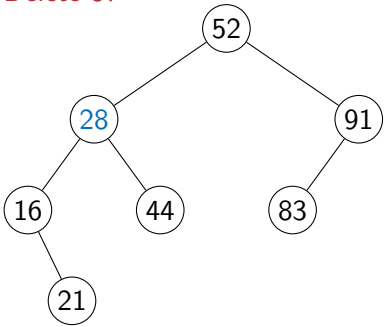
Delete 37




```
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

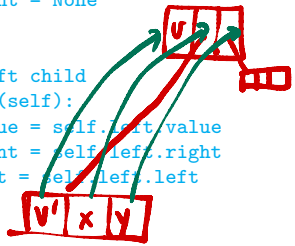



```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            return
        if v == self.value:
            if self.isleaf():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

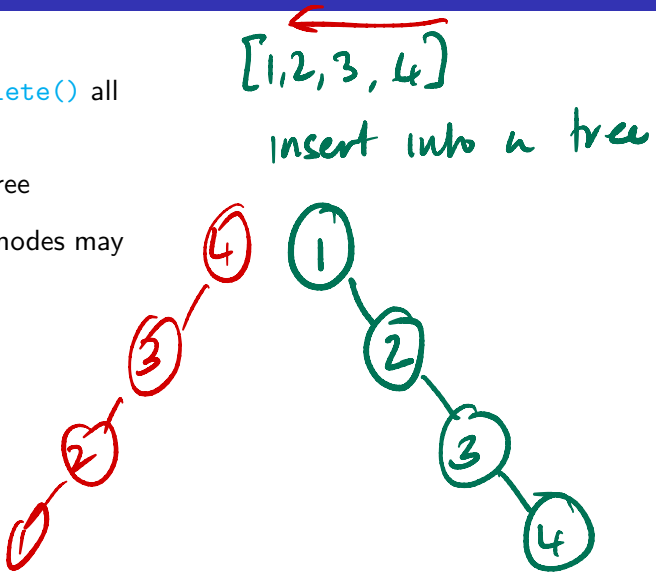


Complexity

- `find()`, `insert()` and `delete()` all walk down a single path
- Worst-case: height of the tree

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 $O(n)$



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

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

Defining balance

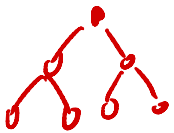
- Left and right subtrees should be "equal"
  - Two possible measures: `size` and `height`

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

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



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

Defining balance

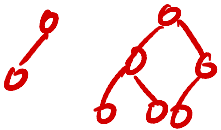
- Left and right subtrees should be "equal"
  - Two possible measures: `size` and `height`
- `self.left.size()` and `self.right.size()` are equal?
  - Only possible for complete binary trees

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

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

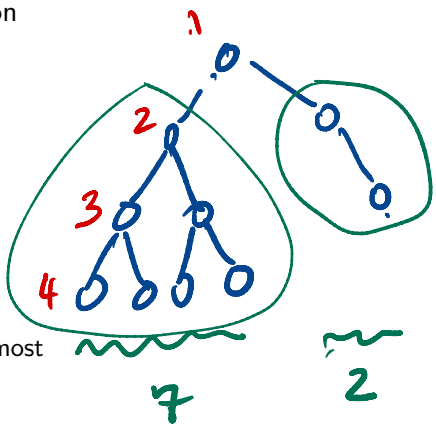


Height balanced trees

- `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



Want height = log(size)
Want size = exp(height)

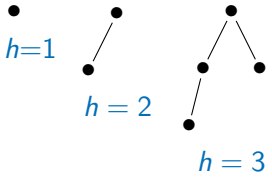
Height balanced trees

- `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?

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?

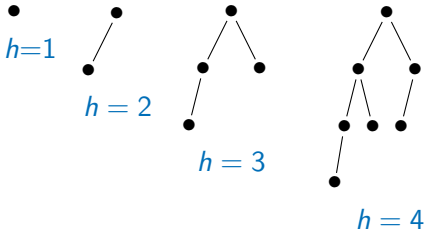
- Minimum size height-balanced trees



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?

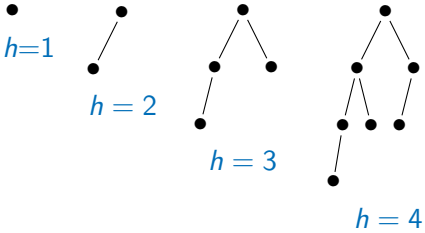
- Minimum size height-balanced trees



- 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

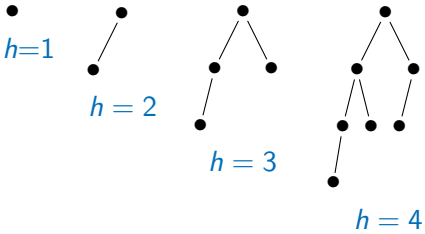
- Minimum size height-balanced trees



- 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

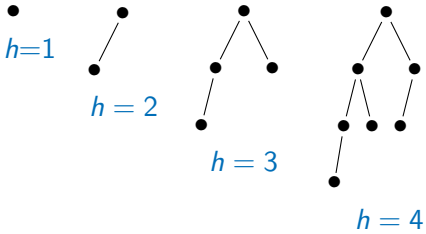


- General strategy to build a small balanced tree of height h
  - Smallest balanced tree of height $h-1$ as left subtree
  - Smallest balanced tree of height $h-2$ as right subtree

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

Height balanced trees

- Minimum size height-balanced trees



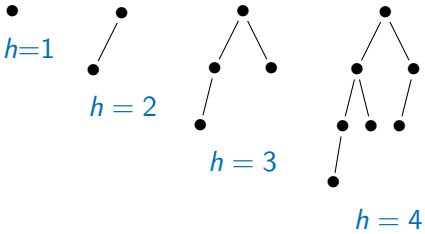
- General strategy to build a small balanced tree of height h
  - Smallest balanced tree of height $h-1$ as left subtree
  - Smallest balanced tree of height $h-2$ as right subtree

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

root left right

Height balanced trees

- Minimum size height-balanced trees

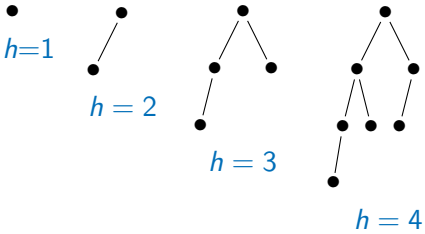


- General strategy to build a small balanced tree of height h
  - Smallest balanced tree of height $h-1$ as left subtree
  - Smallest balanced tree of height $h-2$ as right subtree

- $S(h)$, 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

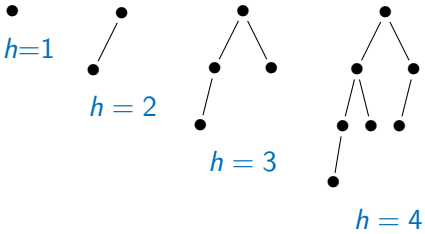


- General strategy to build a small balanced tree of height h
  - Smallest balanced tree of height $h-1$ as left subtree
  - Smallest balanced tree of height $h-2$ as right subtree

- $S(h)$, 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 $O(\log n)$