Data structures

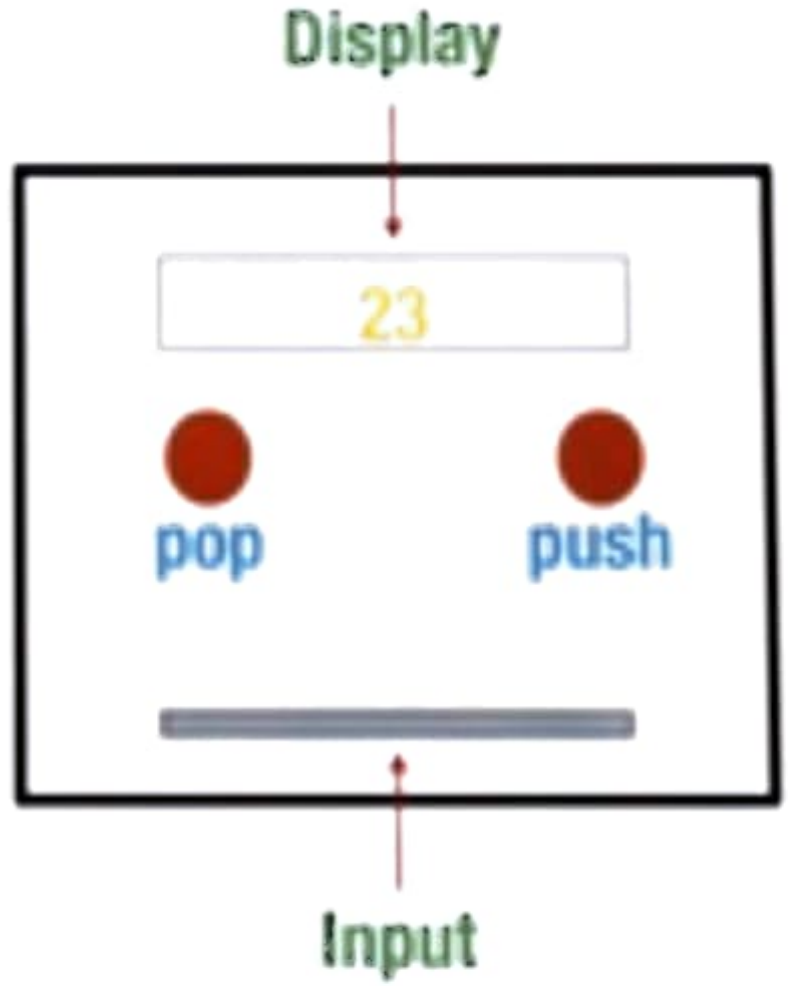
- Behaviour defined through interface
  - Allowed set of operations
- Stack: `push()` and `pop()`
- Queue: `addq()` and `removeq()`
- Heap: `insert()` and `delete_max()`
  - Heap implemented as a list `h`, does not mean `h.append(7)` is legal

Abstract datatype

- Define behaviour in terms of operations
  - `(s.push(v)).pop() == v`
  - `((q.addq(u)).addq(v)).removeq() == u`
- No reference to implementation details
- Implementation can be optimized without affecting functionality

Black box view

- Imagine the data structure as a black box
- Designated buttons to interact
- Slot for input
- Display for output
- No other manipulation allowed



Built in datatypes

```
l = []
```

- List operations `l.append()`, `l.extend()` permitted
  - … but not dictionary operations like `l.keys()`
- Likewise, after `d = {}`, `d.values()` is OK
  - … but not `d.append()`
- Can we do this for stacks, queues, heaps, …?

Object Oriented programming

- Data type definition with
  - Public interface
    - Operations allowed on the data
  - Private implementation
    - Match the specification of the interface

Classes and objects

- Class
  - Template for a data type
    - How data is stored
    - How public functions manipulate data

Classes and objects

- Class
  - Template for a data type
    - How data is stored
    - How public functions manipulate data
- Object
  - Concrete instance of template

Classes and objects

```
class Heap:
  def __init__(self,l):
    # Create heap
    # from list l

  def insert(self,x):
    # insert x into heap

  def delete_max(self,x):
    # return max element
```

Classes and objects

```
class Heap:
  def __init__(self,l):
    # Create heap
    # from list l

  def insert(self,x):
    # insert x into heap

  def delete_max(self,x):
    # return max element
```

```
# Create object,
# calls __init__()
l = [14,32,15]
h = Heap(l)

# Apply operation
h.insert(17)

h.insert(28)

v = h.delete_max()
```

Summary

- An abstract data type is a black box description
  - Public interface — update/query the data type
  - Private implementation — change does not affect functionality
- Classes and objects can be used for this
- More details in the next lecture

Classes and objects

- Class
  - Template for a data type
    - How data is stored
    - How public functions manipulate data
- Object
  - Concrete instance of template

Classes and objects

```
class Heap:
  def __init__(self,l):
    # Create heap
    # from list l

  def insert(self,x):
    # insert x into heap

  def delete_max(self,x):
    # return max element
```

```
# Create object,
# calls __init__()
l = [14,32,15]
h = Heap(l)

# Apply operation
h.insert(17)

h.insert(28)

v = h.delete_max()
```

Points on a plane

```
class Point:
  def __init__(self,a,b):
    self.x = a
    self.y = b

  def translate(self,deltax,deltay):
    # shift (x,y) to (x+deltax,y+deltay)
    self.x += deltax # same as selfx =
                     #         self.x + deltax
    self.y += deltay
```

Points on a plane

```
p = Point(3,2)
```

```
class Point:
  def __init__(self,a,b):
    self.x = a
    self.y = b

  def translate(self,deltax,deltay):
    # shift (x,y) to (x+deltax,y+deltay)
    self.x += deltax # same as selfx =
                     #         self.x + deltax
    self.y += deltay
```

Points on a plane

```
p = Point(3,2)
p.translate(2,1)
```


```
class Point:
  def __init__(self,a,b):
    self.x = a
    self.y = b

  def translate(self,deltax,deltay):
    # shift (x,y) to (x+deltax,y+deltay)
    self.x += deltax # same as selfx =
                     #          self.x + deltax
    self.y += deltay
```

Points on a plane

```
class Point:
  . . .
  def odistance(self):
    # Distance from (0,0)
    # from math import *
    return(
      sqrt(
        (self.x*self.x) + (self.y*self.y)
      ))
```

Points on a plane


```
class Point:
  . . .
  def odistance(self):
    # Distance from (0,0)
    # from math import *
    return(
      sqrt(
        (self.x*self.x) + (self.y*self.y)
      ))
```

```
p = Point(3,4)
p.odistance()
```

Polar coordinates

- Recall polar coordinates
- Instead of (x,y), use (r,θ)
  - $x = r \cos \theta$
  - $y = r \sin \theta$
  - $r = \sqrt{(x^2 + y^2)}$ — same as distance
  - $\theta = \tan^{-1}(y/x)$

Points on a plane

```
class Point:
  def __init__(self,a,b):
    self.r = sqrt(a*a + b*b)
    if a == 0:
      self.theta = 0
    else:
      self.theta = atan(b/a)

  def odistance(self):
    return(self.r)

  def translate(self,deltax,deltay):
    # Convert (r,theta) to (x,y) and back!
```

Points on a plane

```
class Point:
  def __init__(self,a,b):
    self.r = sqrt(a*a + b*b)
    if a == 0:
      self.theta = 0
    else:
      self.theta = atan(b/a)

  def odistance(self):
    return(self.r)

  def translate(self,deltax,deltay):
    # Convert (r,theta) to (x,y) and back!
```

Points on a plane

```
class Point:
  def __init__(self,a,b):
    self.r = sqrt(a*a + b*b)
    if a == 0:
      self.theta = 0
    else:
      self.theta = atan(b/a)

  def odistance(self):
    return(self.r)

  def translate(self,deltax,deltay):
    # Convert (r,theta) to (x,y) and back!
```

- Private implementation has changed
- Functionality of public interface remains same

Default arguments

```
class Point:
  def __init(self,a=0,b=0):
    self.x = a
    self.y = b

  . . .
```

Default arguments

```
class Point:
  def __init(self,a=0,b=0):
    self.x = a
    self.y = b

  . . .
```

```
# Point at (3,4)
p1 = Point(3,4)

# Point at (0,0)
p2 = Point()
```

Special functions

- `__init__()`
  - Constructor, called when object is created

Special functions

- `__init__()`
  - Constructor, called when object is created
- `__str__()`
  - Return string representation of object
  - `str(o) == o.__str__()`
  - Implicitly invoked by `print()`

```
def __str__(self):  # For Point()
  return('('+str(self.x)+','+str(self.y)+')')
```

Special functions

- `__add__()`
  - Invoked implicitly by `+`
  - `p1 + p2 == p1.__add__(p2)`

```
def __add__(self,p):  # For Point()
  return(Point(self.x+p.x,self.y+p.y)

p1 = Point(1,2)
p2 = Point(2,5)
p3 = p1 + p2  # p3 is now (3,7)
```

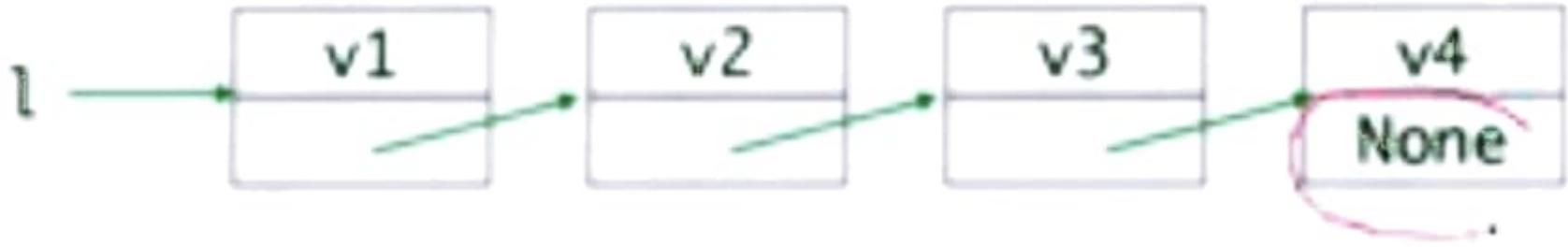
Special functions

- `__mult__()`
  - Called implicitly by *
- `__lt__()`, `__gt__()`, `__le__()`, . . .
  - Called implicitly by <, >, <=
- Many others, see Python documentation

Designing our own list

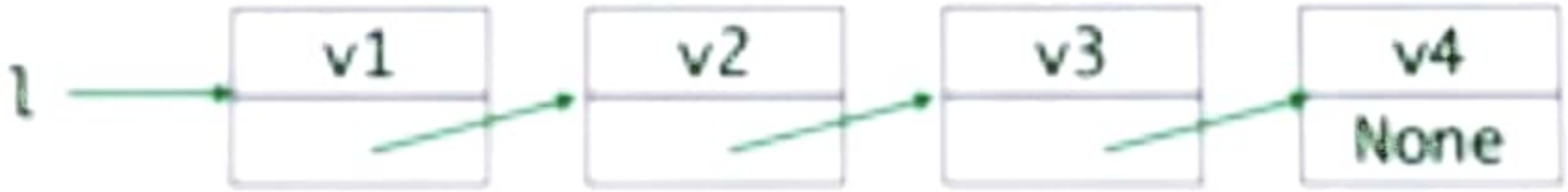
l = [v1, v2, v3, v4]

- A list is a sequence of nodes
- Each node stores a value, points to next node



Designing our own list

- A list is a sequence of nodes
- Each node stores a value, points to next node
- How do we represent the empty list?



Class Node

```
class Node:

  def __init__(self,initval=None):
    self.value = initial
    self.next = None

  def isempty(self):
    return(self.value == None)
```

Class Node

```
class Node:

  def __init__(self,initval=None):
    self.value = initial
    self.next = None

  def isempty(self):
    return(self.value == None)
```

```
# Create empty list
l1 = Node()

# Create singleton
l2 = Node(5)

l1.isempty()==True
l2.isempty()==False
```

Append a value v

- If list is empty, replace `None` by `v`
- If at last element of list (`next` is `None`)
  - Create a node with value `v`
  - Set next to point to new node
- Otherwise, recursively append to rest of the list

Append a value v

```
def append(self,v):
    if self.isempty():
        self.value = v
    elif self.next == None:
        newnode = Node(v)
        self.next = newnode
    else:
        (self.next).append(v)
    return()
```

- If list is empty, replace None by v
- Scan the list till we reach the last element
- Append the element at the last element

Append value iteratively

```
def appendi(self,v):
    if self.isempty():
        self.value = v
        return()
    temp = self
    while temp.next != None:
        temp = temp.next
    newnode = Node(v)
    temp.next = newnode
    return()
```

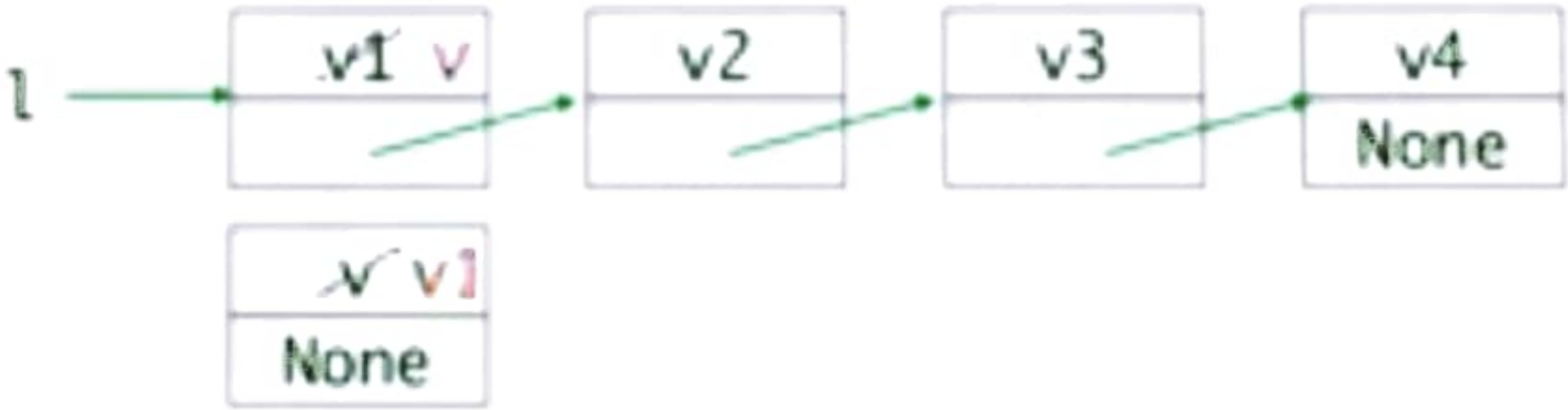
Append value iteratively

```
def appendi(self,v):
    if self.isempty():
        self.value = v
        return()
    temp = self
    while temp.next != None:
        temp = temp.next
    newnode = Node(v)
    temp.next = newnode
    return()
```

temp.next != None

Insert a value v

- Want to insert v at the head of the list
- Create a new node with v
  - But we cannot change where l points to!
- Instead, swap the contents of v with the current first node

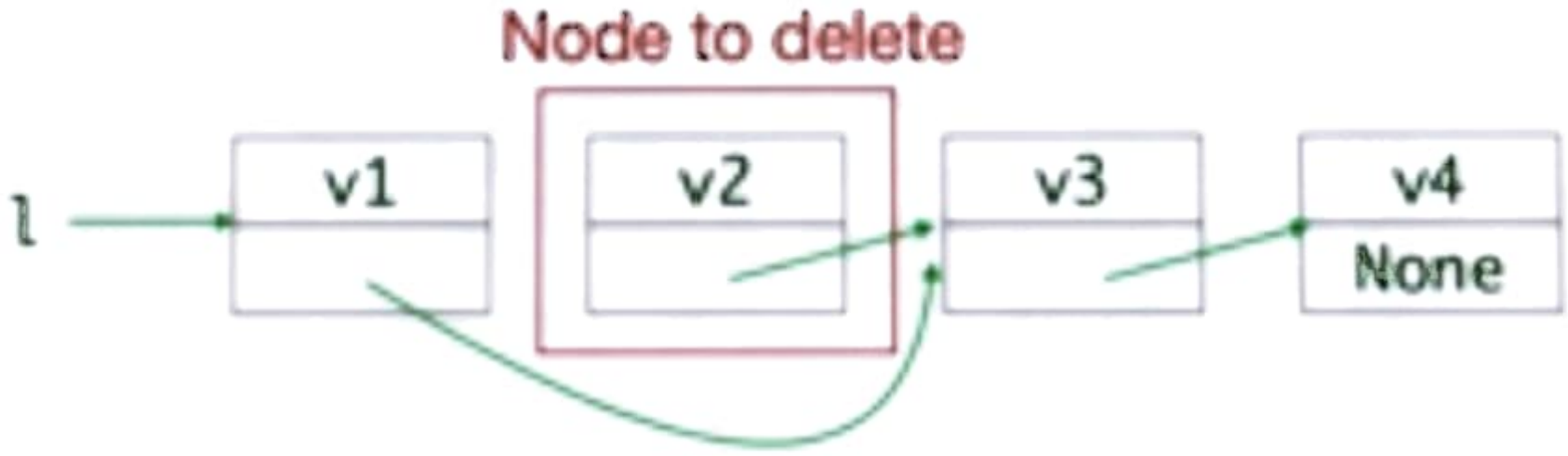


Insert a value v

```
def insert(self,v):

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

  newnode = Node(v)

  # Exchange values in self and newnode
  (self.value, newnode.value) =
                          (newnode.value, self.value)
  (self.next, newnode.next) = (newnode, self.next)

  return()
```

Deleting a node

- Do some plumbing on the list
- Reset next pointer to bypass deleted node



Delete a value v

- Remove first occurrence of v
- Scan list for first v
- If `self.next.value == v`, bypass `self.next`
  - `self.next = self.next.next`
- What if first value in the list is v?

Deleting first value in list

- `l.delete(v1)`
- Cannot delete the node that `l` points to
  - Reassigning name in function creates a new object
- Instead, copy `v2` from next node and delete second node!

Delete a value v

```
def delete(self,x):
  if self.isempty():
    return()

  if self.value == x: # value to delete
                      # is in first node
    if self.next == None
      self.value = None
    else:
      self.value = self.next.value
      self.next = self.next.next
      return()
```

Delete a value v

```
def delete(self,x):
  if self.isempty():
    return()

  if self.value == x: # value to delete
                      # is in first node

    . . .

  temp = self # find first x to delete
  while temp.next != None:
    if temp.next.value == x:
      temp.next = temp.next.next
      return()
    else:
      temp = temp.next

  return()
```

Delete value v, recursively

- If v occurs in first node, delete as before
- Otherwise, if there is a next node, recursively delete v from there
  - If `next.value == v` and `next.next == None`, `next.value` becomes `None`
  - If so, terminate the list here

Delete value v, recursively

```
def deleter(self,x):
  if self.isempty():
    return()
  if self.value == x: # value to delete is in first node
    if self.next == None
      self.value = None
    else:
      self.value = self.next.value
      self.next = self.next.next
      return()
  else: # recursive delete
    if self.next != None:
      self.next.deleter(v)
      if self.next.value == None:
        self.next = self.next.next
  return()
```

Printing out the list

```
def __str__(self):
  selflist = []

  if self.value == None:
    return(str(selflist))

  temp = self
  selflist.append(temp.value)

  while temp.next != None:
    temp = temp.next
    selflist.append(temp.value)

  return(str(selflist))
```

```
lass Node:
    def __init__(self, v = None):
        self.value = v
        self.next = None
        return

    def isempty(self):
        if self.value == None:
            return(True)
        else:
            return(False)

    def append(self,v):    # append, recursive
        if self.isempty():
            self.value = v
        elif self.next == None:
            newnode = Node(v)
            self.next = newnode
        else:
            self.next.append(v)
        return()

    def insert(self,v):
```

U-:----F1 list.py Top L1 (Python)

```
def delete(self,v):    # delete, recursive
    if self.isempty():
        return

    if self.value == v:
        self.value = None
        if self.next != None:
            self.value = self.next.value
            self.next = self.next.next
        return
    else:
        if self.next != None:
            self.next.delete(v)
            if self.next.value == None:
                self.next = None

    return

def __str__(self):
    selflist = []
    if self.value == None:
        return(str(selflist))
```

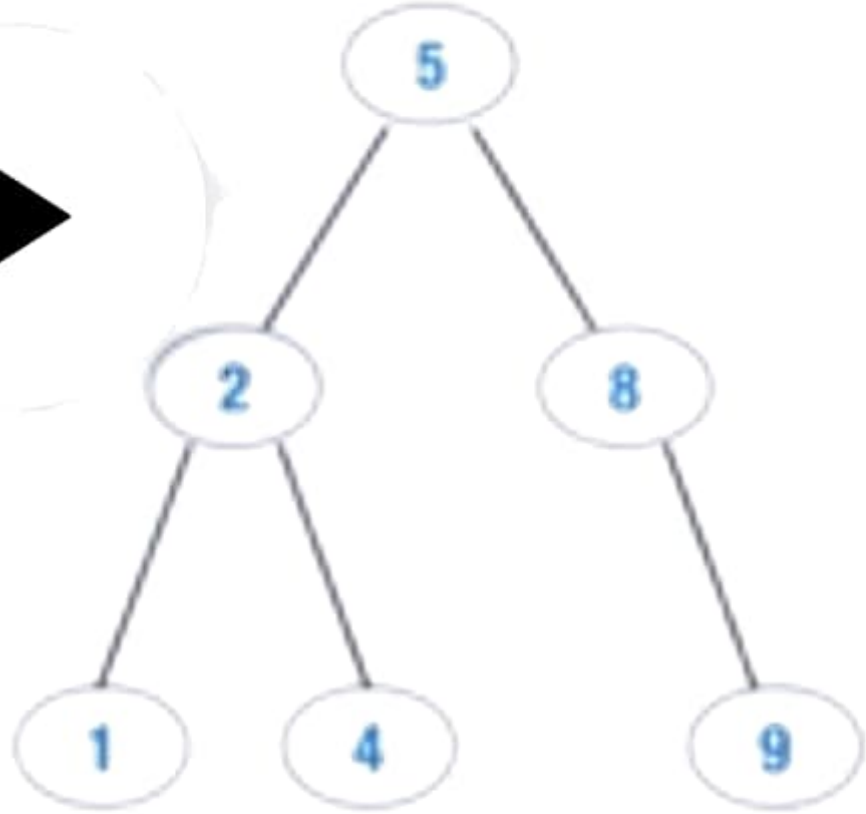

>>> print(l)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> l.delete(4)
>>> print(l)
[0, 1, 2, 3, 5, 6, 7, 8, 9, 10]
>>> l.insert(12)
>>> print(l)
[12, 0, 1, 2, 3, 5, 6, 7, 8, 9, 10]
>>>

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 sorted list take time O(n)
- Like priority queues, move to a tree structure

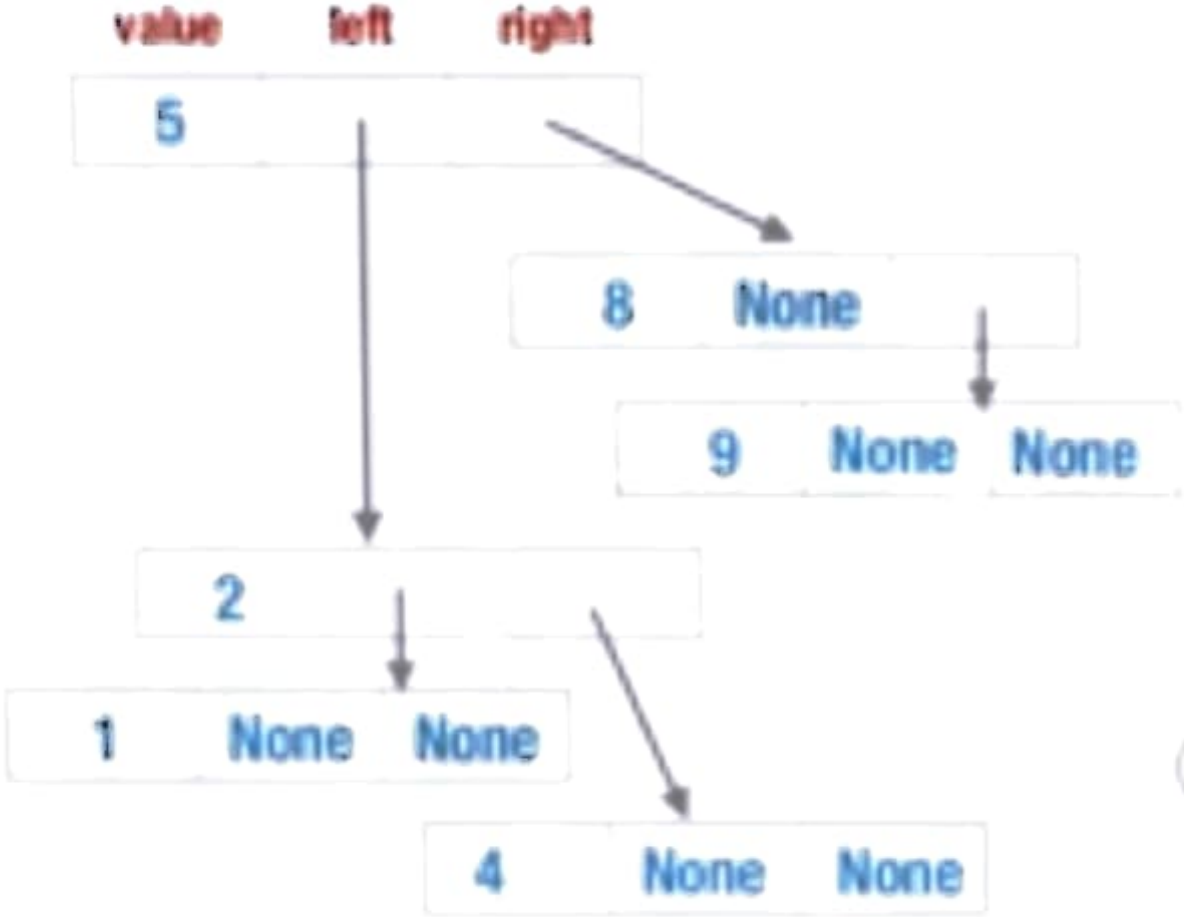
Binary search tree

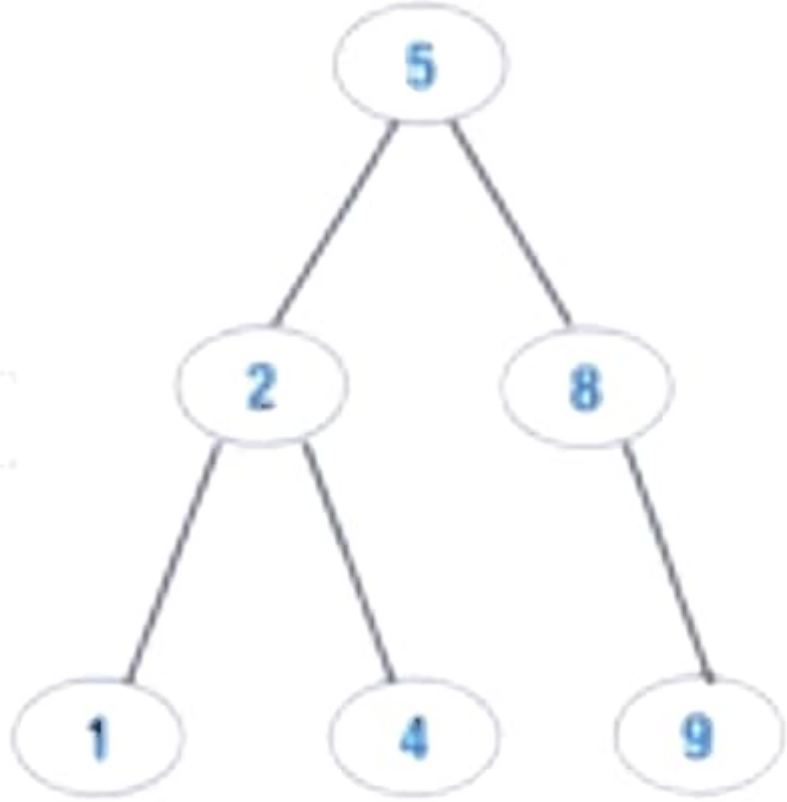
- For each node with value v
  - Values in left subtree < v
  - Values in right subtree > v
- No duplicate values



Binary search tree

- Each node has a value and points to its children

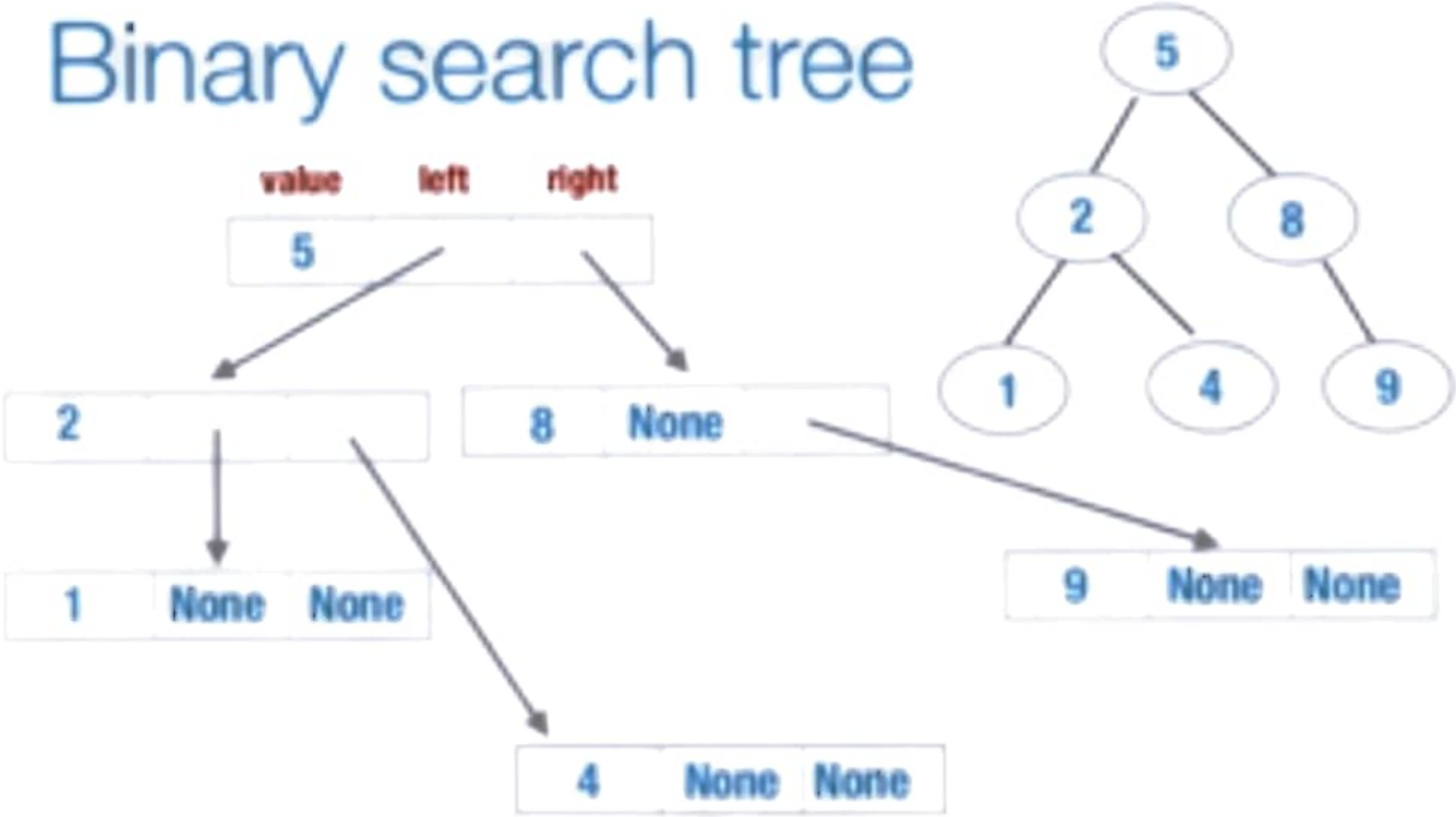




A better representation

- Add a frontier with empty node: all fields `None`
- Empty tree is a single empty node
- Leaf node has value that is not `None`, left and right children point to empty nodes
- Makes it easier to write recursive functions to traverse the tree

Binary search tree

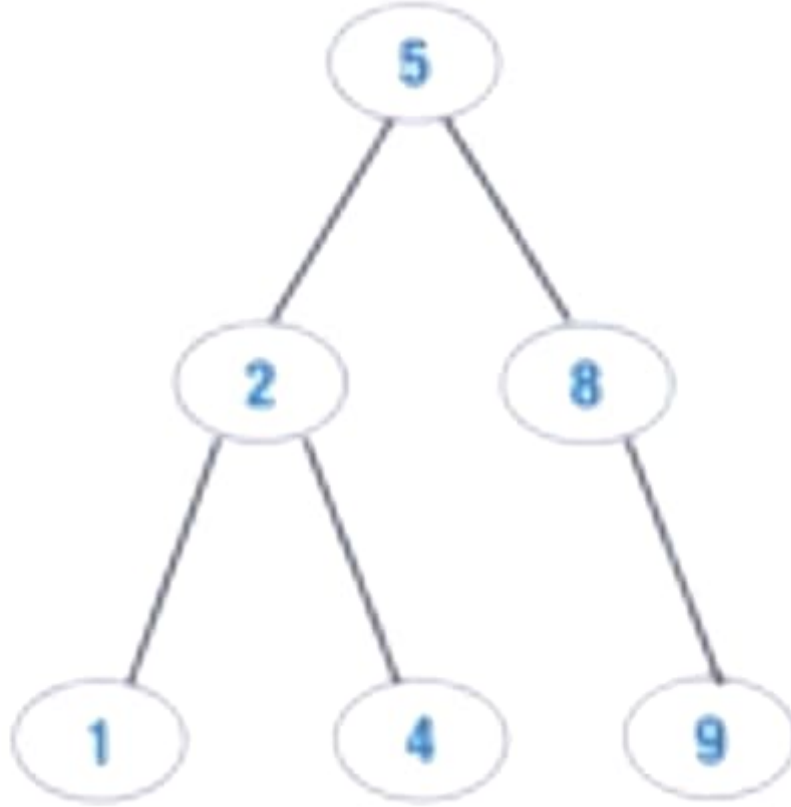


The class Tree

```
class Tree:
  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()

  def isempty(self):
    return(self.value == None)
```

Inorder traversal

```
def inorder(self):
 if self.isempty():
   return([])
 else:
   return(
     self.left.inorder() +
     [self.value] +
     self.right.inorder()
     )

def __str__(self):
  return(str(self.inorder()))
```

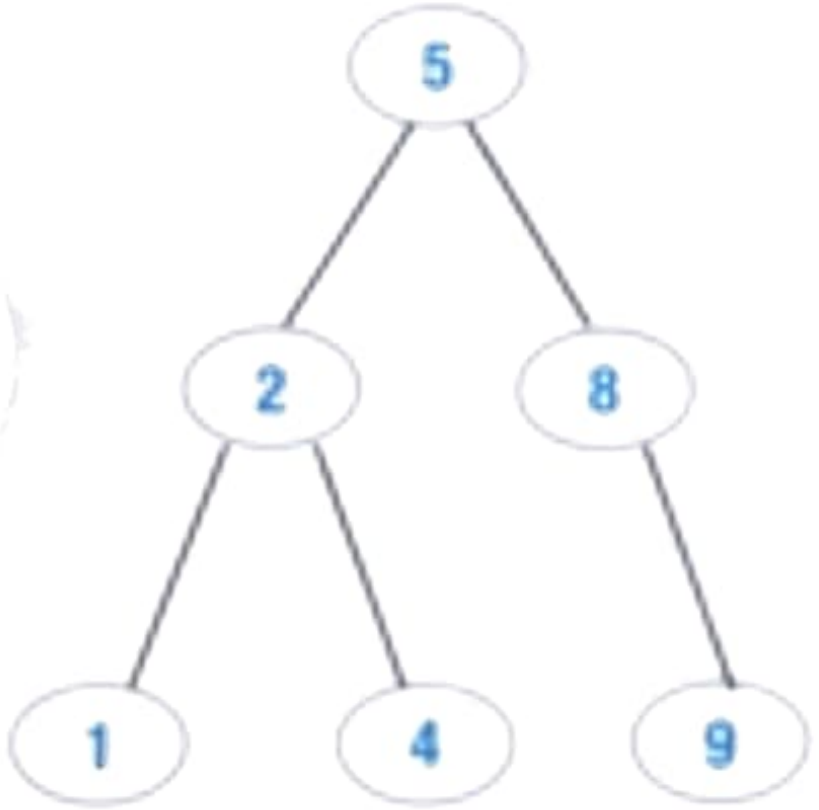


Find a value v

- Scan the current node
- Go left if v is smaller than this node
- Go right if v is larger than this node
- Natural generalization of binary search

Inorder traversal

```
def inorder(self):
 if self.isempty():
   return([])
 else:
   return(
     self.left.inorde
     [self.value] +
     self.right.inorder()
     )

def __str__(self):
  return(str(self.inorder()))
```



- Lists values in sorted order

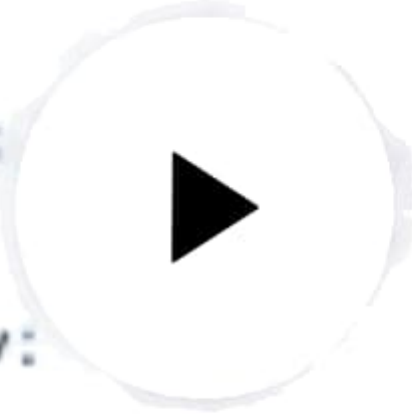
1 2 4 5 8 9

Find a value v

- Scan the current node
- Go left if v is smaller than this node
- Go right if v is larger than this node
- Natural generalization of binary search

Find a value v

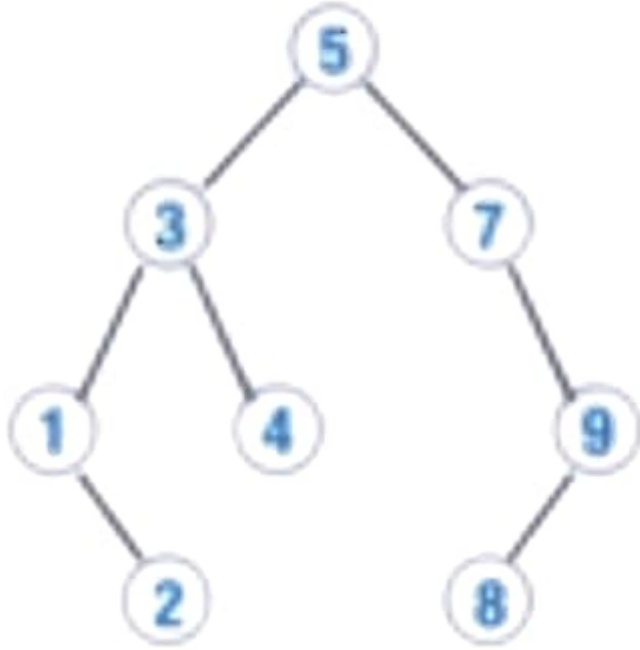
```
def find(self,v):

  if self.isempty():
    return(False)

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

  if v < t.value:
    return(self.left.find(v))
  else
    return(self.right.find(v))
```



Minimum

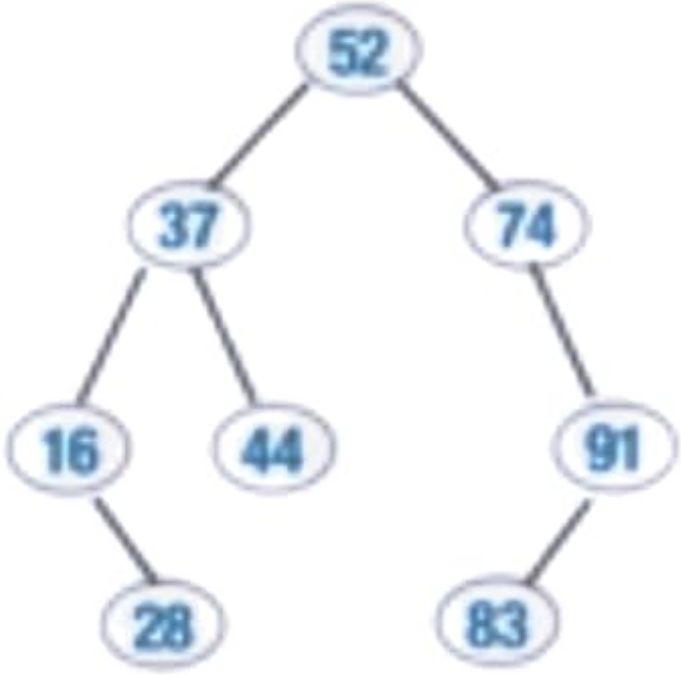
- Left most node in the tree

```
def minval(self):

# Assume t is not empty

  if self.left == None:
    return(self.value)
  else:
    return(self.left.minval())
```



Insert v

- Try to find v
- If it is not present, add it where the search fails

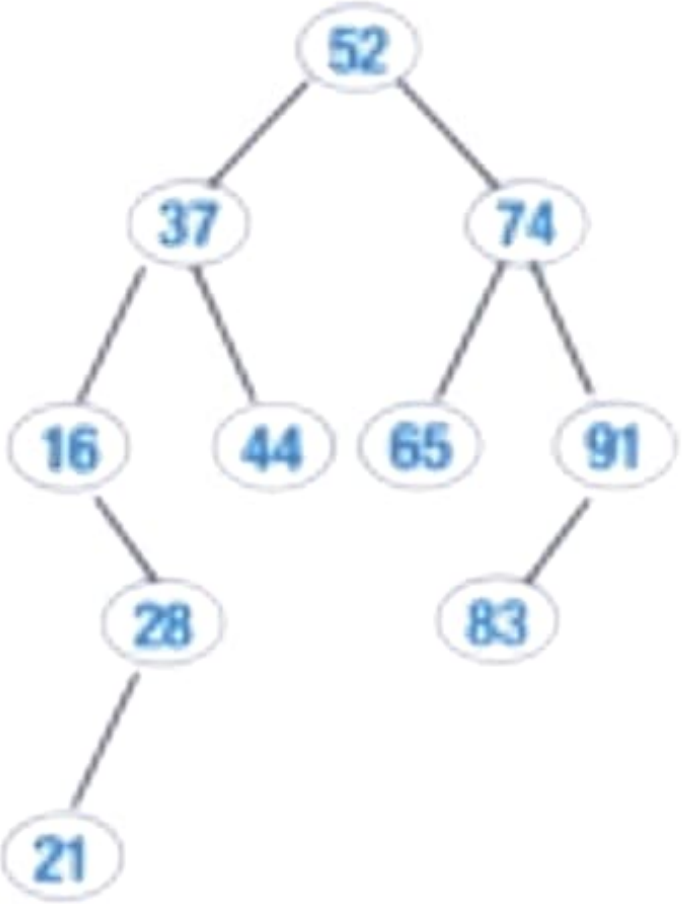


Insert v

```
def insert(self,v):

  if self.isempty():    # Add v as a new leaf
    self.value = v
    self.left = Tree()
    self.right = Tree()

  if self.value == v:  # Value found, do nothing
    return

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

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

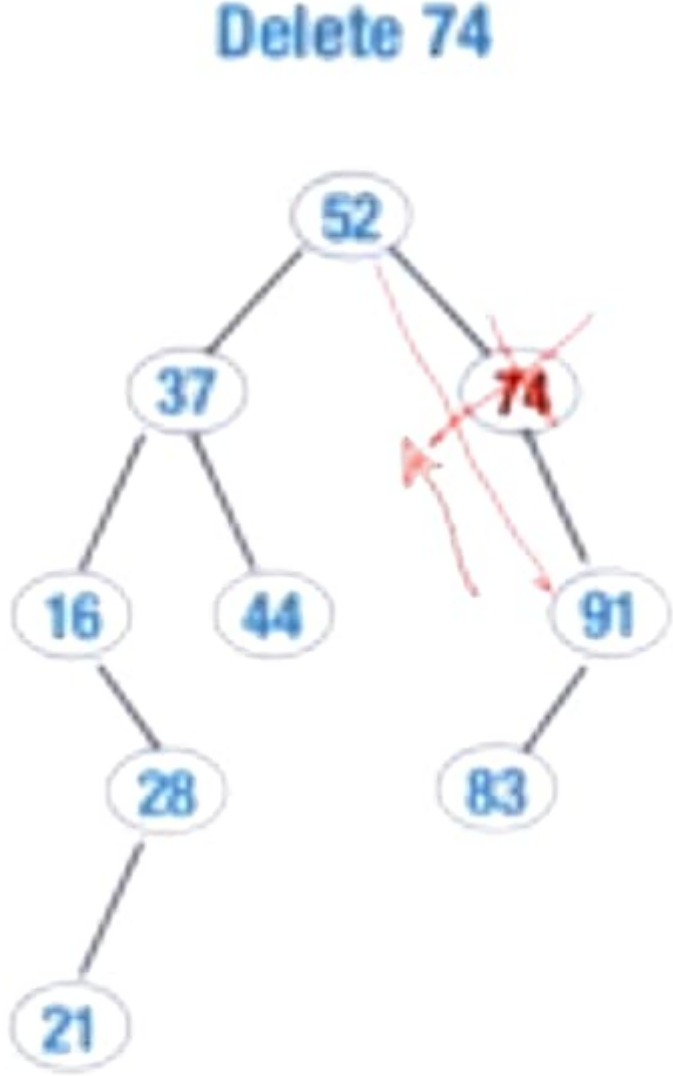
Delete v

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()` — must be leaf or have only one child



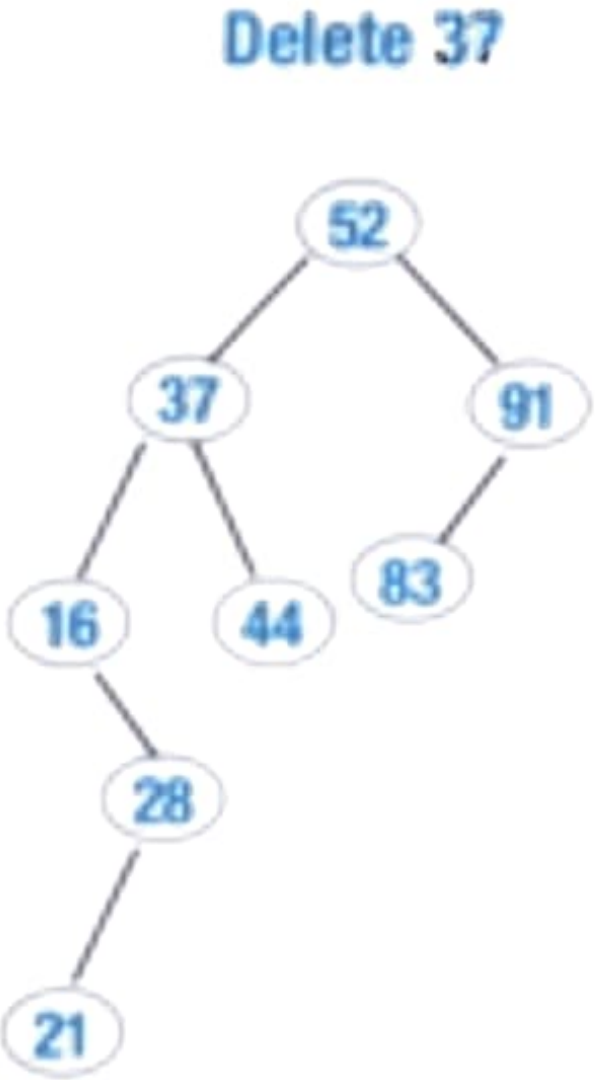
Delete v

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()` — must be leaf or have only one child



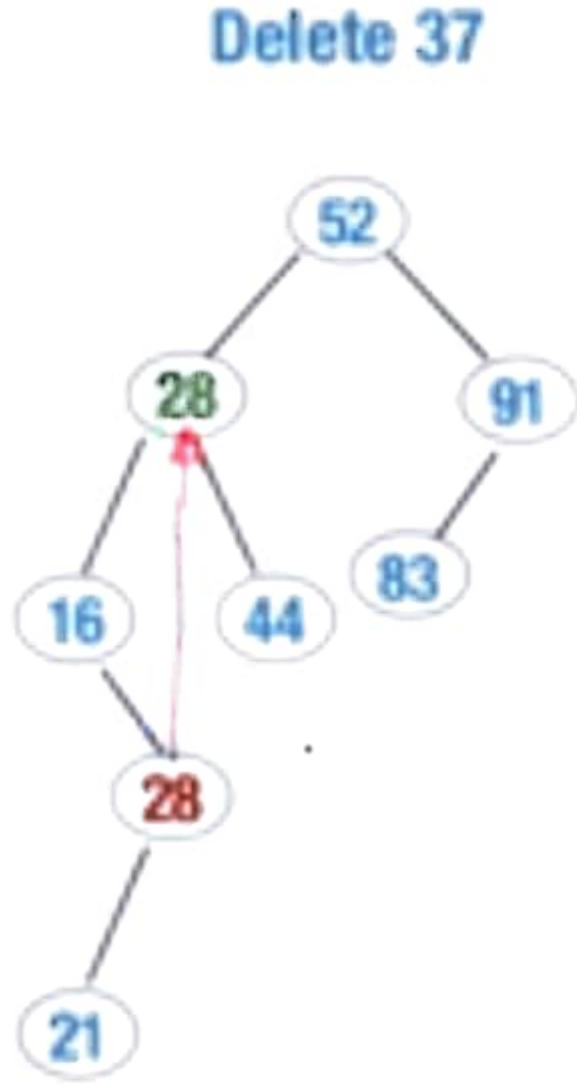
Delete v

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()` — must be leaf or have only one child



Delete v

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()` — must be leaf or have only one child



Delete v

```
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()
    else:
       self.value = self.left.maxval()
       self.left.delete(self.left.maxval())
    return
```

Delete v

```
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()
    else:
      self.value = self.left.maxval()
      self.left.delete(self.left.maxval())
    return
```

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

# Copy right child values
# to current node
def copyright(self):
  self.value =
self.right.value
  self.left =
self.right.left
  self.right =
self.right.right
  return
```

Complexity

- All operations on search trees walk down a single path
- Worst-case: height of the tree
- Balanced trees: height is $O(\log n)$ for n nodes
- Tree can be balanced using rotations — look up AVL trees

```
class Tree:

    # Empty node has self.value, self.left, self.right = None
    # Leaf has self.value != None, and self.left, self.right point to empty node

    # Constructor: create an empty node or a leaf node, depending on initval
    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.left.isempty() and self.right.isempty())

    # Convert a leaf node to an empty node
```

```
[redacted]:...on-2016-jul/week7/python3 python3.5
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from searchtree import *
>>> t = Tree()
>>> for i = [ 1 , 3 , 2, 18, 7, 5, 4, 22, 14]
  File "<stdin>", line 1
    for i = [ 1 , 3 , 2, 18, 7, 5, 4, 22, 14]
          ^
SyntaxError: invalid syntax
>>> for i in  [ 1 , 3 , 2, 18, 7, 5, 4, 22, 14]:
...     t.insert(i)
...
>>> print(t)
[1, 2, 3, 4, 5, 7, 14, 18, 22]
>>> t.insert(17)
>>> print(t)
[1, 2, 3, 4, 5, 7, 14, 17, 18, 22]
>>> t.insert(4.5)
>>> print(t)
[1, 2, 3, 4, 4.5, 5, 7, 14, 17, 18, 22]
>>> t.delete(3)
>>> print(t)
[1, 2, 4, 4.5, 5, 7, 14, 17, 18, 22]
>>>
```