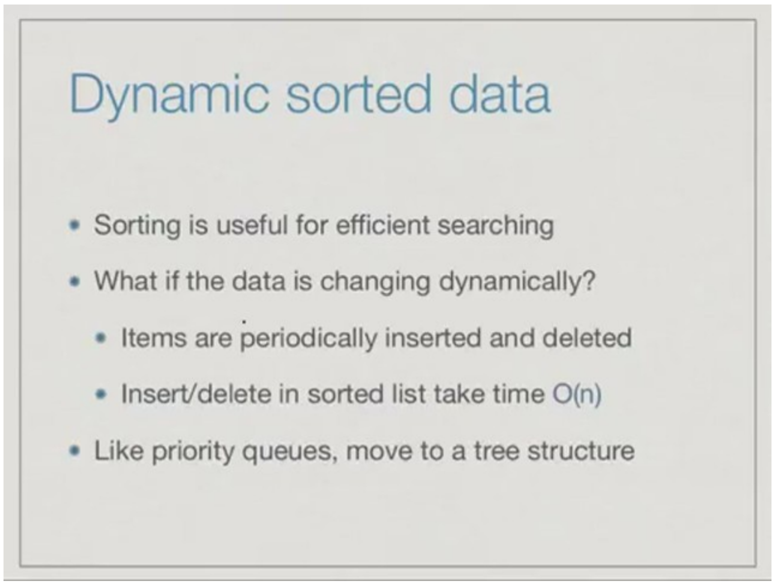
Programming, Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 07
Lecture - 04
Search Trees

(Refer Slide Time: 00:02)



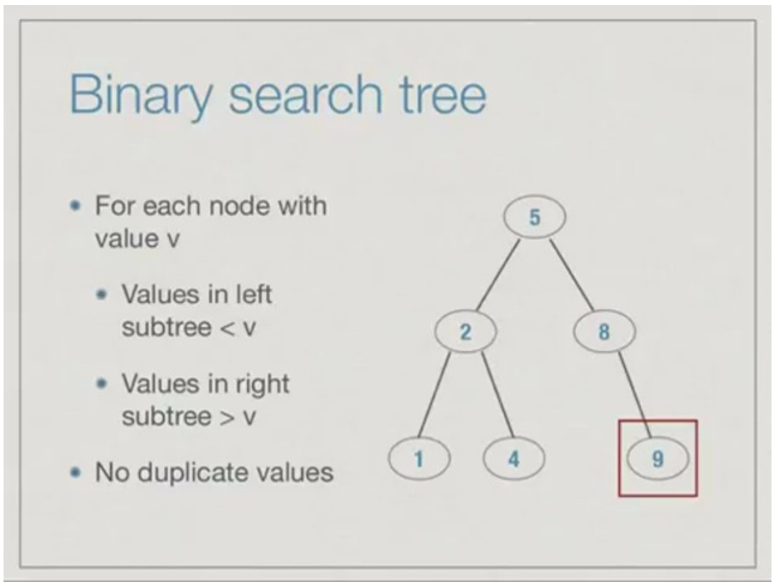
Dynamic sorted data

- 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

As a final example of a user defined data structure we will look at binary search keys. We are looking at a situation where we need to maintain dynamic data in a sorted manner, remember that one of the byproducts of sorting is that we can use binary search to efficiently search for a value, but binary search can be used if we can sort data once and for all and keep it in sorted order if the data is changing dynamically then in order to exploit binary search will have to keep resorting the data which is not efficient.

Supposing, we have a sequence of items and items are periodically being inserted and deleted now as we saw with heaps, for instance, if we try to maintain a sorted list and then keep track of inserts then each insert or delete, in this case would take order and time and that would also be expensive. However, it turns out that we can move to a tree like structure or a tree structure like in a priority queue it move to a heap and then do insert and delete also efficiently alongside searching.

(Refer Slide Time: 01:11)



Binary search tree

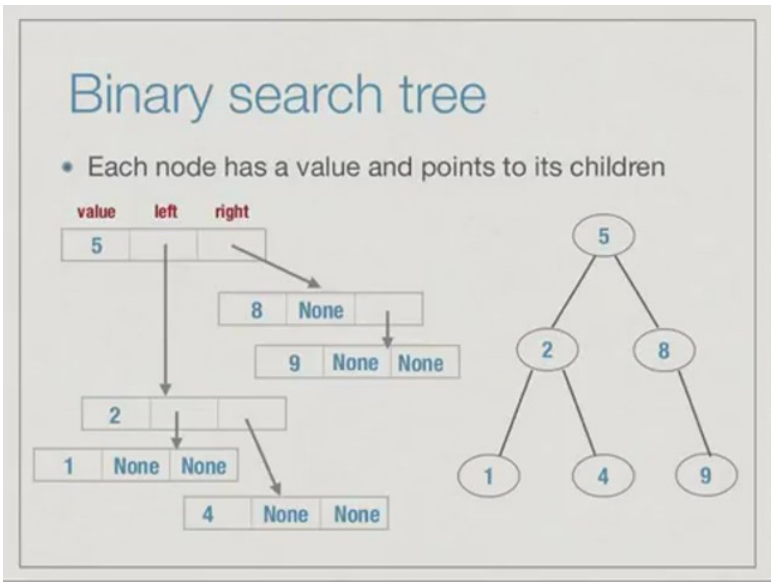
- For each node with value v
  - Values in left subtree < v
  - Values in right subtree > v
- No duplicate values

The data structure we have in mind is called a binary search tree. So, in a binary search tree we keep exactly one copy of every value. It is like a set we do not keep duplicates and the values are organized as follows for every node all values which are smaller than the current nodes value are to the left and all values that are bigger than the current node value are to the right.

Here is an example of a binary search tree, you can check for instance that to the left of the root 5, we have all values 1, 2 and 4 which are smaller than 5 and to the right of 5 we have the values 8 and 9 which are bigger, now this is a recursive property. If you go down, for instance, if you look at the node label two then below it has values 1 and 4 since 1 is smaller than 2, 1 is to the left of 2 and since 4 is bigger than 2, 4 is to the right of 2.

Similarly, if we look at the node 8, it has only one value below it namely 9 and therefore, it has no left child, but 9 is in the right subtree of 8.

(Refer Slide Time: 02:14)



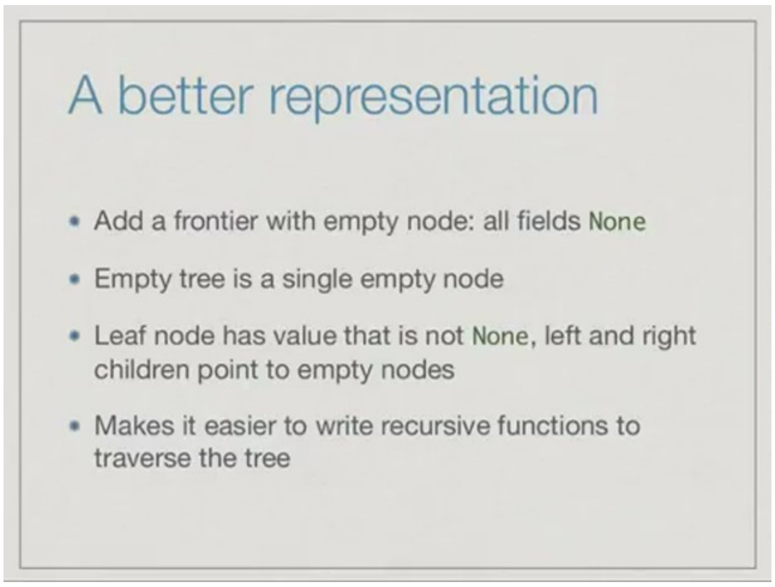
Binary search tree

- Each node has a value and points to its children

We can maintain a binary search tree using nodes exactly like we did for a user defined lists except in a list we had a value and a next pointer in a binary search tree we have two values below each node potentially a left child and a right child. So, each node now consist of three items the value being stored the left child and the right child.

If we look at the same example that we had 4 on the right then the root node 5 will have a pointer to the nodes with 2 and 8, the node 2 will have a pointer to the nodes 1 and 4, these are now what are called leaf nodes. So, they have no children. Their left and right pointers will be None indicating there is nothing in that direction. Similarly, 8 has got None as his left pointer because it has no left child and the right pointer points to 9 and the node with nine again has two None pointers because it is a leaf node.

(Refer Slide Time: 03:10)



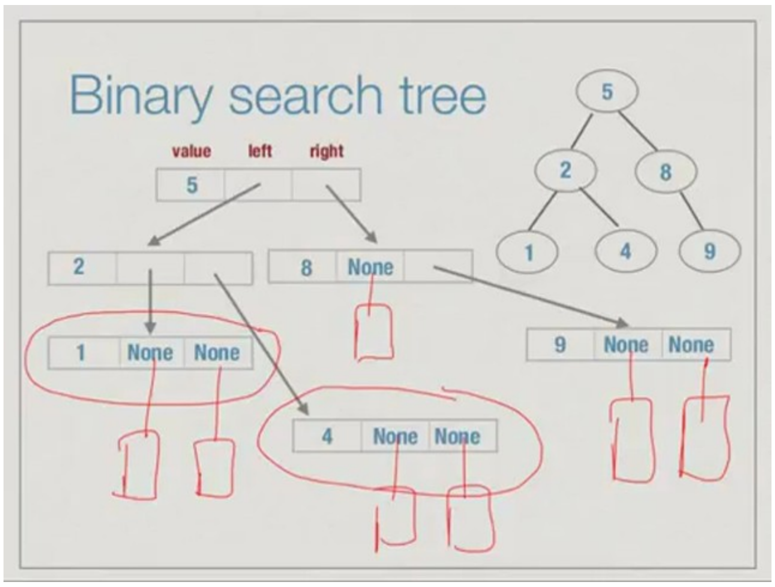
A better representation

- 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

Now, it will turn out that we will want to expand this representation in order to exploit it better for recursive presentations. So, what we will do is that we will not just terminate the tree with the value and the two pointers none you will actually add a layer of empty nodes with all fields None with this the empty tree will be a single empty node and a leaf node that is not none will have a value and both it is children will be empty nodes.

It would not directly have None as it is left and right pointers it will actually help children which are empty. So, this makes it easier to write recursive functions and if we do not do this then it is a bit harder to directly implement recursive functions.

(Refer Slide Time: 03:56)

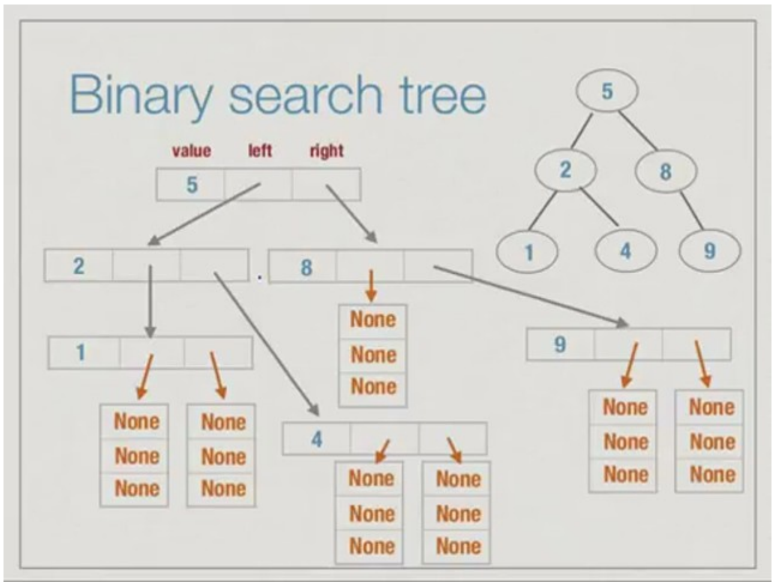


Binary search tree

Just to understand how our tree structure changes this is the structure that we had before the same example. Here, notice that in the leaf nodes the leaf nodes have a value and both the child pointers left and right are directly none. So, we want to change this, what we want to do is to we want to insert below this an empty node at everywhere, where we see None, we want to insert and extract the node.

This of course, grows up the tree a little bit, but then this cost is not that much as we will as you can calculate. So, if we do this then we get a new tree which looks like this right

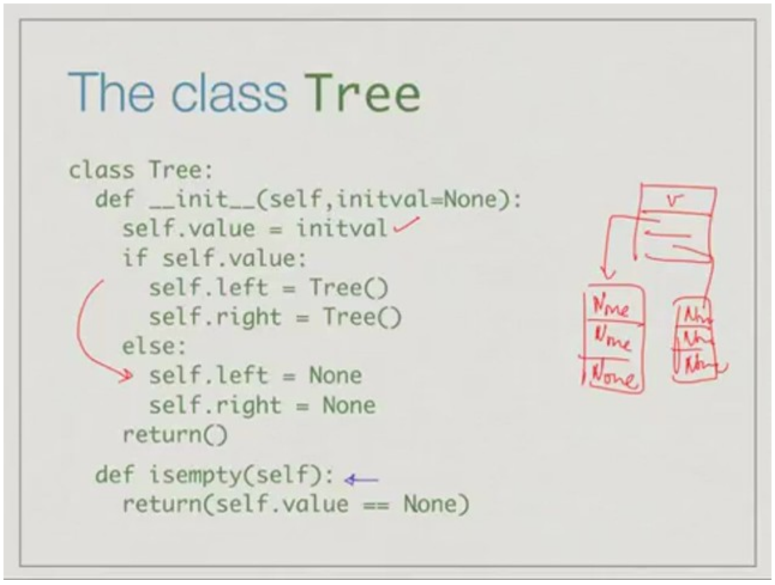
(Refer Slide Time: 04:28)



Binary search tree

Below every leaf node wherever we normally had a None pointer indicating that the path has ended we will explicitly add one extra node which has all three fields none and it will turn out that this is very useful to clean up our programming later on. So, this is a representation that we will use for a binary search tree each node has three pointers and at the leaf's we have an extra layer of empty nodes with all three values none, none, none.

(Refer Slide Time: 04:58)



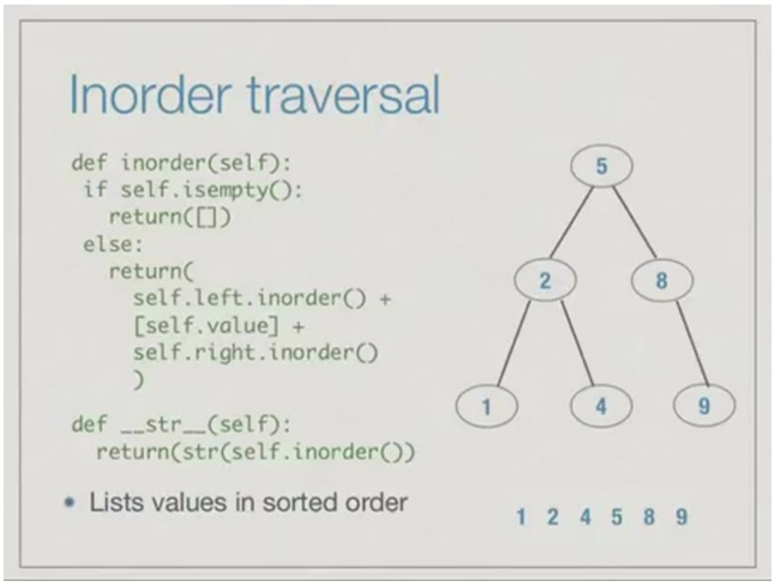
The class Tree

```
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)
```

Here is the basic class tree. So, as before we have an init function which takes an initial values which is by default none. The init function works as follows right, we first setup the value to be initval which could be none if there is a value, then we create an tree with one node in which case we have to create these empty nodes. Now, notice that if we go back and we go do not give a values. So, maybe it is better to look at this case right if we do not give a value then we end up with a tree we just says none, none, none. So, this is our empty.

The initial value is none we get this tree if the init value is not none then we get a tree in which we put the value v and then we make the left in the right pointers both point to this none, none. So, this is a tree that will contain exactly one value. So, depending on that the init values none or not none we end up either a tree with three nodes with two dummy nodes below or a single empty node denoting the empty tree. So, given this as before we have the function isempty which basically checks if the value is none, if I start looking at a tree and the very first node says none then that tree is empty otherwise it is not empty.

(Refer Slide Time: 06:15)



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()))
```

- Lists values in sorted order

5
2
8
1
4
9
1 2 4 5 8 9

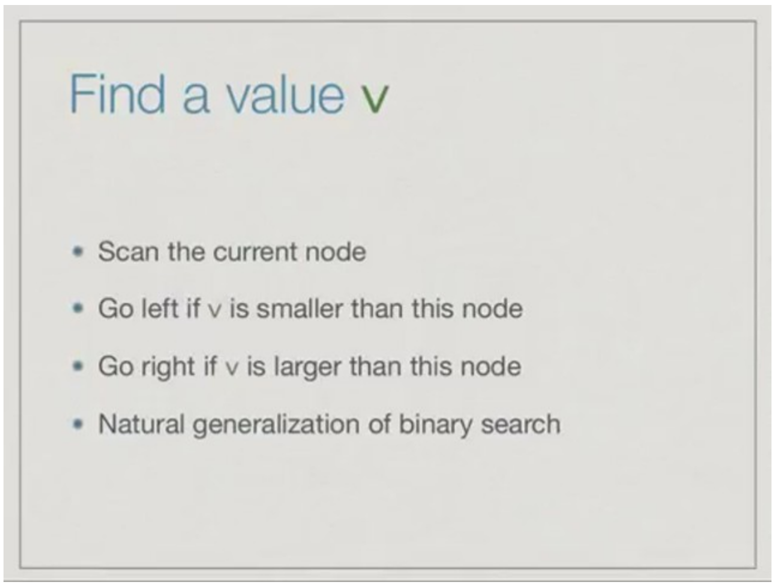
Let us first look at a way to systematically explore the values in a tree. These are called traversals and one traversal which is very useful for a binary search tree is an inorder traversal. So, what an inorder traversal does is that it first explores the left side. So, it will first explore this recursively again using an inorder traversal then it will display this and then it will explore the right.

If you see the code you can see that if the tree is not empty you first do an inorder traversal of the left self tree then you pick up the value at the current node and then you do an inorder traversal of the right self tree and this produces the list of values. So, if we execute this step by step, 5 if we reach it says first do an inorder traversal of the left.

We come down to two this is again not at a trivial tree. So, again we have to do a inorder traversal. So, we go it is left and now when we have one and inorder traversal of one consists of it is left child which is empty one and then it is right child is empty. So, this produces one now I come back and I list out the node two and now I do an inorder traversal of it is right. So, I have got 1, 2, 4. So, this completes inorder traversal of the left subtree of 5. Now, I list out 5 itself and then I do an inorder traversal of 8 and 9 since 8 has no left child the next values are comes out as a 8 itself and then 9.

So, what you can see is that since we print out the values on the left child before the current value when the value is the right child after the current value with respect to the current value all these values are sorted because that is how the search key is organized and since is recursively done at every level down the final output of a inorder traversal of a search tree is always a sorted list. This is one way to ensure that the tree that you have constructed is sorted you do an inorder sub traversal that the key of constructed is a search tree you do an inorder traversal and ensure that the output that you get from this traversal is a sorted list.

(Refer Slide Time: 08:16)



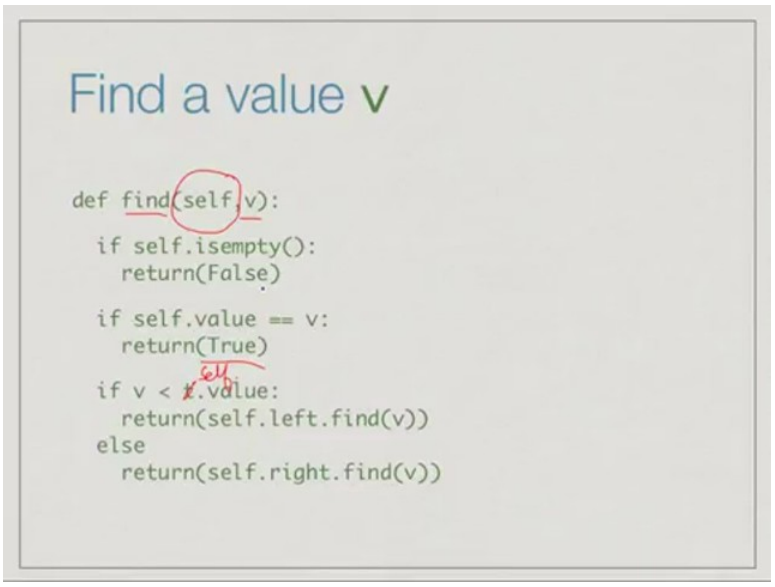
Find a value v

- 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

As we mentioned that the beginning of this lecture, one of the main reasons to construct a search tree is to be able to do something like binary search and this is with dynamic data. So, we will also have insert and delete as operations, but the main fundamental operation is find.

Like in binary search we start at the root. So, you imagine that this is the middle of an list of an array, for example, we look at this element if we have found it it is fine if we have not found it then we need to look in the appropriate sub tree since the search tree is organized with the left values smaller in the right value is bigger we go left if the value we were searching for is smaller than the current node and we go right if it is larger than the current node. So, this is very much a generalization of binary search in the tree.

(Refer Slide Time: 09:04)



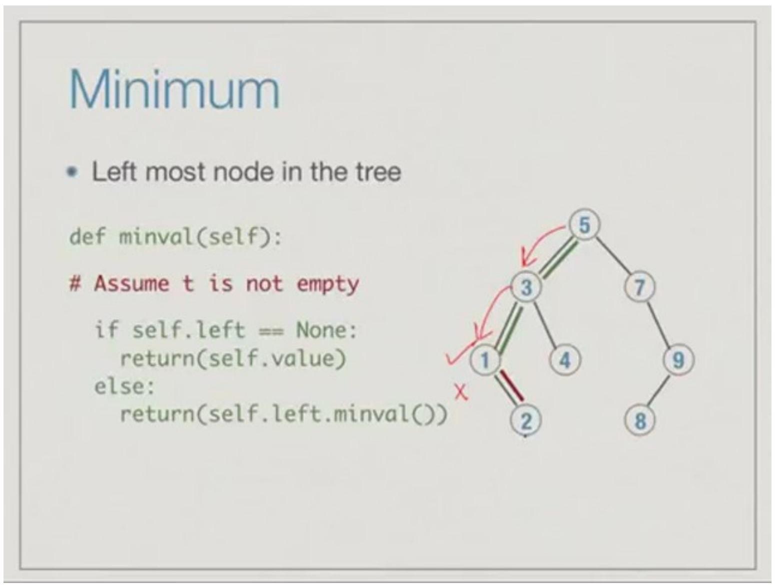
Find a value v

```
def find(self,v):
  if self.isempty():
    return(False)

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

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

Here is the code, it is very straight forward we want to find value of v remember, this is this python syntax. We always have the self as the first parameter to our function. So, if the current node is empty we cannot find it.

So, it is v return false if we do find v then we return true and if we do not find v then if it is. This is not, it should be self if it is smaller than this current value then we go left and search for it otherwise we go right and search for it. So, this is exactly a binary search and it is a very simple recursive thing which exactly follows a structure of a search tree.

(Refer Slide Time: 09:43)



Minimum

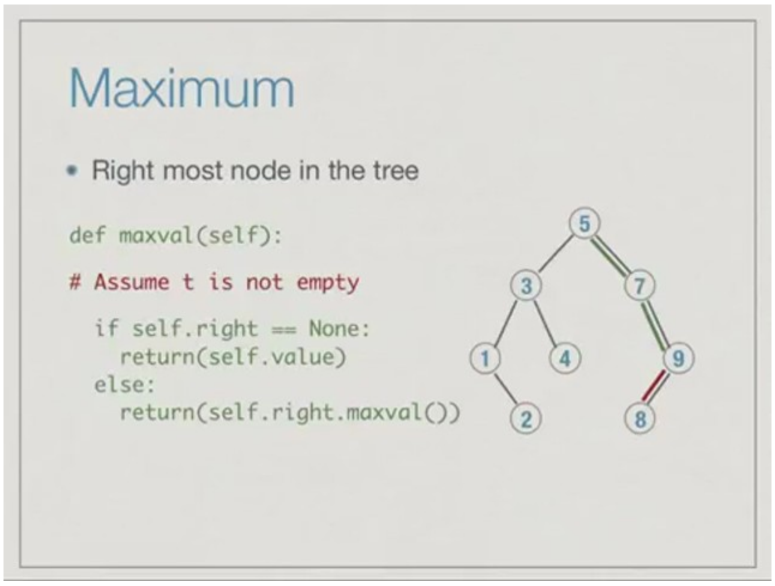
- Left most node in the tree

```
def minval(self):
# Assume t is not empty
  if self.left == None:
    return(self.value)
  else:
    return(self.left.minval())
```

It will be useful later on to be able to find the smallest and largest values in a search tree. So, notice that as we keep going left we go smaller and smaller. So, where is the minimum value in a tree it is along the smaller left most path. So, if I have to go from the left most path and if I cannot go any further then I find it. So, we will always apply this function only when tree is non empty. Let us assume that we are looking for the minimum value in a non empty tree. Well, we find the left most path.

If I cannot go further left then I found it. In this case, if I reach one since I cannot go further one is the minimum value otherwise if I can go left then I will go one more step. So, if we start this, for instance, say 5 it will say that 5 have left a subtree. So, the minimum value below 5 is the minimum value below it is left child. So, you go to three now we say that the minimum value below three is the minimum value below it is left child. So, you go to one then we say that the minimum value is the minimum value at 1 because there is no left child and therefore, we get the answer 1 and it is indeed 2 that anything below that is only on the right that is 2.

(Refer Slide Time: 10:50)



Maximum

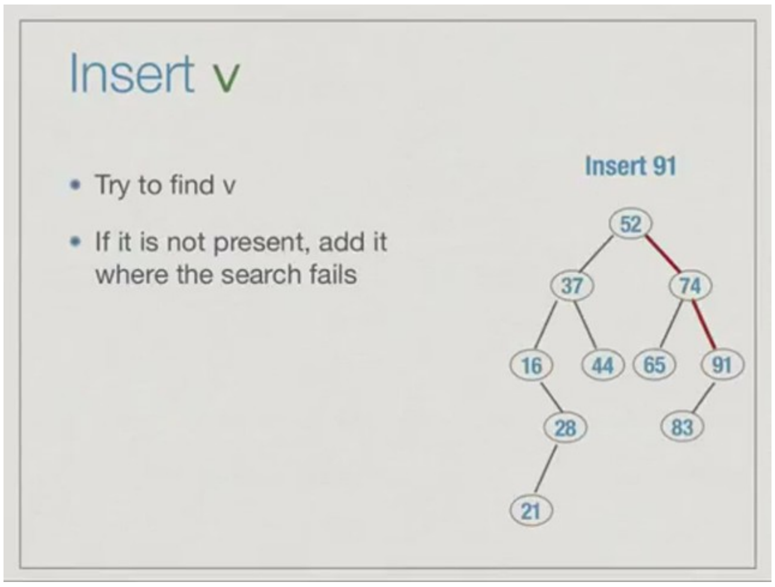
- Right most node in the tree

```
def maxval(self):
# Assume t is not empty
  if self.right == None:
    return(self.value)
  else:
    return(self.right.maxval())
```

Dually, one can find the maximum value by following the right most path right, if the right is none then we return the current value otherwise we recursively look for the maximum value to our right.

In this case we start at 5, we go down to 7, then we go down to 9 and since there is no further right path from 9, 9 must be the maximum value in this tree. We will come back later on and see why we need this minimum and maximum value, but any way it is useful thing to be able to find and it is very easy to do, again using the structure of the binary search.

(Refer Slide Time: 11:25)



Insert v

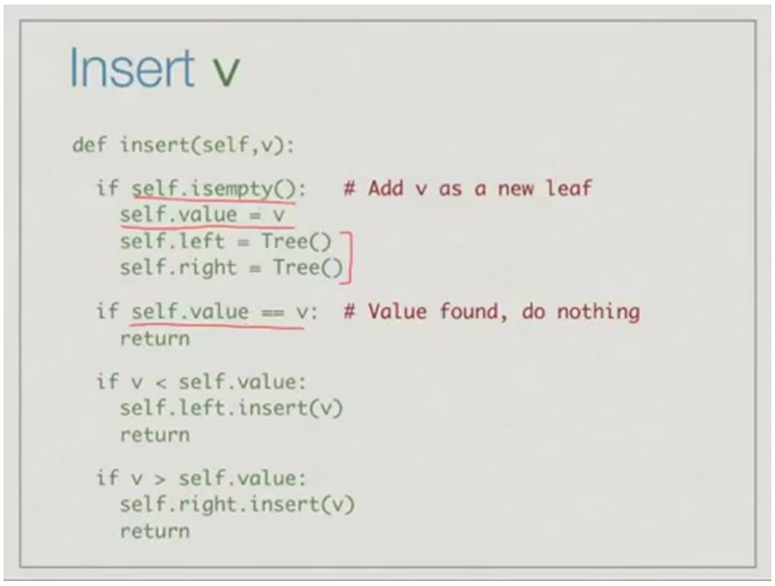
- Try to find v
- If it is not present, add it where the search fails

So, one of the things we said is that we need to be able to dynamically add and remove elements from the tree. The first function that we look for is insert, how do we insert an element in the tree well it is quite easy we look for it if we do not find it then the place where are search concludes is exactly where it should have been. So, we insert it at that point right.

For example, supposing we try to insert 21 in this tree, when we look at 52 and when go left when we go left then we come to 16 again and we go right then we come to 21, 28 and we find that we have exhausted this path and there is no possible 21 in this tree, but had we found 21 it should be to the left of 28. So, we insert it there. So, we look for where it should find it and if we do not find it we insert it with the appropriate place. Similarly, you can start and look for 65. So, 65 is bigger than 52. So, we go over right then it is smaller than 74. So, we go left, but there is nothing to the left 74. So, we out it to the left of 74.

Now, insert will not put in a value that is already there because we have no duplicates. So, if now we try to for instance insert ninety one then we go right from 52 we go right from 74 and now we find that 91 is already present in the tree. So, insert ninety one has no effect on this tree.

(Refer Slide Time: 12:50)



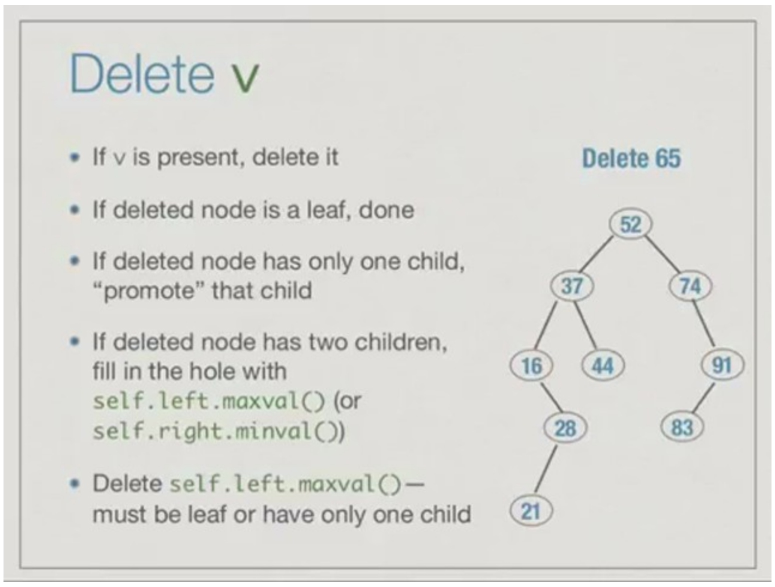
Insert v

```
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
```

This is a very simple modification of the find algorithm. So, we keep searching and if we reach the leaf node then we come to an empty node right. If you find that we have reached an empty node then we do the equivalent of creating a new node here we set this value to be v and we create a new frontier below by adding two empty nodes in the left and right rather than just having none.

On the other hand, if we find the value in the tree we do nothing and if we do not find the value then we just use the search tree property and try to insert either on the left or on the right as appropriate.

(Refer Slide Time: 13:32)



Delete v

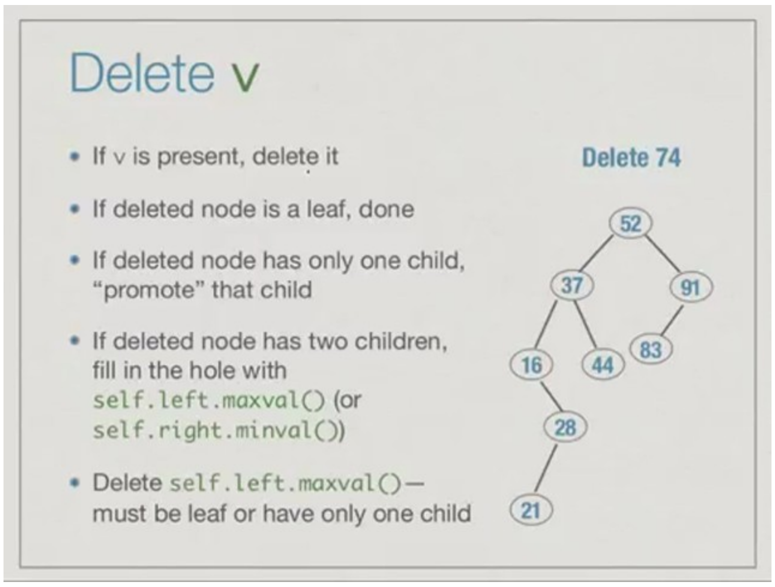
- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()`— must be leaf or have only one child

Delete 65
52
37
74
16
44
91
28
83
21

How about delete. So, delete is a little bit more complicated than insert. So, basically whenever we find v in the tree and that can only be one v remember this is not like deleting from the list that we had before where we were removing the first copy of v in a search tree we have only one copy of every value if at all. If we find v we must delete it.

So, we search for v as usual now if the node that we are searching for is a leaf then we are done we just delete it and nothing happens, if it has only one child then we can promote child. If it has two children we have a hole we have a leaf we have a node which we have to remove value, but we have values on both sides below it and now we will use this maximum function maxval or minval in order to do the work. Let us just see how this works through some examples right. So, supposing we first delete 65 then we first search for 65, we find it since it is a leaf then we are in this case the first case we just I have to remove this leaf and we are done.

(Refer Slide Time: 14:40)



Delete v

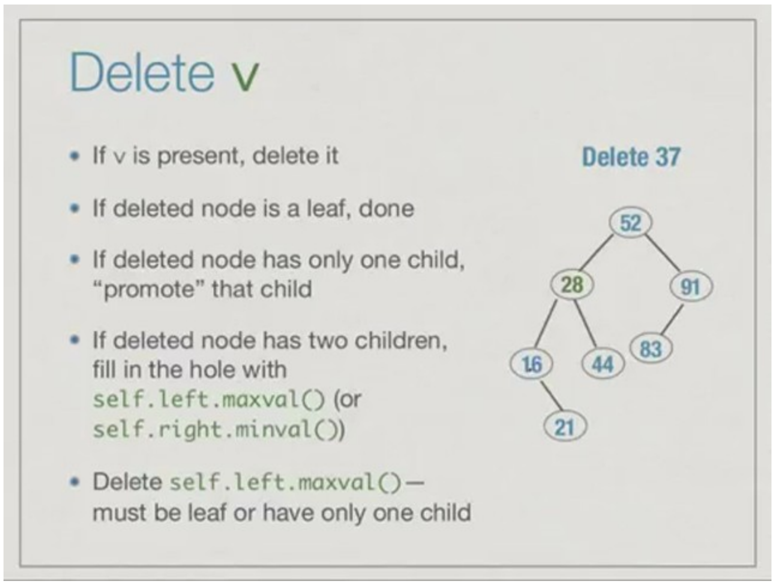
- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()`— must be leaf or have only one child

Delete 74

52
37
91
16
44
83
28
21

Now, we try to delete 74. So, we find 74 and we find that it has only one child. So, if it has only one child then we move this out then we can just effectively short circuit this link and move this whole thing up and make 52 point to 91 directly. So, we are in this second case this is what it means to promote the child. So, the deleted node has only one child we can bypass the child and directly connect the parent of the deleted node to the one child of the deleted node. So, this will result in 91 moving up there.

(Refer Slide Time: 15:14)



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 37

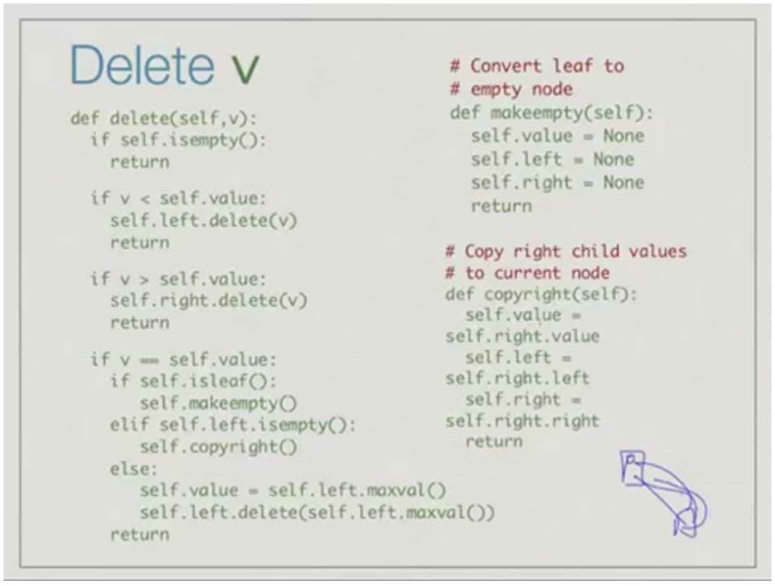
52
28
91
16
44
83
21

Now, finally, we have the difficult case which is you want to delete a node which is in the middle of the tree, in case this case 37. So, we come to 37 and we want to remove this. Now, if we remove this there will be a vacancy now, what do we fill the vacancy again and how do we adjust the shape of the tree. So, we look to the left and find the maximum value remember that everything to the left is smaller than 37 and everything to the right is bigger than 37.

So, among the left nodes we take the biggest one and we move it there then everything to the left will remain smaller than that node you could also do it the other way and take the smallest values from the right, but you would not do that you will stick to taking the maximum value from the left. We go to the left and find the maximum value is 28. So, basically we have taken this 28 and moving it up there

Now, we should not have two copies of 28. So, we need to remove this 28. So, how do we do that well within this subtree, we delete 28 now this looks like a problem because in order to delete a node we are again deleting a node, remember that the way that the maximum value was defined it is along with the right most path. So, the right most path will either end in leaf or it will end in the node like this which has only one child and we know that when we have a leaf or only one child we are in the first two cases which we can handle without doing this maximum value that. So, we can just walked out remove the 28 and promote the 21. So, this is exactly how delete works.

(Refer Slide Time: 16:37)



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
```

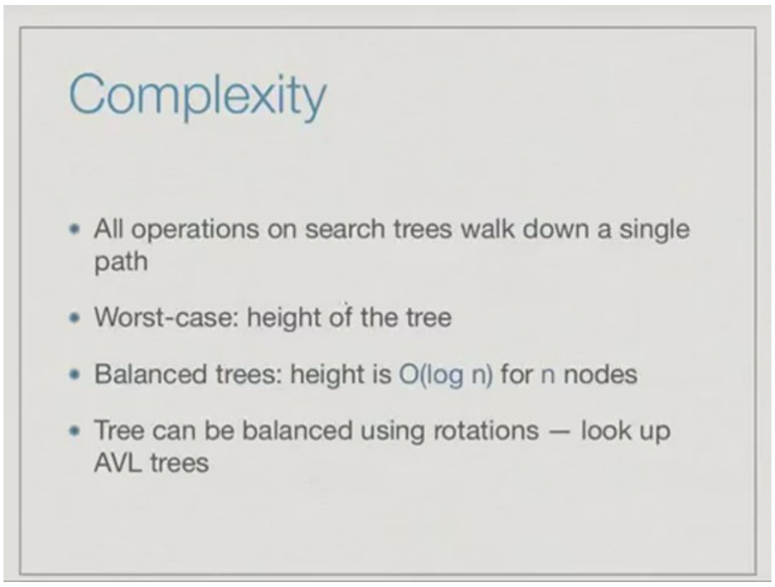
We can now look at the function. If there is no value v then we just return the easy cases are when we do not find we are the current thing. So, if it is less than the current value then we go to the left and delete if it is bigger than the current value we go to the right and delete. So, the hard work comes then we actually find v at the current value. If this is a leaf now we have not shown, how write this function if this is a leaf this means that it has left and right child both as empty nodes if this is a leaf then we will make it empty we will see how we do this in a minute I will just show you the code for this.

So, if this is a leaf we delete it right this is the first case is simple case this is the leaf we just delete it and we make this node empty if on the other hand it has only one child, if. So, actually in this case if the left is empty then we just promote the right and if it is left is not empty then we will copy the maximum value from the left and delete the maximum value on the left.

We need to just see these two functions here make empty and copy right. So, make empty just says convert this into an empty node an empty node is one which all three fields are none. So, we will just say self dot value is none self dot left is none self dot right is none. If it had an empty node hanging of it those empty nodes are now disconnected from return.

This is this make empty function and now the copy right function just takes everything from the right and moves it up. It takes the right value and makes it the current value the left value right dot left and makes with the left. So, we just take basically this node and copy these values one by one here. So, we copy right dot value to the current value right dot left to the current left right dot right to the current right.

(Refer Slide Time: 18:36)



Complexity

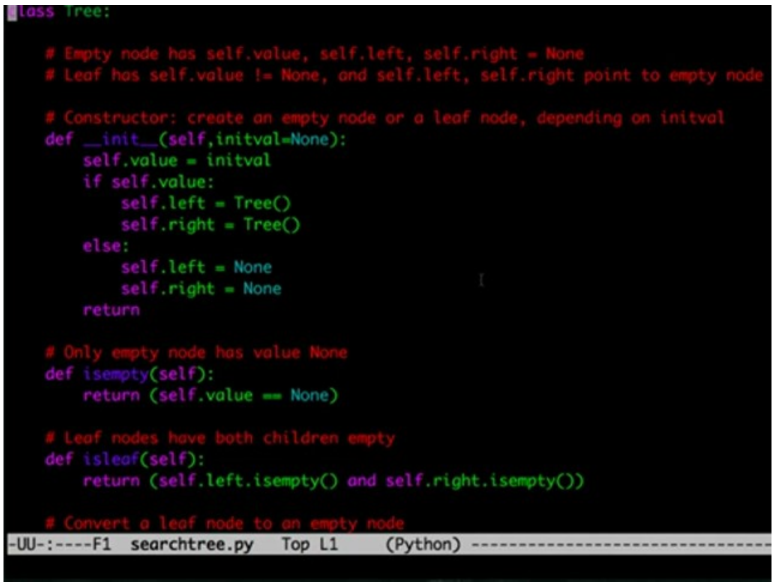
- All operations on search trees walk down a single path
- Worst-case: height of the tree
- Balanced trees: height is O(log n) for n nodes
- Tree can be balanced using rotations — look up AVL trees

So, how much time do all these take well if you examined the thing carefully you would realize that in every case we are just walking down one path searching for the value and along that path either we find it or we go down to the end and then we stop. So, the complexity of every operation is actually written by the height of the tree if we have a balanced tree a balanced tree is one where you can define that each time we come to a node the left in the right chair roughly have the same size.

If we have a balanced tree then it is not difficult to see then we have height logarithmic in an this is like a heap a heap is an example of a balanced tree now search tree will not be has nicely subset of heap because we will have some holes, but we will have a logarithmic height in general we will not explain how to balance a tree in this particular course we can look it up you can look for topic called AVL trees which is one variety of balanced trees which are balanced by rotating sub trees. So, it is possible while doing insert and delete to maintain balance at every node and ensure that all these operations are logarithmic.

Let us just look at the code directly and execute it and convenience ourselves that all the things that we wrote here actually work as intended.

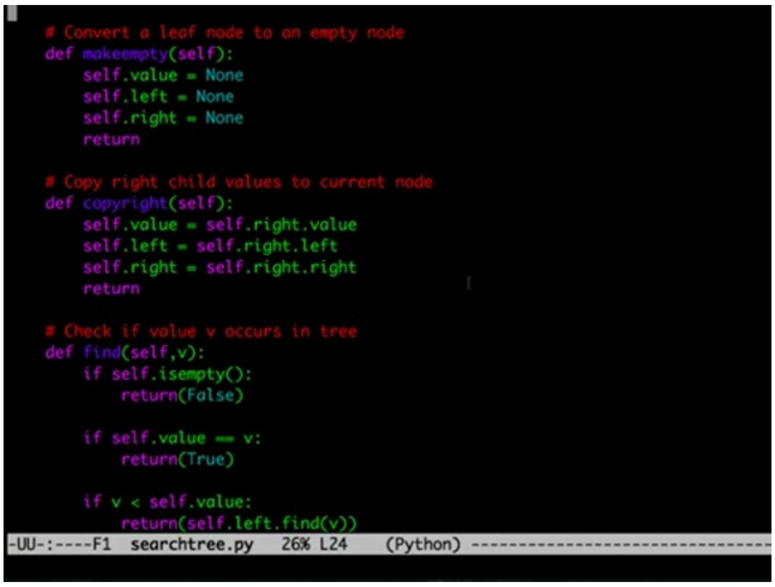
(Refer Slide Time: 19:55)



```
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
-UU-:----F1  searchtree.py   Top L1     (Python) ------------------------------
```

Here we have a python code for the class tree that we showed in the lectures. So, we are just added a comment about how the empty node is organized and the leaves are organized. So, there is the constructor which sets up either an empty node or a leaf node with two empty children then we have isempty and iseleaf we check whether the current value is none or both the left and right children are empty, respectively.

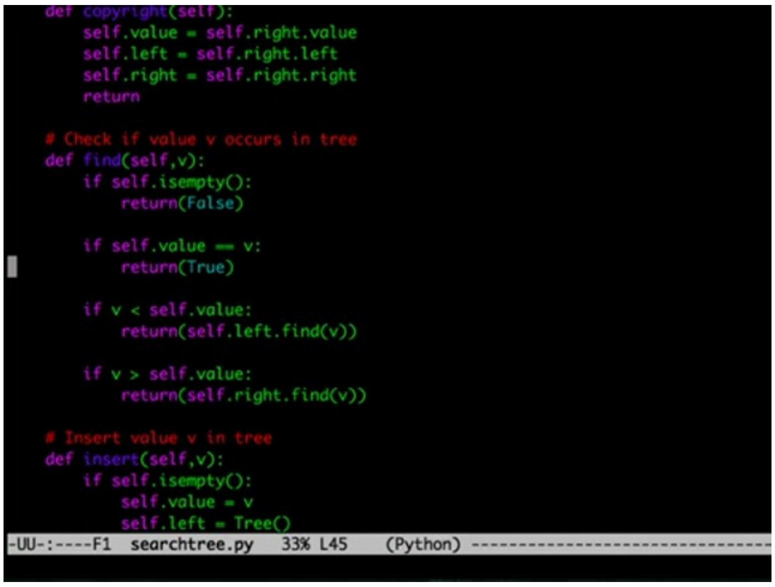
(Refer Slide Time: 20:26)



```
    # Convert a leaf node to an 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

    # 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))
-UU-:----F1  searchtree.py   26% L24     (Python) ------------------------------
```

Then we have this function makeempty which converts the leaf to an empty node copyright, copies a right child values to the current node and then we have the basic recursive functions.

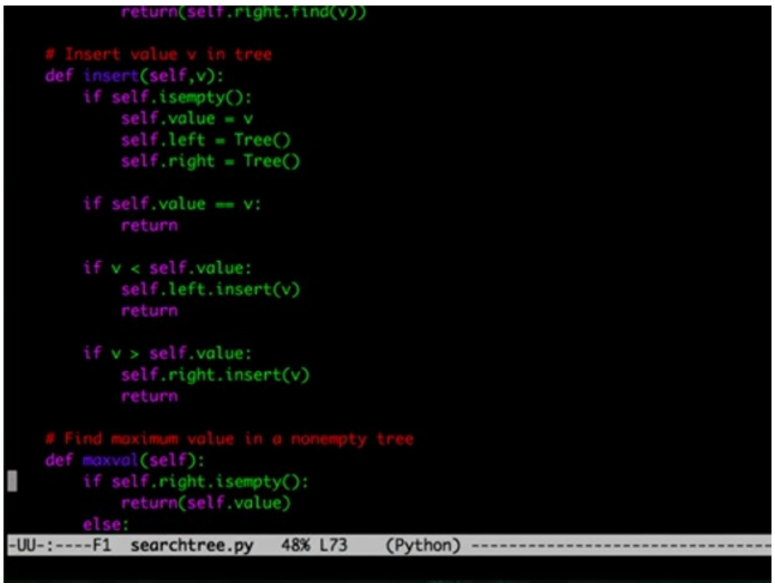
(Refer Slide Time: 20:39)



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

# 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))

# Insert value v in tree
def insert(self,v):
    if self.isempty():
        self.value = v
        self.left = Tree()
-UU-:----F1  searchtree.py    33% L45    (Python) ------------------------------
```

So, we start with find. So, find is the one which is equivalent to binary search then insert is like find and there it where it does not find it tries to insert.

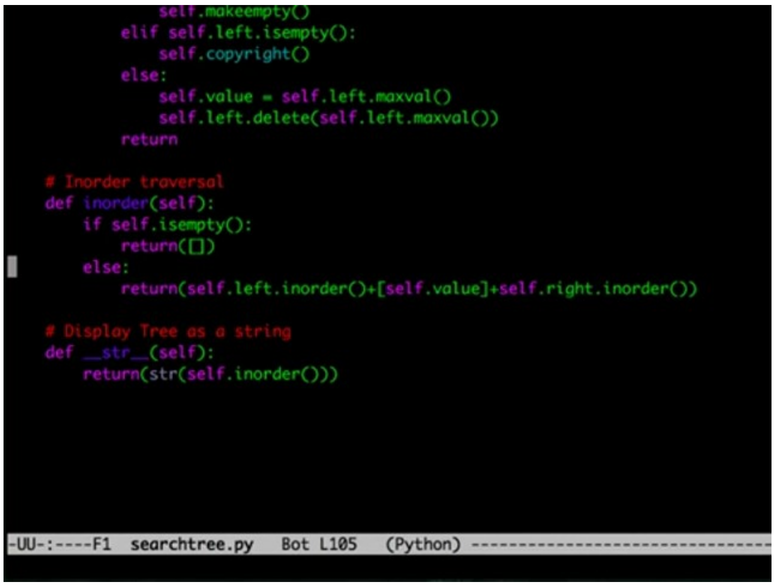
(Refer Slide Time: 20:48)



```
        return(self.right.find(v))

# Insert value v in 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 maximum value in a nonempty tree
def maxval(self):
    if self.right.isempty():
        return(self.value)
    else:
-UU-:----F1  searchtree.py    48% L73    (Python) ------------------------------
```

And finally, we have maxval which we made for delete and delete now when we reach the situation where we are found a value to that needs to be deleted if it is a leaf then we remove the leaf and make it empty if it is the left child is empty then we copy the right child up otherwise we delete the maximum from the left and promote that maximum value to a current.

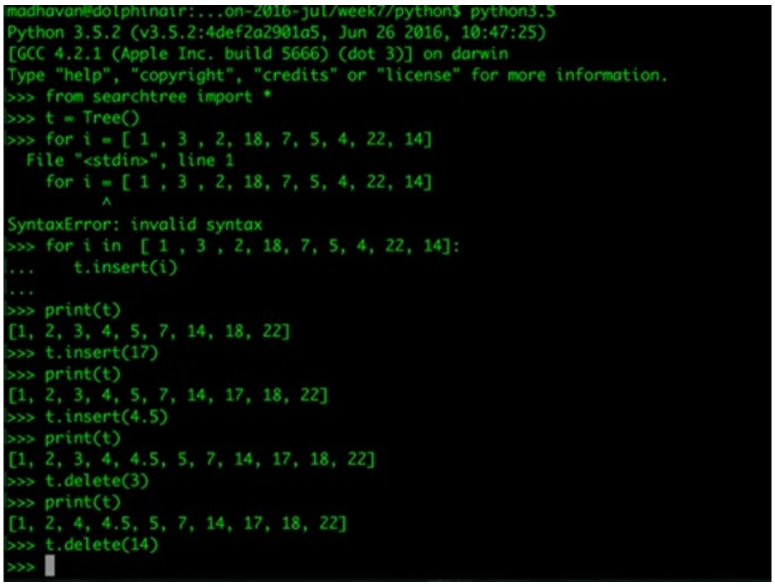
(Refer Slide Time: 21:12)



```
            self.makeempty()
        elif self.left.isempty():
            self.copyright()
        else:
            self.value = self.left.maxval()
            self.left.delete(self.left.maxval())
        return

    # 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()))




-UU-:----F1  searchtree.py   Bot L105   (Python) ------------------------------
```

Finally, we have this inorder traversal which generates a sorted list from the tree values and the str function just displays the inorder traversal.

(Refer Slide Time: 21:22)

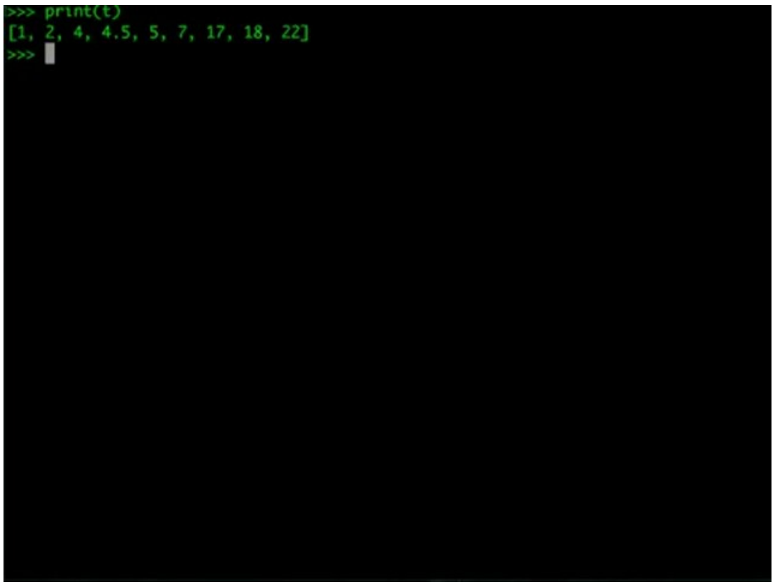


```
madhavan@dolphinair:...on-2016-jul/week7/python$ python3.5
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from searchtree import *
>>> t = Tree()
>>> for i = [ 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]
>>> t.delete(14)
>>> 
```

Now if we go to this then we can for instance import this package set up an empty tree and then put in some random values it is important not put in sorted order, otherwise a sorted tree if you just insert one at a time it will just generate one long path. So, I am just trying to put it in some random order. We insert into the tree all these values and now we are print t give me a sorted version of this. So, 1, 2, 3, 4, I can now insert more values.

So, I can for random insert 17 and verify that now 17 is there before 14 and 18 and I can keep doing this, I can insert I can even insert values in between like 4.5 because I have not in specified the integers right. So, it puts a between 4 and 5 and so on and now I can delete, for example if I delete 3 then I find that I have 1, 2, 4. If I delete, for example, 14 then I have no longer 14 between 7 and 17 and so on.

(Refer Slide Time: 22:45)



```
>>> print(t)
[1, 2, 4, 4.5, 5, 7, 17, 18, 22]
>>> 
```

This incrementation definitely works, although we have a balanced it. If we do not balance it then the danger is that if we keep inserting and sorted sequence then we keep inserting larger values it keeps adding on the right child. So, the tree actually looks like a long path right. Then it becomes like a sorted list and every insert will take order n time, but if we do have rotations built in as we do, we could be using an AVL tree then we can ensure that the tree never grows to height more than log n.

So, all the operations insert, find and delete they always be logarithmic respect to the number of values currently being maintained.