



IIT Madras

ONLINE DEGREE

Programing Data Structures and Algorithms using Python
Professor Madhavan Mukund
Designing a Flexible List and Operations on the same

(Refer Slide Time: 00:10)

Lists

- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
- "Linked" list
- Easy to modify
 - Inserting and deletion is easy via local "plumbing"
- Flexible size
- Need to follow links to access $A[i]$
- Takes time $O(i)$

Madhavan Mukund Designing a flexible list

So, we talked about flexible lists, which can expand and contract, I mean, which are like a train. So, they have this kind of linked structure. So, let us try and understand, how we might actually program such a list. So, a list is now a sequence of nodes. Each node consists of a value, and something that points the next node.

And we argued that using this plumbing operation, it is easy to insert and delete locally. But in order to navigate to a particular point in the list, you will have to start from the head each time, so it is going to take you some time to reach that point. So, the typical way to represent these is as collections of these nodes. So, we need something which represents one node in the list, and then we will string them together.

(Refer Slide Time: 00:52)

Implementing lists in Python

■ Python class `Node`

■ A list is a sequence of nodes

- `self.value` is the stored value
- `self.next` points to next node

```
class 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)
```

Handwritten notes:

- A box with `None` and `None` inside, with `v` and `next` written next to them.
- `Empty list`
- `l = []`
- `l.append(0)`

Madhavan Mukund Designing a flexible list

Implementing lists in Python

■ Python class `Node`

■ A list is a sequence of nodes

- `self.value` is the stored value
- `self.next` points to next node

■ Empty list?

- `self.value` is `None`

■ Creating lists

- `l1 = Node()` — empty list
- `l2 = Node(5)` — singleton list

```
class 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)
```

Handwritten notes:

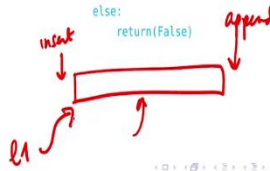
- A box with `5` inside, with `l2` written next to it.

Madhavan Mukund Designing a flexible list

- Python class `Node`
- A list is a sequence of nodes
 - `self.value` is the stored value
 - `self.next` points to next node
- Empty list?
 - `self.value` is `None`
- Creating lists
 - `l1 = Node()` — empty list
 - `l2 = Node(5)` — singleton list
 - `l1.isempty() == True`
 - `l2.isempty() == False`

```
class 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)
```



Madhavan Mukund

Designing a flexible list

So, we define a class `node`, and this class `node` will have two parts. So, it will have one part which stores the value, so let us just call itself dot `value`. So, remember, this object-oriented thing that we discussed in the very first week. So, each object refers to itself through this name, `self`. So, `self` dot `value` is my value field, and `self` dot `next` is my name for the next node in the list.

So, if there is no next node I will just use the default Python value `none`. So, this is the initialization. So, when I initialize a list, I pass a value, I set the value to that list. And initially, this list has only one node, so the next is always going to be `none`. So, one of the important things we have to deal with is this notion of an empty list. So, when we write something like in normal Python, if we want an empty list, we have to write this. Because if we do not write this, and later on we try to add something to it.

Supposing I want to start with a list with the value 0, and I write `l` dot `append` 0, then Python will say, I do not know why you are allowed to use `append` on `l`, because you have not told me it is a list. So, we always need to initialize value to an empty data structure, data type of that value in order to tell Python, what is the legal operation allowed on that? So, this initialization is important.

Now, the problem with Python is that there is no way to tell it, other than to initialize it. There is no way to say `l` is a list. So, other programming languages, if you are familiar with programming languages, like C or C plus plus or Java, whenever you use a variable or a name, you first have to tell the compiler which type of value it is going to hold. So, you have to say I am using `i` and is going to be an `int`, I am using `l` and it is going to be an array.

So, this information is there independent of the value that you store, so there is a separate piece of meta data, as it is called about your variables, which is known in advance. Now in Python, we do not have that. So, we have to be able to represent these empty lists. So, in our context, think of a node as a box with two parts. So, this is our value and this is our next. So, we know that if it is a singleton, the next is none.

So, what we will use as a convention is that if the value is also none, then this is an empty list. The only situation in which you could have a node, which has no value is when it is the first node of an empty list. And if it is the first node of an empty list, by definition, there are no other nodes. So, this gives us this motivation for this function. So, this says, is the list I am looking at empty or not. So, it says, if the value is none, yes, it is empty return true.

If the value is not none, then it cannot be empty, return false. And if the value is none, implicitly, the next must be none. We will never ever check the next if the value is none because we will believe it is an empty list. So, the empty list is one where self dot value is none. So, the way we would use this, remember, is to just call this node class, as we would call a function with an argument, which is optional, because we have this default value.

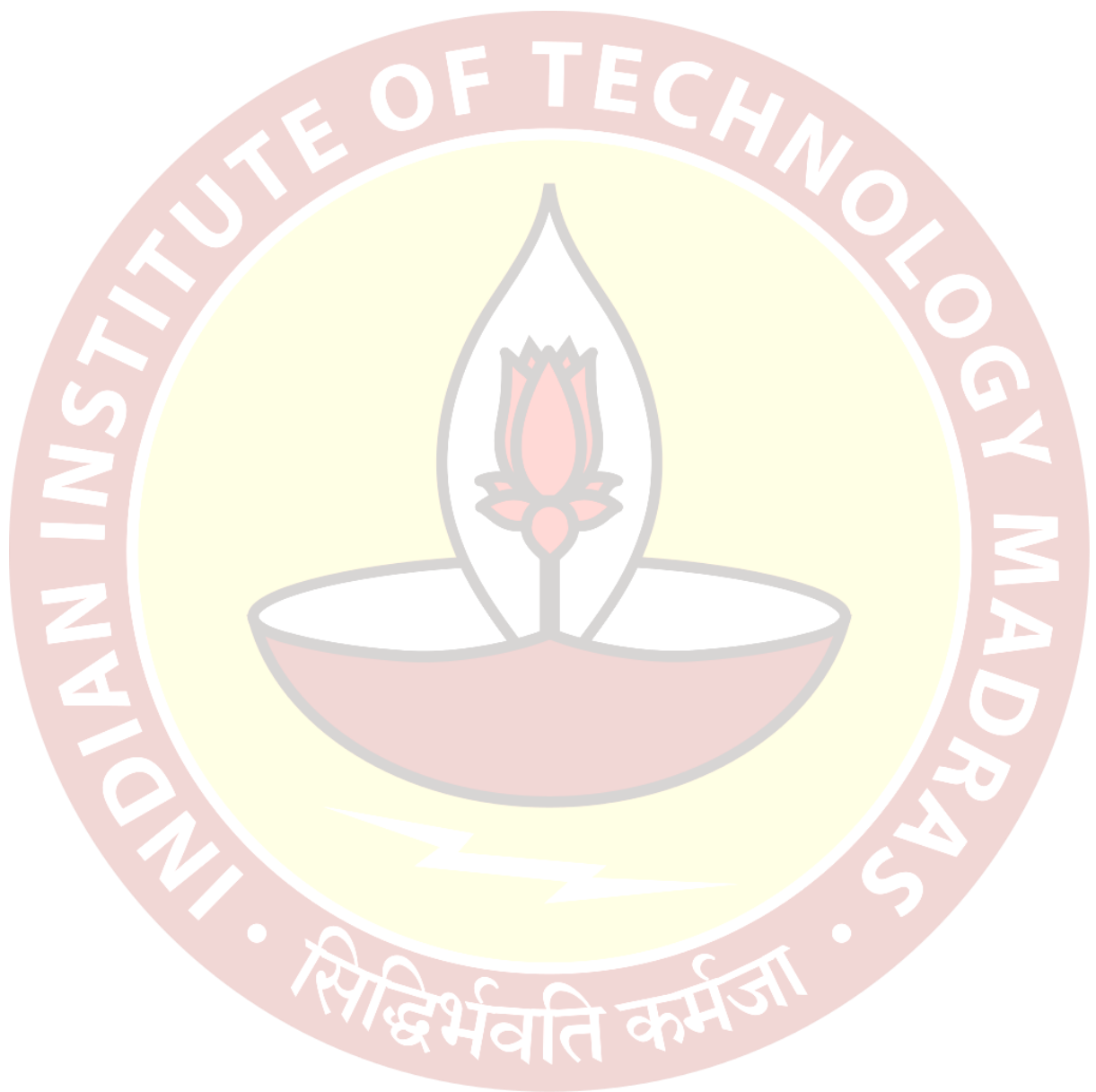
So, if we call node with no argument, we get an empty list. The self dot V by default is set to the value not none. On the other hand, if you call it with a concrete argument like a 5, then we will get a node containing this is in Python notation, this would lead the list 5, and this would be the empty list. So, this is what we are saying.

So, if I now for instance, check it, then it will say that l1 is empty is true and l2 two is empty is false, because l1, if I look at it self dot values none, l2, if I look at it self dot values, not none. So, this is our basic building block. So, this is how we build up a list. Now we have to think about how we will actually grow and shrink these lists.

So, in a list, there are many different ways you could define the operations, but the most conventional thing to do with a list is to take a list and either add at the end, which we call append or insert at the beginning. Now you can modify it to same, I mean, you can apply the same principle to insert anywhere in the middle also, but the two interesting cases are append and insert and they have different properties.

Actually, append turns out to be much easier than insert, as we will see. So, typically, what we have through this kind of thing is, say l1 is pointing to the first node in the list, the first

node is pointing to the next node, and so on. So, what we want to say is take the node point the list pointed to by 11, and append a value V to it.

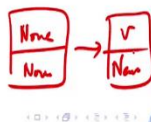


(Refer Slide Time: 05:30)

Appending to a list

- Add v to the end of list l
- If l is empty, update $l.value$ from $None$ to v
- If at last value, $l.next$ is $None$
 - Point $next$ at new node with value v
- Otherwise, recursively append to rest of list

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




Madhavan Mukund Designing a flexible list

Appending to a list

- Add v to the end of list l
- If l is empty, update $l.value$ from $None$ to v
- If at last value, $l.next$ is $None$
 - Point $next$ at new node with value v
- Otherwise, recursively append to rest of list

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



Madhavan Mukund Designing a flexible list

So, we want to add a value at the end of the list. So, what are the possibilities? So, the first possibility is that there is nothing in the list, the list is empty. So, if the list is empty, then we basically take this list currently looks like none, none and we change it to the list looks like V none. So, we just update the value field to v , and we leave the next field unchanged and we are done.

Now, if we are at the end of the list, the last node will have some value, and it will say, there is no, there are no more nodes in this. So, in this case, what we do is, we create a new node. So, we call this node constructor, as it is called. We create a new node, which will be the

form V comma none because we need a node, which has the value V that we're trying to append, and we set this to point there.

So, self dot next, if self dot next is none, which means, I am at the end of the list, then instead of making it none, I make it point to a new node that I create now. I create a node with the value V and I make this node point to that. And if I am in the middle of the list. So, supposing I am here, and it says that there is, there are some more nodes to my right then I just postpone the work by saying that, appending to this node is the same as appending to the next node, so I just say, append on the next node.

So, if I say l dot append I am appending on the first node, the first node has an X node, it will say, okay, append to the second node. Finally, we'll keep walking down in this recursion will end at the last node, the last node will say, why have no next node, and then it will create the node and append it. And there is this base case where the initial node itself was empty, where I just update, I do not create a node, I just change the value from none to v.

So, this is a very straightforward recursive implementation of append. It has three cases. Am I empty? Am I the last node or if I am not the last node then recursively call? So, in this case, it is quite easy to think of how you would do an iterative implementation, essentially, you do the same thing. I check it is empty. If it is not empty, I keep following this next pointer.

I can create a loop, which says, I keep going to next, next, next until next is not then I know I am at the last node and then I do the same thing. So instead of using recursion to go this way, I can use a loop saying while the next pointer is available, go to the next pointer.

(Refer Slide Time: 08:04)

- Add v to the end of list l
- If l is empty, update $l.value$ from $None$ to v
- If at last value, $l.next$ is $None$
 - Point $next$ at new node with value v
- Otherwise, recursively append to rest of list
- Iterative implementation
 - If empty, replace $l.value$ by v
 - Loop through $l.next$ to end of list
 - Add v at the end of the list

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

```
temp.next = Node(v)
return
```

Navigation icons: back, forward, search, etc.



Madhavan Mukund

Designing a flexible list

So, here is an iterative implementation. So, if it is empty, replace the value from none to v , otherwise, we loop through this next until we find the last node. How do we find the last node? Next is none. And then at that point in the same way as we did before, we append. So, here is the thing. So, if it is empty, we do the same thing. If it is empty, we just change the value to V and return, otherwise, we have to start here and we have to keep walking down.

So, we use a new name to do this walking, so we call it $temp$. So, we first point $temp$ the first thing. Then if $temp$ has a next, then we will move $temp$ to the next, so $temp$ will become $temp$ dot next. So, $temp$ will keep moving from one node to the next, so long as the next is defined. It will exit from this when it reaches the last node. So, when I have here and I reached $temp$, supposing this is the last node, then it will have no successor.

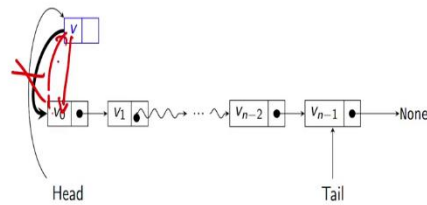
So, when $temp$ dot next is none this while loop exit. So, now I know, the $temp$ is pointing to the last node in my list. At this point, I just create a new node V as before and I make $temp$ dot next point to it and I am done. So, this is an iterative version of that append. So, both of these are very simple, because we do not have to do anything except for in some sense go to the end of the list and add something at the end.

(Refer Slide Time: 09:16)

Insert at the start of the list



- Want to insert v at head
- Create a new node with v
- Cannot change where the head points!



◀ ▶ 🔍 ↺ ↻

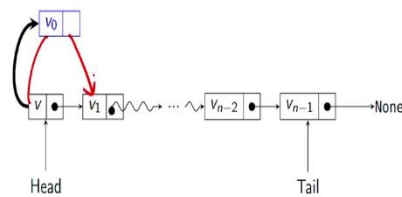
Madhavan Mukund

Designing a flexible list

Insert at the start of the list



- Want to insert v at head
- Exchange the values v_0, v
- Create a new node with v
- Cannot change where the head points!



◀ ▶ 🔍 ↺ ↻

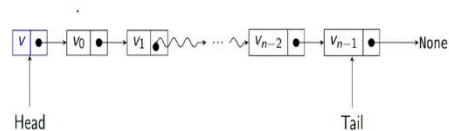
Madhavan Mukund

Designing a flexible list

Insert at the start of the list



- Want to insert v at head
- Exchange the values v_0, v
- Create a new node with v
- Make new node point to head.next
- Cannot change where the head points!
- Make head.next point to new node



◀ ▶ 🔍 ↺ ↻

Madhavan Mukund

Designing a flexible list

For the other operation, as I said is insert. I want to insert not at the end of the list, but at the beginning, so I call that append, I call this insert. So, I have a value V, which I want to stick I want to put this here. So, what is the logical thing to do? The logical thing to do, is to say, that I made this head point there and V point here.

Now, unfortunately, this is a problem. Because inside a function, when I call an object, it is like calling a function. If I call a function with the value l which is like this, something which can change, but if I update it inside then it gets lost. So, if I, it will get create, I will lose this list. I cannot take the head and point it inside the function change what is pointing to it.

It is the same as if I pass an l to a list processing function and I reassign l inside, then the l inside becomes different from the l outside, so outside nothing changes. So, the same will happen here. If I go inside this thing and I change where head is pointing to, then the old head will remain where it was and this whatever changes I make will happen inside the function, but will not happen outside. So, this insert will have no global effect. So, this is not allowed.

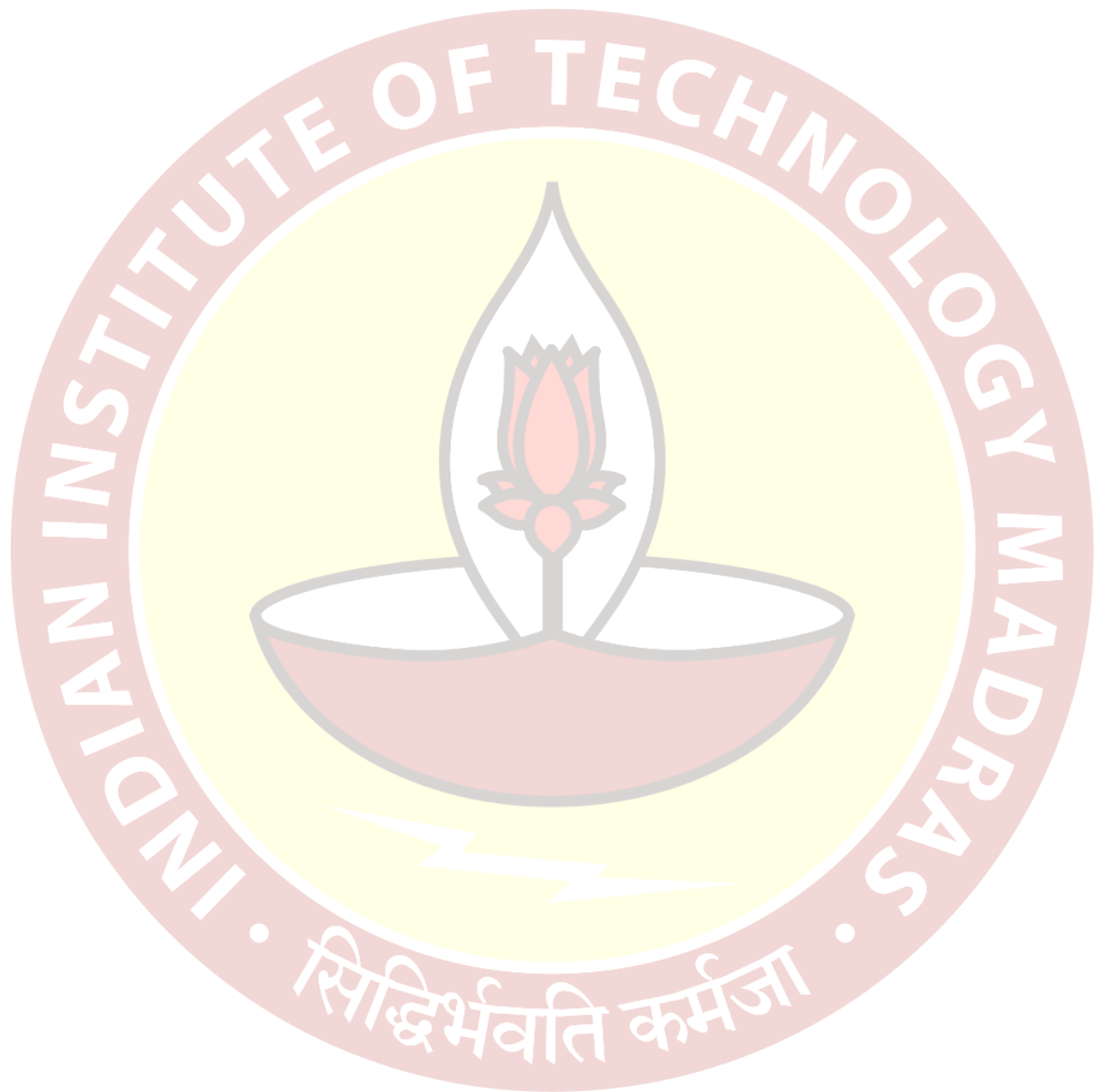
So, this would be the easiest solution. Just stick a new node at the beginning and point to that instead of the first node it that, but that is not a lot. So, now what do we do? So, what we do is, we have to do something which is kind of not obvious initially, which is, we have to so, what is the status of this node?

Remember that in a linked list, the physical position of a node makes no difference only the logical position matters. So, what we want to do is make this new value V logically come before V0, but we for us, the first value you point to is always in this node. So, the first place I point to must contain V. So, I need to move this V here. But if I move this V here, then where does the V0 go? Well, I have made space for it, so I will move this V0 there.

So, what we can do instead is we can exchange the positions of V and V0. So, I create a new node, I have a new node, which I want to add, which has a value V, and I have a list with such as V0, so I just swap the values. So, the nodes are still where they are. So, my head is now pointing to a new list, the same old list rather, which starts with the new value V and then goes to V1 , and V0 is now sitting up there.

But now I can do this plumbing, I can make V0, V point V0 and V, so I can make this point there and this point here. So, once I do that, then I am done. So, this is how we insert at the

beginning of the list. So, what we do is we swap the value of what head is pointing to with the value we want to exchange and then we do this update.

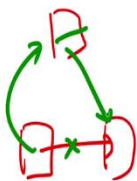



(Refer Slide Time: 12:04)

Appending to a list

- Create a new node with v
- Exchange the values v_0, v
- Make new node point to `head.next`
- Make `head.next` point to new node

```
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)  
    # Switch links  
    (self.next, newnode.next) =  
        (newnode, self.next)  
    return
```





Madhavan Mukund

Designing a flexible list

So, here is some code for that. It is very straightforward it is not very complicated. So again, if it is empty, inserting into an empty list is equivalent to appending it to an empty list. If I have an empty list, it is the first and the last element of the new list. So, I just update the value to V . Otherwise, I create a new node, which I am going to add to my list at the beginning, but I cannot add it directly.

So, what I do is, I first exchange this value and the value of `self`. So, `self`, remember, is pointing to the beginning of the list. So, `self` is the zeroth value in my old list. So, I exchange the value of the old list head and the new node that I have created and then I do the swapping of links. So, I make the new node point to the next of the old list, and the new thing next point to the old. So, I take `self.next` and make it point to the new node, and I take `newnode.next`.

So basically, I had this link, and I have this new node here, so I am now going to change this. So, I am going to say that, instead of pointing here, I am going to make this point there, so that is the `self.next` is equal to `newnode`. And instead of having, I mean, right now, this is not pointing to anything, because it is a new node it was `None`, so I am going to make that point here. So, I do this plumbing, and I am done. So, this is my insert.

(Refer Slide Time: 13:19)

Delete a value v

- Remove first occurrence of v
- Scan list for first v — look ahead at next node
- If next node value is v , bypass it

Madhavan Mukund Designing a flexible list

Delete a value v

- Remove first occurrence of v
- Scan list for first v — look ahead at next node
- If next node value is v , bypass it
- Cannot bypass the first node in the list
 - Instead, copy the second node value to head
 - Bypass second node

Madhavan Mukund Designing a flexible list

What about delete? So, now delete is more interesting when you actually tell me what value to delete. When I insert a value, it is I said that we will look at the case where the insertion happens at a fixed place, either the beginning or the end. The end is append, the beginnings insert, but deleting the first value of node is not typically, of a list is not typically what we want. What want to say is, I give you a value V , please delete it from the list.

Now, deleting from a list can mean many things. It could mean, delete all the values or delete the first value or delete the last value or delete some value, so let us just for the sake of concreteness, say that when we say delete a value V , we mean, remove the first occurrence of

V. And in particular, if there is no V in the list, then it will just do nothing, right. So, deleting a V from a list that does not contain a V does not create an error it merely does nothing.

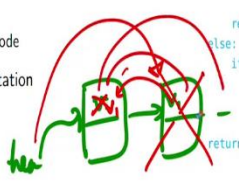
So, what we have to do is we have to go through the list and look for the first V, and then we have to delete it. Now, the problem is that, I do not know if you are seen, but old style people demolishing a house they will, you find these people who are workers who are standing on a wall, and they will have a hammer and they will be hammering the wall below them. And I have always wondered how they know when the wall is going to be safe enough to stand on.

And when they have to jump off and then finish the job standing, but it is kind of easier to hammer down like this. So, in the same way, if you are sitting on the node, which has a V and you want to delete it, it is kind of difficult. So, if you are sitting here and you want to delete this V, this is not a good way to do. So, what you want to do is actually sit here and look ahead and see is the next node that I want to delete.

And the next node is the one that I want to delete then I will make myself point to that node, so I can bypass it. So, this bypassing is basically just a way of saying that my next pointer points to the node with V. Instead, I make it to point to the node after that. And how do I know that because the node with V tells me where the next node is.


(Refer Slide Time: 15:26)

Delete a value v



- Remove first occurrence of v
- Scan list for first v — look ahead at next node
- If next node value is v, bypass it
- Cannot bypass the first node in the list
 - Instead, copy the second node value to head
 - Bypass second node
- Recursive implementation

```
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
```



Madhavan Mukund Designing a flexible list

Delete a value v

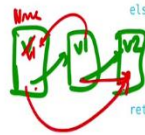


- Remove first occurrence of v
- Scan list for first v — look ahead at next node
- If next node value is v , bypass it
- Cannot bypass the first node in the list
 - Instead, copy the second node value to head
 - Bypass second node
- Recursive implementation

```
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
```



Madhavan Mukund

Designing a flexible list

So, let us look at this code. So, this is the bypass. So, what we say is, that I want to delete the value V . So, if I, if the current value is V , so this is a difficult case that we will come to. So, the recursive case just says that, if the next, if I am not deleting it here, then I delete the V from the next point onwards. So, but so the difficult cases really now that we are pointing from the head to a node containing V , and then $V1$, and then so on.

So, this is the thing where I want to delete this node, we have the same problem as before with insert, which is I cannot make this head point there directly, because I cannot change the node that the head is pointing to. So, it is exactly the same problem when we said when you insert the beginning, we would like to make the head point to the new node and then come down to the old node, but we cannot do that. So, what did we do there? We exchanged.

We exchange the two values and then made this, the new node actually the second node in the list by making this, the zeroth node point to that. So, we can do the same thing here. So, what we do is, instead of trying to delete this, we move this $V1$ here, so logically, there is no V anymore. And then we delete this. So, that is what is happening here.

So, we say that if I want to delete the value that I am pointing to, then what I do is, I check. I mean, there may be no next node, maybe this is a singleton, in which case, I just set it to none. But if I have a next node, then I copy the value from the next node and now I bypass, so this is bypass. So, `self dot next`, my next is pointing to my next, next.

So, whatever the next node says is, the next is the node that is two steps down, is what I am going to point to. So, this is a simple recursive version. So basically, it has now three cases.

Either the, I have reached the end of the list and there is nothing to delete. So, I have an empty road list and nothing is there, I just get out. Otherwise, if I have to delete the value here, then I first deleted, so I just change this value to none.

And now I check whether there was something after me. Because I cannot remember, if I cannot have value none unless it is the empty list. So, if it is not the empty list, I have to do some work. So, in that case, what I have to do is, I have to copy this value and then I have to bypass. So, to draw it again.

So, basically, if I have let us say I have three nodes and this is my V and this is V1 and this is V2, then what I will do is, I will first set this to none, then I will copy V1 here, and then I will say self dot next dot next is this this thing. So, I will say move this to there, so that is the bypass. So, this is a recursive implementation. And as an exercise, you can write an iterative version just to check that you understand what is going.

(Refer Slide Time: 18:18)

Summary

- Use a linked list of nodes to implement a flexible list
- Append is easy
- Insert requires some care, cannot change where the head points to
- When deleting, look one step ahead to bypass the node to be deleted

Madhavan Mukund

Designing a flexible list

So, what we have seen, is that, by taking this class node, which contains a value and a next pointer, and we can string it together like a train and make a flexible list. So, programming a flexible list is not very difficult, we can make one of our own. In this, because we start from the beginning, adding at the end of the list is quite easy, because we do not, all the difficult part happens when we have some changes to make at the node which head points to.

So, the node that is pointed to by the first name of the list is the one that is tricky to handle. So, append is easy, insert requires some care. And when deleting basically what we do is,

effectively we look one step ahead, and so we try to delete the value at the next node by bypassing it. But then if we are doing it recursively it amounts to then coming to the node and then deleting it and then we have to do something else.

If you are doing it iteratively you will look at the next node and then bypass. So, this is what a flexible list looks like. But as we will see, actually the Python lists as they are implemented in the language are not these flexible list. So, all this analysis of what flexible lists do, does not really hold for Python, because of that.

