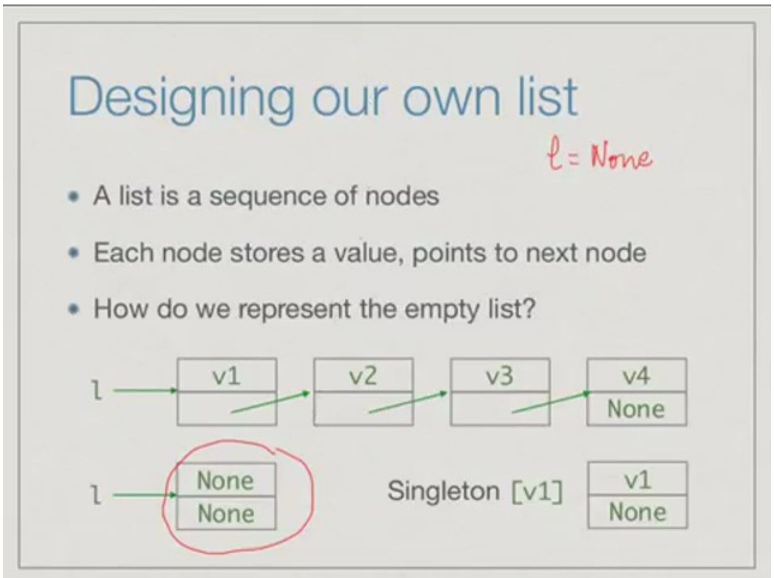
Programming, Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Chennai Mathematical Institute,Madras

Week - 07
Lecture - 02
User Defined Lists

(Refer Slide Time: 00:03)



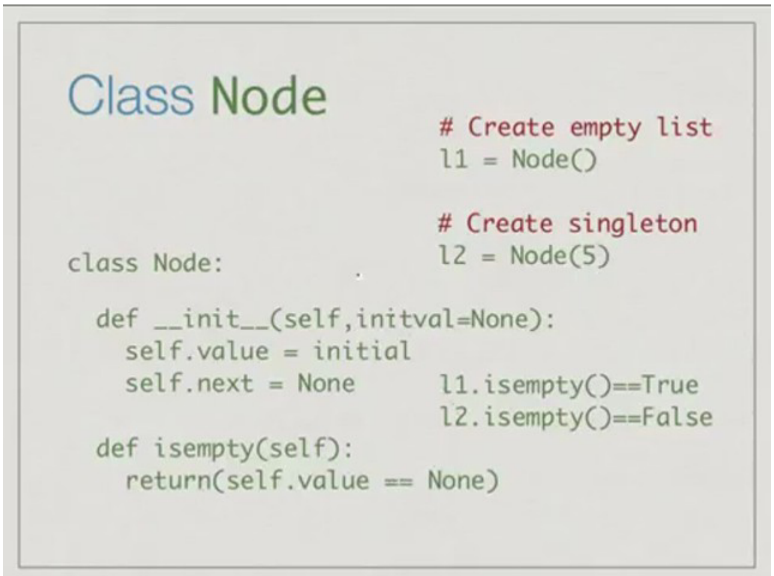
Now that we have seen the basics about how to define classes and objects, let us define an interesting data structure.

Suppose we want to implement our own version of the list, a list is basically a sequence of nodes and each node in the sequence stores a value and points to the next node. So, in order to go through the list we have to start at the beginning and walk following these pointers till we reach the last pointer which points to nothing. We have a list of the form v1, v2, v3, v4 in python notation. Then this is how we would imagine it is actually represented. There are 4 nodes. The list l itself which we have set up points to the first node in this list, v 1 points to v 2, v 2 points to v 3 and v 3 points to be v 4. The final node points to nothing and that indicates we have reached the end of the list.

In this representation what would the empty list look like well it is natural to assume that the empty list will consist of a single node which has both the value and the next node pointers set to none, whereas for instance the singleton would be a single node in which we have the value v 1 and the next set to none. So, this is the convention that we shall follow for our representation of a list. So, notice that unless we have an empty list with a single node none, none no other node in a list can have value none, right. This is something that we will implicitly assume and use that checking for the value none will tell us it is an empty list and we will never find none in the middle of a list.

We distinguish between a singleton and an empty list purely based on the value. Both of them consist of a single node. Now the reason that we have to do this is because actually python does not allow us to create an empty list if we say something like l is equal to none and we want this to denote the empty list the problem is that none does not have a type as far as python's value system is concerned. So, once we have none, we cannot apply the object functions we are going to create for this list type. So, we need to create something which is empty of the correct type. So, we need to create at least 1 node and that is why we need to use this kind of representation in order to denote an empty list.

(Refer Slide Time: 02:26)

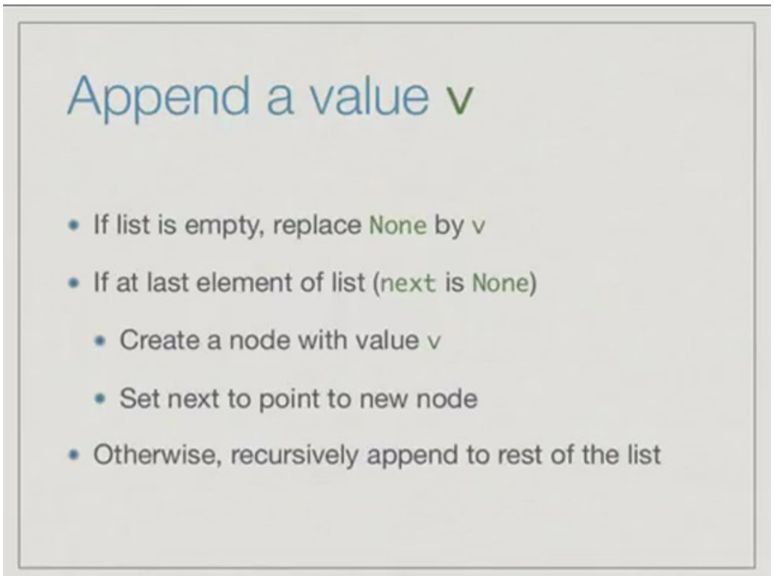


Here is the basic class that we are going to use, it is a class node. So, inside each node we have 2 attributes value and next as we said and remember that self is always used with every function to denote the object under consideration. We will use this default scheme that if we do not say anything we create an empty list. The init value, this should be init val.

The initial value is by default none unless I provide you an initial value in which case you create a list with that value and because of our assumption about empty list all we need to do to check whether a list is empty is to check whether the value at the initial node is none or not. We just take the list we are pointing to and look at the very first value which will be self dot value and ask whether it is none. If it is none, it is empty. If it is not none, it is not empty.

Here is a typical thing. We say l1 is equal to node; this creates an empty list because it is not provided any value. So, the default initial value is going to be none. If I say l2 is equal to node 5 this will create a node with the value 5. It will create the singleton list that we would normally write in python like this. If I ask whether l1 is empty, the answer will be true. If I ask whether l2 is empty, the answer will be false because self dot value is not none.

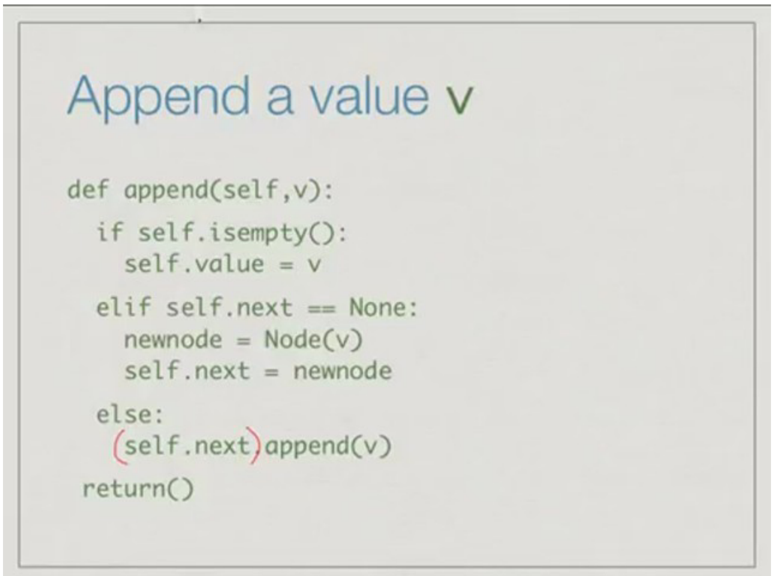
(Refer Slide Time: 03:50)



Now, once we have a list what we would like to do is manipulate it. The first thing that we might want to do is add a value at the end of the list. If the list is already empty, then we have a single node which has value none and we want to make it a singleton node, a singleton list with value v. So, we want to go from this to this, remember that in a singleton node we just have instead of none we have the value v over here so that is all we need to do. We need to just replace the none by v, if we are at the last element of the list and we know that we are at the last element of the list because the next value is none then what we need to do is create a new value.

We walk to the end of the list and then we reach none. Now, we create a new element here with the value v and we make this element point to this, we create a new element with the node v and set the next field of the last node to point to the new node and if this is not the last value then well we can just recursively say to the rest of the list treat this as a new list starting at the next element, take the next element and recursively append v to that.

(Refer Slide Time: 05:05)

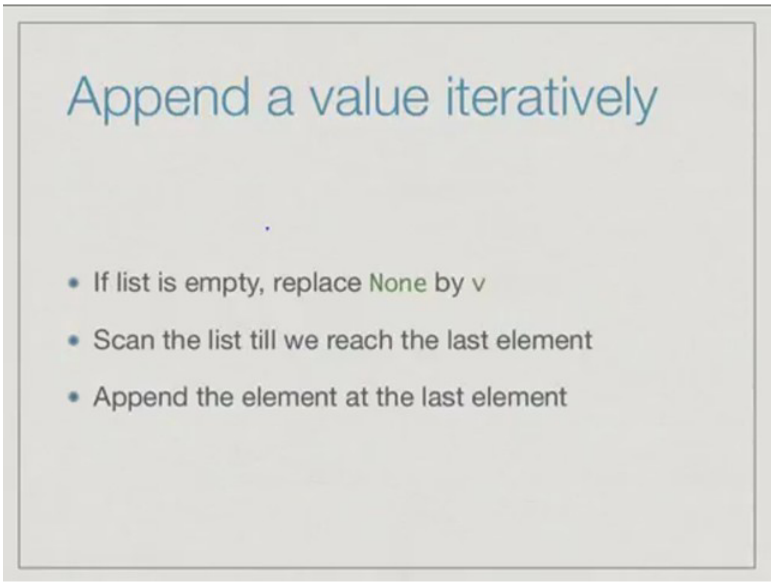


This gives us a very simple recursive definition of append. So, we take append and we want to append v to this list. If it is empty, then we just set the value to v. So, this just converts the single node with value none to the single node with value of v, otherwise if we are at the last node that is self dot next is none then we create a new node with the value v and we set our next pointer to point at the new node, remember when we create a new node the new node automatically is created by our init function with next none.

We would now create a new node which looks like v and none and we will set our next pointer to point to it and the final thing is that if it is not none then we have something else after us. So, we go that next element self dot next and with respect to that next element we reapply the append function with the value v, this is the recursive call.

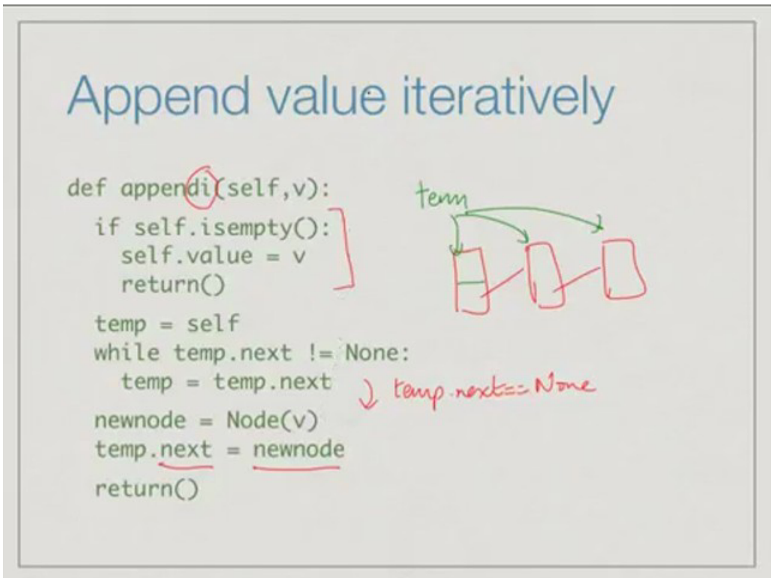
We have been abundantly careful in making sure that this is parsable. So, we have put this bracket saying that we take the object self dot next and apply append to that actually python will do this correctly. We need not actually put the bracket, we can just write self dot next dot append v and python will correctly bracket this as self dot next dot append. So, this dot is taken from the right.

(Refer Slide Time: 06:23)



Now, instead of recursively going to the end of the list we can also scan the end of the list till the end iteratively. We can write a loop which keeps traversing these pointers until we reach a node whose next is none. If the list is empty as before we replace the value none by v, otherwise we scan the list till we reach the last element and then once we reach the last element as in the earlier case we create a new node and make the last element point to it.

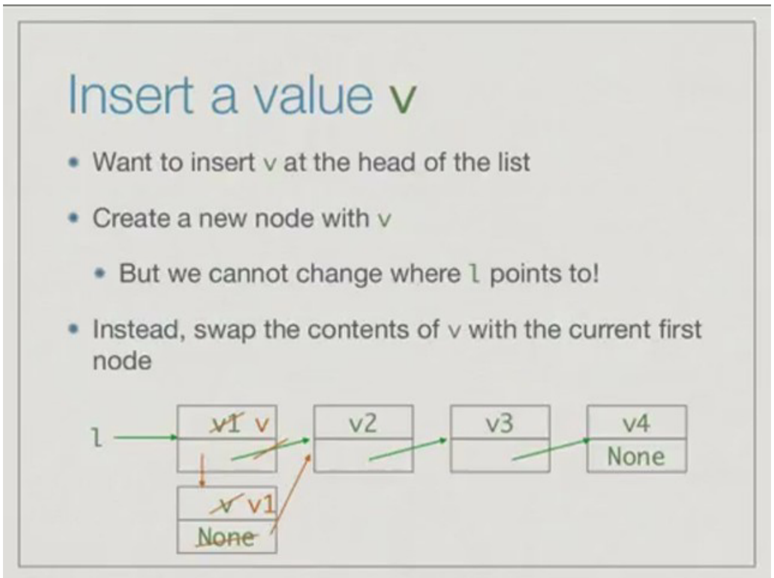
(Refer Slide Time: 06:53)



This gives us an append which is iterative. So, we call it append i just to indicate that it is iterative. The first part is the same if the current list is empty then we just set the value to be v and we return, otherwise we now want to walk down the list. We set up a temporary pointer to point to the current node that we are at and so long as the next is none we keep shifting temp to the next value. So, we just write a loop which says while temp dot next is not none just keep going from temp to temp dot next.

So, just keep shifting temp. Finally when we come out of this loop at this point we know that temp dot next is none. This is the condition under which we exit the loop. We have reached, the node temp is now pointing to the last node in the current list. At this point we do exactly what we did in the recursive case we create a new node with a value v and we make this last node point to this new node. So, we reset next of temp from none to the new node.

(Refer Slide Time: 07:57)



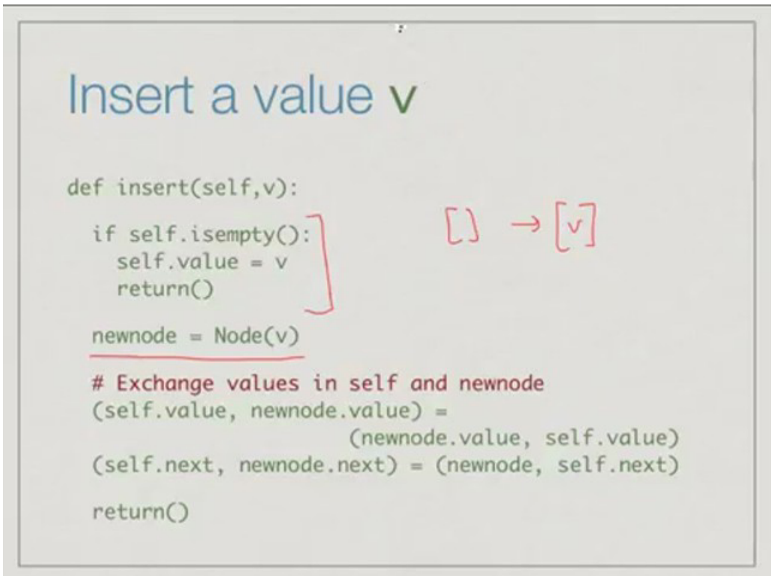
What if we do not want to append, but we want to insert. Now it looks normally that insert should be easier that append, but actually insert is a bit tricky. So, by insert we mean that we want to put a value at the beginning. We want to put a node here which has v and make this pointer.

This is what we want to do now. The problem with this really is that after we create a new node we cannot make this point here and this point here there is no problem in making the new node point to v 1, but if we reassign the value of l or inside a object if we reassign the value of self then this creates a completely different object. We saw this when we were looking at how parameters are passed and immutable value are passed.

We said that if we pass a mutable value to a function so long as we do not reassign that thing any mutation inside the function will be reflected outside the function, but if we reassign to the list or dictionary inside the function we get a new copy and then after that any change we make is off. So same way if we reassign l or self to point to a new node then we will lose the connection between the parameter we passed to the function and the parameter we get back. So, we must be careful not to make l point to this thing. We cannot change where l points to. So, how do we get around this problem? We have created a new node, we want to make l point to it, but we are not allowed to do so, because if we do so, then python will disconnect the new l from the old l. So, there is a very simple trick. What we do is we do not change the identity of the node, we change what it contains. So, we know now that v 1 is the old first node and v is a new first node, but we cannot make l point to the new first node, so we exchange the values. So, what we do is we replace v 1 by v and v by v 1.

Now, the values are swapped and we also have to do a similar thing for what is pointing where. So, l is now pointing to v as the first node, but now we have bypassed v 1 which is a mistake. We must now make the first node point to the new node and the new node point to the old second node. So, by doing this kind of plumbing what we have ensured is that the new list looks like we have inserted v before the v 1, but actually we have inserted a new node in between v and v 2 and we have just changed the links to make it appear as though the new node is second and not first.

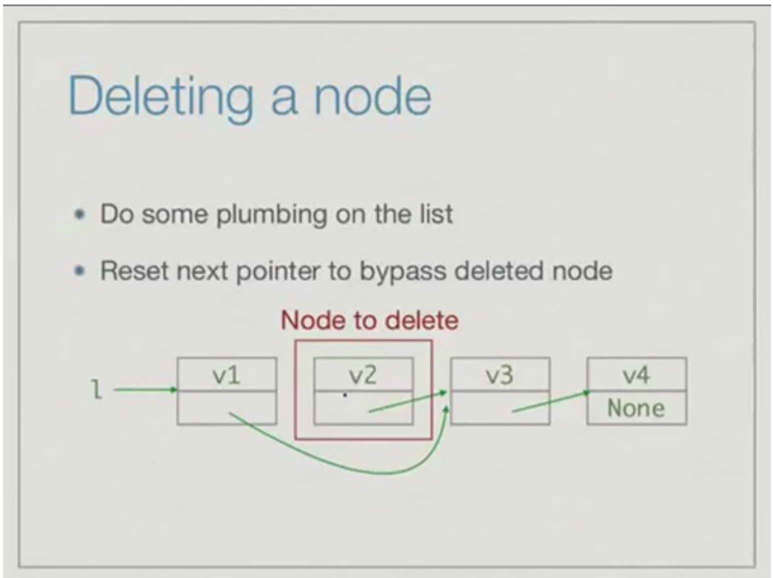
(Refer Slide Time: 10:22)



Here is the code for insert. As usual, if you have an empty list insert is easy. We just have to change none to v. So, insert and append both behave the same way with an empty list. We go from the empty list to the list v. It does not matter whether you are inserting or appending. Otherwise, we create this new node and then we do this swapping of values between the current node that self is pointing to, that is the head of the list and the new node.

We exchange the values; we set self dot value to new node dot value and simultaneously new node dot value to self dot value using this python simultaneous assignment. And similarly we take self dot next which was pointing to the next node and make it point to the new node and the new node instead should point to what we were pointing to earlier. So, new node dot next is self dot next. This is how we insert and insert as we saw is a little bit more complicated than append because of having to handle the initial way in which l points to the list or self points to the list.

(Refer Slide Time: 11:19)

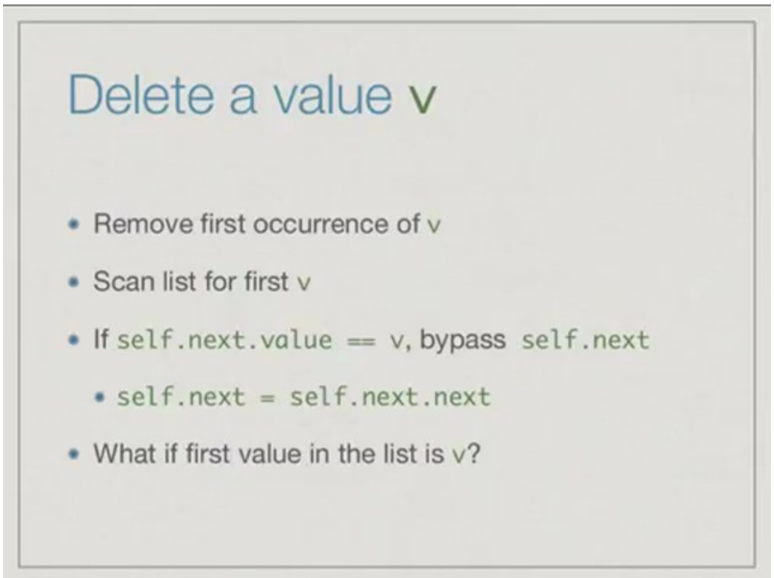


What if we want to delete a node? How do we specify to delete a node? Well we specify it by a value, but let us just suppose you want to delete say the second node in this list.

Now, how would we delete it? Well again just as we did insert we would do some re plumbing or re connection. So, we take the node that we want to delete and we just make the link that points to v 2 bypass it. So, we take the link from v 1 and make it directly point to v 3. So, in essense, all that delete requires us to do is to reassign the pointer from before the deleted node with the pointer after the deleted node. It actually does not physically remove that object from memory, but it just makes it inaccessible from the link end.

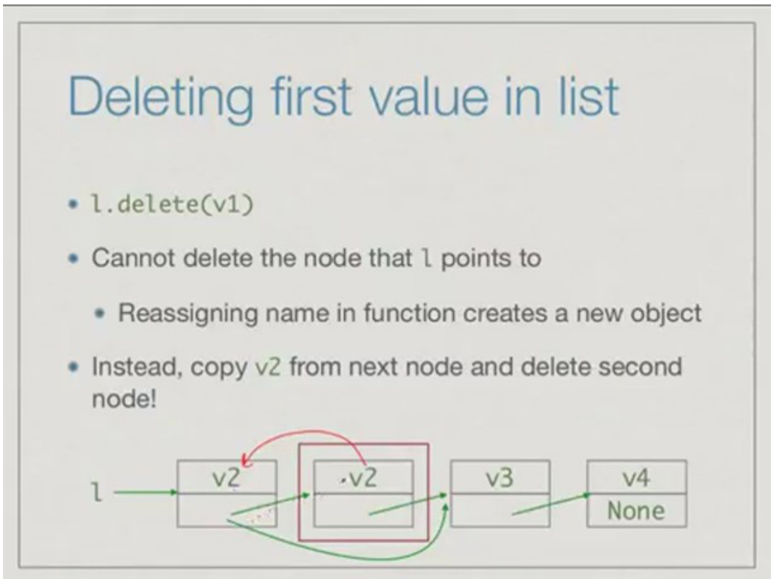
We provide a value v and we want to remove the first occurrence of v. We scan the list for the first v. Now notice that in this plumbing procedure we need to be at v 1 in order to make it point to v3. If we wanted to delete the next node then we are in good shape because we can take the next dot next and assign it to the current next. So, we should look 1 step ahead. If you are already at v 2 then we have gone past v 1 and we cannot go back to v 1, easily the way we have set up our list because it only goes forward; we cannot go back to v 1 and change it.

(Refer Slide Time: 12:38)



What we will actually do is we will scan by looking at the next value. If the self dot next dot value is v that is if the next node is to be deleted then we bypass it by saying the current node's next is not the next node that we had, but the next node's next. So, self dot next is reassigned to self dot next dot next - bypass the next node. As before like with insert the only thing we have to be careful about is if we have to delete actually the first value in the list.

(Refer Slide Time: 13:12)



If you want to delete the first value in the list exactly like we had with insert the natural thing would be to, now say that l should point to the second value in the list, but we cannot point l there because if we reassign the node that l points to then it will create a new object and it will break the connection between the parameter we passed and the thing we get back. We use the same trick. What we do is we copy the value v 2 from the next node and then... So we just copy this value from here to here and then we delete v 2. So, we wanted to delete the first node, we are not allowed to delete the first node because we cannot change what l points to. So, instead we take the value in the second node which was v 2, copy it here and then pretend we deleted v 2 by making the first node point to the third.