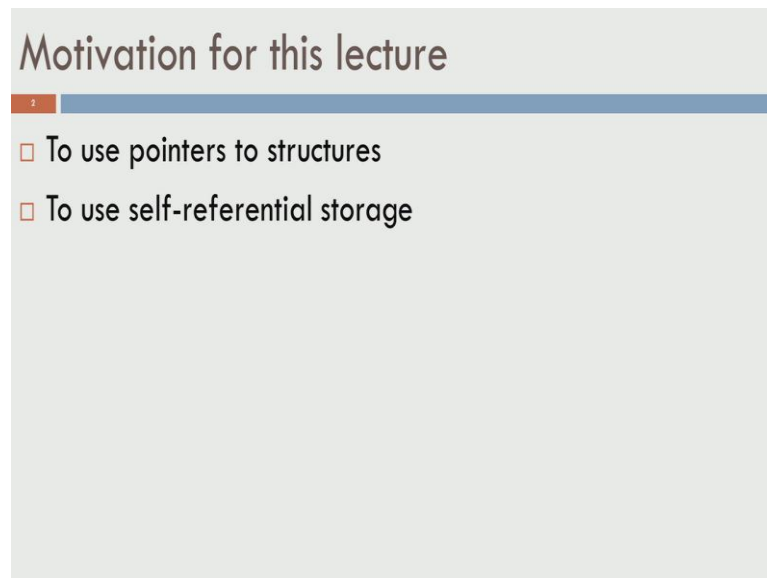


Programming, Data Structures and Algorithms
Prof. Shankar Balachandran
Department of Computer Science
Indian Institute of Technology, Madras

Module 12A
Lecture - 40
Linked lists

Welcome to this module. We will look at what link list are, and we will see how structures can be used to represent link list. So, we have seen how structures are defined. There is one important class used for structures and we will see how to do what is called self-referential storage.

(Refer Slide Time: 00:32)

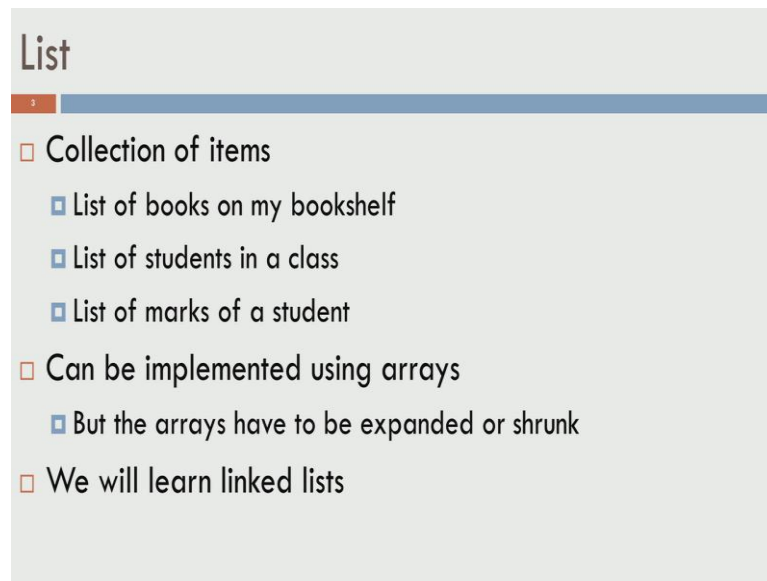


Motivation for this lecture

- To use pointers to structures
- To use self-referential storage

This is a very useful thing to do and this is also something that you are going to learn in a lot of detail when you do what are called data structures. So, I want to do this in C little bit and also, use the example here to motivate the need for c plus plus. So, the reason why we are using c plus plus in the following lectures on data structures is that, the syntax becomes much simpler and cleaner. Even though course is based on c programming, there are few lectures which are done using c plus plus, and it is good for you to know the syntax of c plus plus so, that you can appreciate what is happening in those lectures.

(Refer Slide Time: 01:15)



List

- Collection of items
 - ▣ List of books on my bookshelf
 - ▣ List of students in a class
 - ▣ List of marks of a student
- Can be implemented using arrays
 - ▣ But the arrays have to be expanded or shrunk
- We will learn linked lists

So, let us start with the list. List is nothing, but a collection of items. So, maybe I have a lot of books at home and I want to maintain a list of books on my shelf, or I teach a class and I want list of the students in my class or even for a particular student, I want to track all the list of marks in that particular semester and so, on. So, clearly you can implement this using array. One small issue in arrays is that every time there is change in the number of items that you need, you either have to shrink the array or expand the array. So, C it does not give you a nice feature, to either expand of shrink array. So, if you say array int a of 100, you have 100 integers and nothing more, right. So, this is a small problem. Maybe I have only 90 students in my class, but I allocated space for 100 or worse, I have 110 students in my class and I have space only for 100 students. It would be nice to be able to expand or shrink based on the necessity and one basic way of doing that is using what is called a link list.

(Refer Slide Time: 02:26)

Pointer Implementation (Linked List)

□ First, define a Node.

```
struct Node {  
    double data;    // data  
    Node* next;    // pointer to next  
};
```

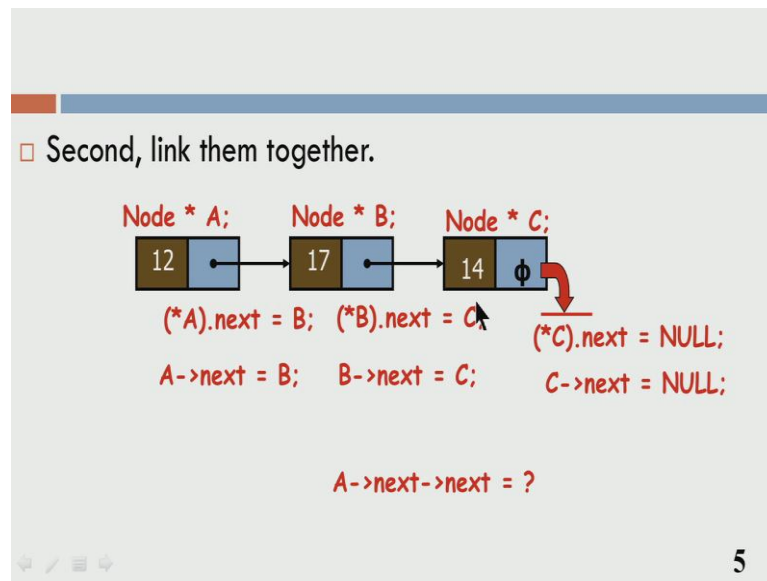
Diagram illustrating the structure of a Node:

4

So, let us see how link list can be implemented. So, usually what you have is, you have the notion of what is called the data. The data is what you want to really maintain, but you define a node, a structure called node which has the data that you want. It also has something called next. So, such a structure would look like this. You have struct node and you have let us say double data node star next. So, what we are going to do is, we are going to use this field called data to store the data that we want and we are going to have this field called next which is going to point to the next node. So, you can see that there is struct node here, and node star here. This is what makes it self-referential. So, you are having one element of the type whatever we want in this case it is double and we are going to point to the next element which is of the same type and so, on.

So, you can think of this as playing treasure hunt. So, you start with one location where there is a clue, and this gives you a pointer to somewhere else, where there is a new clue and you keep chasing the clues till you go to the end where you have a prize. So, this is how this game called treasure hunt works. You can think of link list in a very similar way. You have a data instead of clues and you also have clue which tells you where to go next and then, you go there and you have another clue and so, on. You have a chain of these things, at some point, this list ends. So, I already mentioned self-referentiality comes from the fact that you are referring to node of the same type and here if you notice we have struct node here this node is immediately visible. Here the data type called node, it is immediately visible here. That is not a problem. C allows you to do that and c has that feature specifically because it wants to support the self-referential structures.

(Refer Slide Time: 04:45)

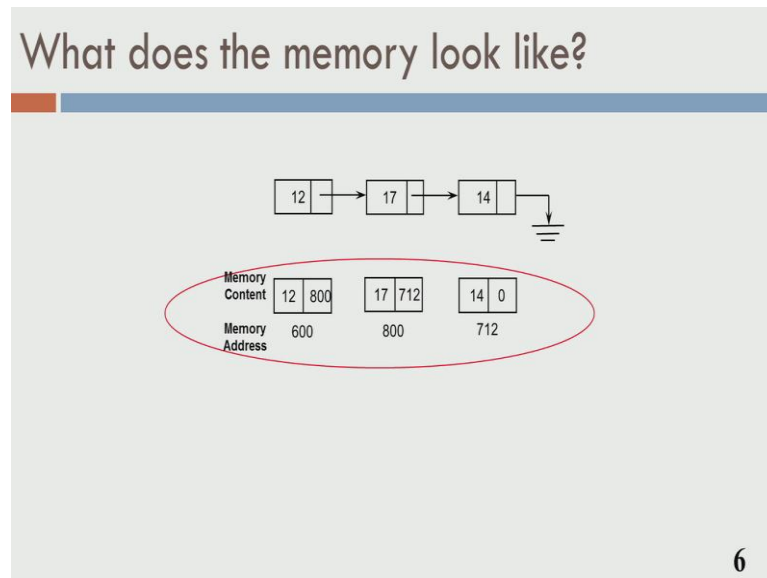


So, now, let us say I have three nodes, namely a, b and c and let us say that node a is supposed to have a value 12, node b is supposed to have value 17 and node c is supposed to have value 14, and we have data, and what are called the pointers. What we are going to do is we are going to chain them together. From a we link its next to b, from b we link its next to c, and c is the last node in the list, and we do not want it to point anything. So, we need to have some null at the end. So, now, we can see that this is the link list. So, it is a list, but to reach one element, you have to start from the first element, follow the links that point to the next element and so, on and keep going until the end. So, if you want to reach 17, you cannot access 17 directly. So, in an array I would have accessed a of 1 that would have taken me to the second element in the array, a of 2 would have taken me to the third element in the array and so, on. However, in link list you access the first element; you chase the pointer here, and get the next element, then chase its pointer to the next element and get to the element after that and so, on.

So, we assume that a, b, c are all pointers here. Therefore, if you do star a, it gives this structure and you can do a dot next equals b. So, remember b is pointer. So, a next field is also a pointer so, these are compatible. Finally, for null I already mentioned that there is keyword called null in c. So, star c dot next is null which means c has nothing following at. So, if you are using some operation, if you are going through the list one after the other, once you see null, you are at the end of list, you cannot go any further. There is another way to do the same thing. Instead of star a dot next, you can also do a arrow next equals b and so, on. We already saw this as pointers to structures and we saw

the arrow operator earlier. So, now, if I ask you what is a's next, next? So, what is next to next a? Then, you start with a, you take next, you go here, one more next that goes here. So, a next to next is actually the pointed to this node called c here. So, whatever c is pointing to that is the point you will get.

(Refer Slide Time: 07:26)



Let us see how this... So, all these looks abstract, right. I have a list and so, on. Let see how this looks in memory. So, we want this abstract list, we want node 12 linking to node 17 and node containing 17 should link to node containing 14, and node containing 14 should not point to anything more, this is what we want and these nodes could be in different locations in memory. For instance, may be node containing 12 is at location 600, nodes containing 17 is at location 800 and node containing 14 is at location 700 and so, on. If you establish the link list finally, this is how it must look like. So, you will have 12 at location 600, and the next field contains the address of the node containing 17. What is that address? It is 800, so, that is what we have here. And if you go and look at node containing 17 that is at location 800, what is its next value. We want it to point to the node containing 14. So, that is at location 7, 12 and that is what we have here in the address and so, on. So, in the memory this would look like this, and as an abstract way, it is what you see here. So, 12 is next to 800 which means if you go to location 800, you get the value next pointer. If you go to location 7, 12, you get the value and next pointer, and null is nothing other than 0. So, null is defined to be 0 in c, and this is fairly simple and straight forward way of using a structure to make a link list.

(Refer Slide Time: 09:15)

Operations on the list

- How to change the list?
 - ▣ Inserting a new node.
 - ▣ Deleting a node.
 - ▣ Finding a node / displaying nodes.

7

So, there are several operations that happen on link list. You may insert a node, you may delete a node, you can find a node, or display the nodes and so, on. This is a very basic set of operations that we do. Let us say I have 100 students to start with and another 50 students signed up for the course. I go on and insert 50 more nodes and have them in order. Maybe some student drops out of my class, I want to delete that node or I want to find the student who has the particular role number. I may have to search through the list and find it and so, on. So, the notion of link list allows me to do all these things one after the other.

(Refer Slide Time: 09:56)

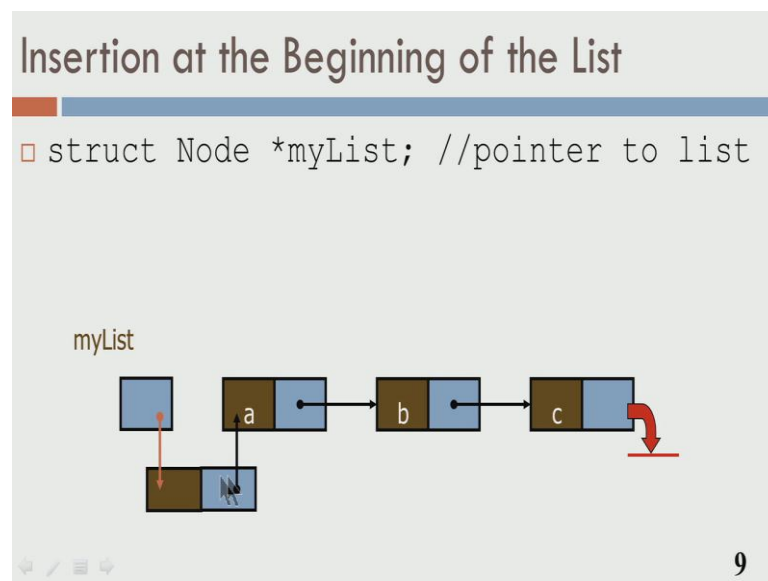
Inserting a node

`B->next = A->next;`
`A->next = B;`

8

Let see how insertion of node might be implemented. Let us say I already have a list where I have nodes containing 12 and nodes containing 14. I have 2 nodes and maybe there is the list that is following that, I want to insert this node containing 17 between 12 and 14. Let us say I want to do that. You have to do a series of pointer manipulations. What you really want to do is, you have a new node which contains the value 17 and since, you wanted between 12 and 14, you want this nodes next pointer to be 14, right. So, how do you do that? If this node is b, b is next should be a's next that is what we have here. B's next is a's next. Now, this does not form a link list. So, it is not a chain of nodes that we have now. Instead we have 2 nodes pointing to 14. There is no way in which you can reach this node containing 17 from the node containing 12. We need to change that and the way to do that is, if you make a's next point to b, then a's next feel pointed to b. Now, if I start from a, so, a's value 12 and a's next will point me to b. B is value 17 and b is next will point to be 14 and so, on. So, now, you have a link list, and you have one element linking to another linking to another and so, on.

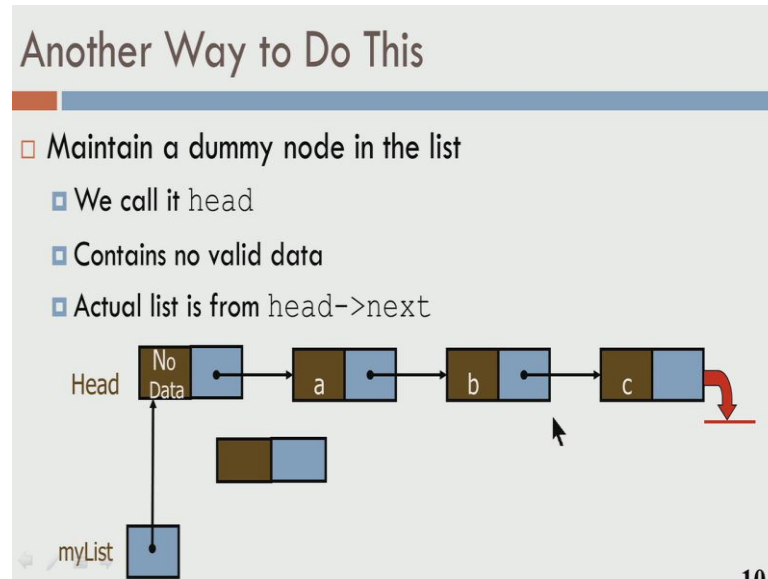
(Refer Slide Time: 11:29)



If I want to insert at the beginning of the list, let say I have struct nodes to my list that is pointed to the whole link list, and I want insert node at the beginning of the list, then the sequence of operations would look like this. So, I created a new node. I will take that and point to the first node that is already in the list, and I would change my list to point to the new node. So, in this case I am not showing what the contents of the new node are, but that is not the point. I want to insert in node and if it add some valid value, let say d and

if I go through the list if I traverse the list from the beginning to the end, I will see d, a, b and c.

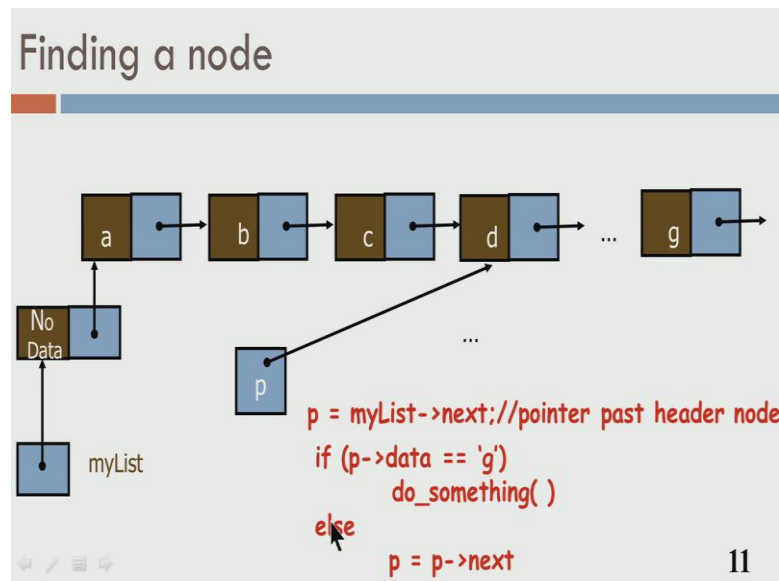
(Refer Slide Time: 12:19)



There is also another way in which you can insert elements to the list, and this is the very typical way of doing it. You will see this in the adt called list later. So, instead of having a list of items and just the list, it is useful to have what is called a head node which has no valid data. So, it is going to be a dummy node, and this dummy node is going to point to the list. So, we will call this the head node. It will not have any valid data and the actual list is only going to start from head dot next. What I want to do is, I want to insert a new node here in the beginning of the list, right. So, clearly this node cannot be pointed from here. My list should not point to this because, here we are assuming that there is always a header node which is dummy and from the dummy node, you should be able to get a list of all the elements. So, let us say I have this new node called temp. What I really want is the dummy node should start pointing to temp and temp should start pointing to a.

So, now, if I wants to traverse the list, I start with the dummy node. I will not look at the value in it because I know that is dummy. So, I will go to its next. There is the valid value there followed by here, here and here that takes me to the end of the list. So, this is the typical implementation of a link list, where there is always a dummy node and having the dummy node in place helps you a lot when you do inserts and deletes and so, on. You will see this in more detail in a later lecture anyway.

(Refer Slide Time: 14:03)

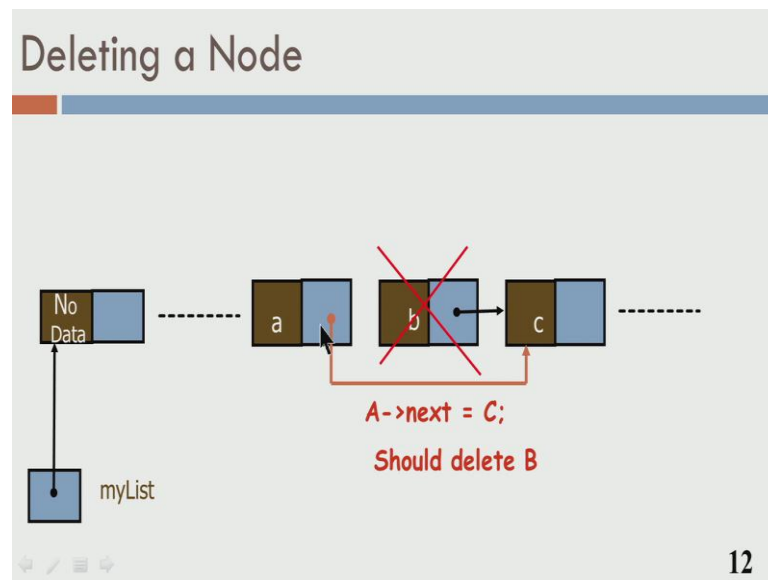


So, let us say look at another method for finding a node. Let us say I have a link list and I am going to look for student by a certain name. Then, I have to start from the beginning of the list, and keep going, keep going along till I do not find the student in the list or the moment I find the student I want, print details of the student. So, I am going to show you a method for finding a node in a list. So, remember my list is pointing to a dummy node called the header node, and the header node is actually pointing to the list. So, clearly I should not start searching from the dummy node itself. I should start from the first valid node which is this. So, we will have a pointer `p` which points to node which is past the header node and we are going to look at one element at a time.

If I already reached the node that I wanted or that I was searching for, I can stop and do whatever I want to do with it, otherwise I have to keep going along, right. Let us see how to write the program for this. So, you start with `p` equals `my list dot next` or `my list arrow next`. So, what this will do is, it will start with `p`, pass the first pointer like this. So, you start with `p`, it starts `my list dot`, `my list arrow next` is this field and that is pointing here. `P` will also start pointing there. So, it starts from there and the very first data itself could be `g` and if it is `g`, I want to do something about it. If not, I have to go and chase the pointer. I have to move from the first node to the second node and so, on. So, to do that we have `p` equals `p dot p arrow next`. So, we start from here and if it does not satisfy the condition that I am searching for, then I move here and so, on. So, you have `else p` equals `p dot next`.

So, this takes you one step away and if you put this whole thing, if then else condition if inside a while loop, it will move one element at a time till you either reach the null which is the end of list or when you reach the condition. So, you need a while loop around this structure, but this is the basic traversal step. If and else is the basic traversal step.

(Refer Slide Time: 16:26)



Now, let us look at how to delete a node. So, I have this list somewhere a, b, c is in the middle of my link list, and header node is here. There are may be other nodes here and other nodes here, and I want to delete the node containing b let us say. So, one way to do that is a follows. We know that we have to find out where these pointed to from. So, b's predecessor is a and what I want to do is change the contents of a to point to c as the next element, right. So, b's predecessor is a, right. So, a successor should now be c and not b. So, what we want us is a's next field which is now pointing to b should change to c. This is what we want, right and we should also delete b in the process because right now this node is still in memory and if you do not delete it, is going to be a problem, right. So, we need to be able to delete this also and this will be reclaimed by the operating system. So, let us see how to do that.

So, you start from the beginning. You keep looking at elements till you reach the node, right. In this case you are reaching b, but you cannot go to b and then delete b. So, you keep looking for elements whose next pointer satisfies the criteria. So, if I start from the beginning, I will keep going along till I hit a, because a satisfies the condition that b is the next node, and this is what I want to delete. So, I stop at a, and make a's next pointer

as a's next to next, right. So, a's next is b, its next is c and I take that and I put the value in a.

(Refer Slide Time: 18:33)

More Practice

- How do I delete first node in the list?
 - Assume non-empty

```
tmp = myList->next;
myList->next = myList->next->next;
free(tmp);
tmp = NULL;
```

13

So, let us see how to delete very first node in the list, and I am going to show you program segment now. Let us assume for now that the list is non-empty and I have four nodes in the list shown in the figure. Maybe there are more elements and I want to delete the very first node in the list. If I want to do that, I start with variable called temp which points to my list dot next. So, it is this field here. My list next is this field here and I want my list next to point to my list next to next, right. So, my list next is now pointing to a, I want that point to b, right. So, that means, you actually have this new link that is created, you want that and temp is still pointing to a. A is not deleted from memory, A is also pointing to b, but a is not in your list any more. So, at this point if I go and do a search on a, it would be a failure because it starts with dummy node and I will go past the dummy node, start printing from b, c and d and so, on. A is not in the list, right. However, a is still in memory. You want to remove it.

You free temp which means a is gone, right. So, I talked about this in the dynamic memory allocation class, so, a is gone. However, temp is still pointing to something which is invalid anymore. So, it is not a valid memory location. So, you have to make temp equals null. So, any further access to temp will be identified by during run time. So, if you access temp next now after making it null, the run time, when the program is running, this will be an error and you will have a condition where you are chasing a

pointer which is not valid anymore, this will be indicated during run time. So, this is the simple piece of code to delete the very first node in the list.

(Refer Slide Time: 20:41)

Printing the elements in a Linked List

- void PrintList(Node* ptrToHead)
 - Print the data of all the elements (except header node)
 - Print the number of the nodes in the list

```
void DisplayList(Node* ptrToHead)
{ int num      = 0;
  Node* currNode = ptrToHead->next;
  while (currNode != NULL){
    printf("%c\n", currNode->data);
    currNode = currNode->next;
    num++;
  }
  printf("Number of nodes in the list: %d\n",num);
}
```

14

How do we print elements in the link list? Printing the elements is just traversing the list from past the head node till you reach null. So, I assume that print list is a function which actually takes a pointer to the head node itself, and we want to print out all the nodes except the head node and I also want to print the number of elements in the list. So, display list is this function or print list. This should have been print list. So, we start with zero elements, to begin with, we do know how many elements are there in the list. We assume that there are zero elements and we take current node, this curr node to point to node pass the head node. So, we start with head nodes next. If the list is empty, this ptr to head, it is next will be null. So, current node will also be null, but even if there is just one element, this ptr to head next pointer will be a valid value and current node will start with a valid value. So, while current node is not equal to null, you print the contents of the current node, move the current node to the next element and increment the number of elements. You keep doing this in a loop till you hit current node being null.

Finally, this one prints the number of elements in the list. So, you can clearly see how this notion of a loop takes you from one node to the next node to the next node and so, on. And this current node is a pointer which keeps moving along and current nodes data will be the data one after the other. So, there are several operations that happen on a link list.

(Refer Slide Time: 22:36)

Typical Operations On Lists

□ Operations of List

- CreateList: create a list with just the dummy header node
- IsEmpty: determine whether or not the list is empty
- Append: insert a new node at a particular position
- FindNode: find a node with a given value
- DeleteNode: delete a node with a given value
- Print: print all the nodes in the list

15

So, you have to create list, you may want to check if a list is empty or not, you want to append to a list. You want to find if node is there or not. You want to delete node, may be print the list insert at a particular position there may be several things like that you may wanted to do with the list. So, you will see a subset of these things in the class on list later. So, there are several things and you may want to implement these things and when you do this, you have to be really, really careful. You have to watch out for the lot of details about how this is implemented and so, on right. So, there are lots of things that you have to remember and it is not good to always leave it to a programmer to go and do this every time. So, instead it may be nice to have the whole thing packaged. So, just like what we had for the basic data types like integer and floating point and so, on. Sometimes it is good to actually package the data that go with the operations that are allowed. So, for instance the moment I say integer, I know that only certain operations are allowed and certain operations are not allowed and so, on.

(Refer Slide Time: 23:47)

Issues

- Too many details to remember
- Will be nice to have it packaged
- Data should go with operations
 - ▣ Like it is for int, float etc.
- Quick intro to C++ for dealing with data structures to follow

So, what we are going to do is, I am going to motivate this in the next lecture and I will also introduce c plus plus as a vehicle with which we will do this. And once you have this next lecture on c plus plus as a very quick introduction if you notice, the course will now use c plus plus as a medium of instruction. So, this is for all the lectures on data structures you will see c plus plus being used. So, one reason why we decided to do that with c plus plus is that the language lets you express the program much more clearly and concisely, and c does not let you do that. So, you will see that the lectures on data structures is going to use c plus plus, and it becomes much more cleaner and easily understandable when you use c plus plus. So, this brings me to the end of this module.

Thank you.