Programming and Data Structures
Prof. N. S. Narayanaswamy
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 07
Doubly Linked List

Hi, so in today's lecture, we are going to talk about Doubly Linked List. So, recall that in the lectures last week, we studied specific implementations of different abstract data type. Specifically, we looked at data type linked list and what we did was we described the abstract data type and then, identified the appropriate methods; the appropriate data and we went ahead and implemented it. After implementing the different methods associated with abstract data type, we used these methods in a specific programming exercise.

Let me recall the programming exercise for you, the programming exercise was to take two large integers as input and two large positive integers as input and then, add those two last integers. While this exercise looks to be very simplistic, it is a very interesting exercise, it is an exercise in learning how to manage and maintain linked list. And you can add arbitrarily large integers, we make some simplifying assumptions, we assume that the integers are always positive and that there are no decimal points and so on and so forth.

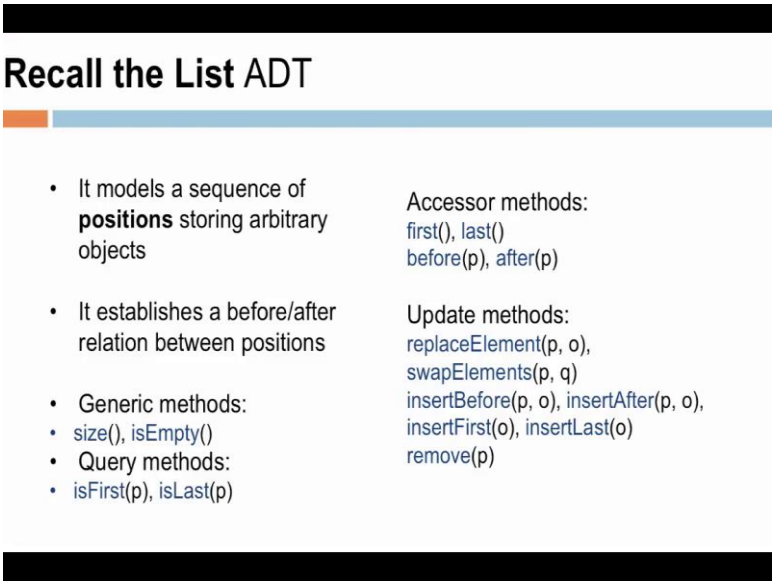
And most importantly while implementing the linked list abstract data type, we also paid significant attention to programming practice. We saw, how to use a C programming language to partition a large program with different kinds of data and different kinds of functions into appropriate pieces of code. I mean each piece of code sat in a separate file, one was referred to as a source, the other was an object file and the third one was a header file. And all these are important programming practices.

In today's lecture, we are going to extend the linked list data type. And when we implement the doubly linked list data type we are going to reuse code and reusing code already develop programs is a standard approach to designing systems. What you do is, you take a piece of code, that is tested and that is deployed and you want to identify some minor functionality that you want to add or change and you modify the code to incorporate this minor addition or the minor change.

Today, what we will do is, we will take this linked list implementation that we have and implement the doubly linked list data type and during this implementation, you will see that one has to be extremely careful, when one reuses code. While, this is not the central theme in the lecture, it is a very important associated theme with programming. So, now let us go to the slides for the doubly linked list data type and in these slides, you will notice that there is a strong motivation for the doubly linked list data type.

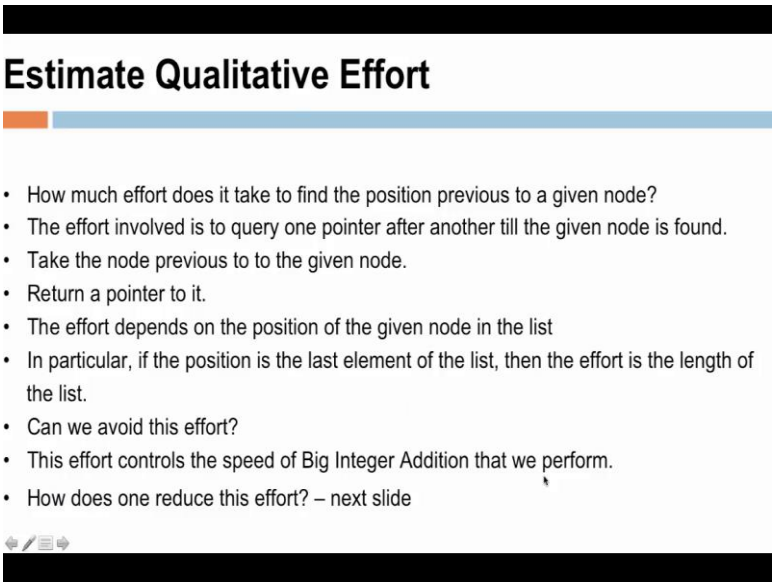
In other words, we are interested in the issues of efficiency associated with this the big integer arithmetic and to deal with these efficiency issues, we essentially introduce this doubly linked list data type and that is a focus on today's lecture. So, let us go to the slides, so doubly linked list.

(Refer Slide Time: 04:04)



So, as I said before we are going to extend the list abstract data type and this is a summary of list abstract data type. Recall that the data is organize in a list like fashion and we have some generic methods to query the list itself, whether it is empty or how many elements are there in the list. We can also query a certain position to find out, whether it is a first and the last position and then, we have access methods which will return pointers to specific positions in the list. And finally, we have update methods which change the data values at different positions in the list.

(Refer Slide Time: 04:43)



Let us look at the qualitative effort in maintaining the link list. I use the word qualitative, because I am not going to specifically talk about the efforts in terms of quantities like time. So, normally efficiency is quantified in terms of time and rather, time is one of the ways of quantify the efficiency. And on the other hand, this discussion is about the qualitative effort in maintaining the link list.

So, let us just look at a simple exercise that we repeatedly perform, while adding a large number. Recall that the addition was from the right most digit; otherwise, the least significant digit and we move towards a left repeatedly adding the individual digits, keep in track for carry that gets generated. Also let us recall that in the list implementation, the number was visualize from the left to right fashion, where as the addition is from the right to left fashion.

Consequently a repeated task that we performed was to find a position, which is just previous to a given node. So, therefore, the effort involved is to find this position that occurs just previous to a given node. The way we do it is, we skip through from one pointer to another pointer, one node to another node by traversing the pointers, traversing meaning, we move from one node to another node by crossing the another pointer, this just the way of visualization.

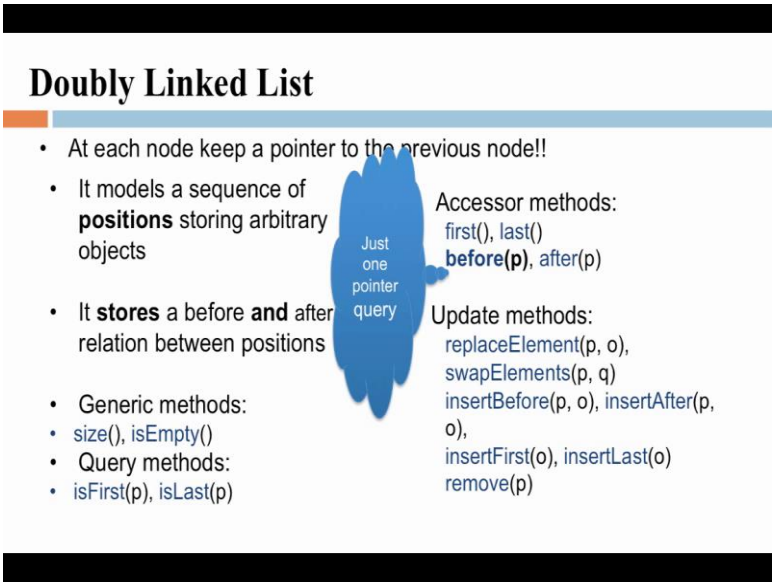
And once, we find the correct node, that is a node, who successor is the given position, we have found the immediate criticizer and this is a repeated effort. Now, observe that

the number of pointers that we traverse is representative of the amount of time that we spend in finding the criticizer node. And therefore, if you are repeatedly querying for the criticizer node, then using the link list data type to represent your data is a bad idea, because, the algorithm will repeated.

The programs at we write will repeatedly, we traversing the list from the head onwards. Because, in the link list data type, we have access only to the head of the list, whenever you need to find a specific position, you have to find the position by traversing that list. Therefore, to avoid the effort of traversing the list, the doubly linked list data type is used and it is obtain by a very minor modification to the link list data type.

It is also clear that the number of traversals that we perform controls the speed of the big integer addition. That is if more the point to traversal longer the running time of the addition algorithm, therefore, our goal is to attend to reduce this effort.

(Refer Slide Time: 08:02)



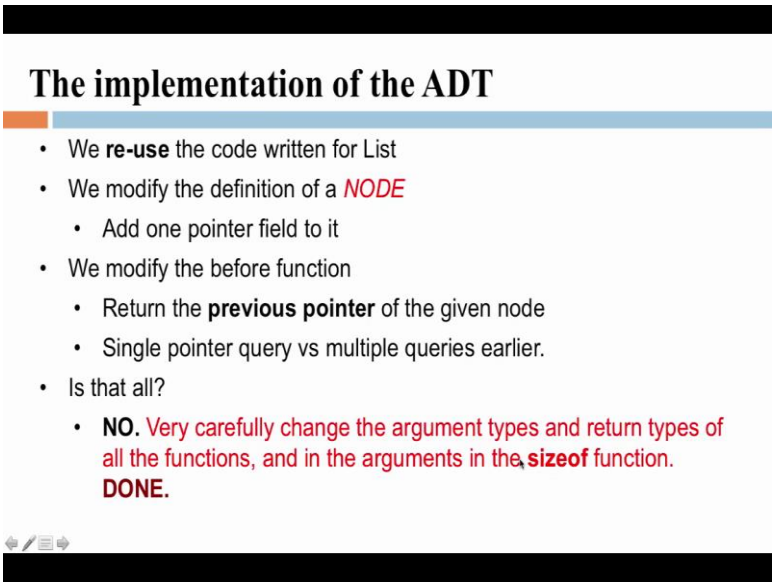
To reduce this effort, we introduce doubly linked list data type before what we do is a very simple modification at each node we also keep a pointer to the criticizer. Therefore, we modify the node data type that we had with the link list implementation. The way we modify it is by just adding one more pointer to the node data type and this pointer is design, so that, it will contain the address of the previous node in the list or the predecessor node in the list.

Therefore, the doubly linked list is obtained by extending the singly linked list in this particular fashion. We essentially key pointers to the before and after nodes for each position, except for the head node of the list for the first node of the list, the immediate criticizer is well defined and everything else is very similar to a list, all the methods are essentially the similar.

But, since we are maintaining additional data, the implementations of some of these functions will definitely change. In particular, let us take the case of before of p, recall the before of p was taking the position and a list as an argument and it would return a pointer to the element, which is before p in the given list. Earlier this function would essentially traverse the whole list and find the criticizer of position p in the list implementation.

But, in the doubly linked list implementation, it is just one pointer query, that is, you only have to take the previous value, that is, you take the previous field of this particular object. And essentially, we get the previous position to p, other methods are conceptually the same, the return values are identical, they will be some minor implementation, minor modifications in the implementation.

(Refer Slide Time: 10:13)



So, how do we know implemented, therefore, here comes a second important concepts, we can of course, start implementing the ADT from scratch. But, in this lecture, we are going to do something interesting; we are going to reuse the code that is return for list.

When, we reuse code, we have to be very careful and modify one the definition of node, this is very clear, there is no conceptually the modification we have already discussed it, we will keep a pointer to the previous and essentially we have one more pointer.

But, when we modify the definition of the node, we should also remember to also change the name of the appropriate types, because we are working with the C programming language. So, especially, if you are going to use the linked list data type in a program and also the doubly linked list data type program, then naming conflicts must be very carefully avoided, we will see that in a minute

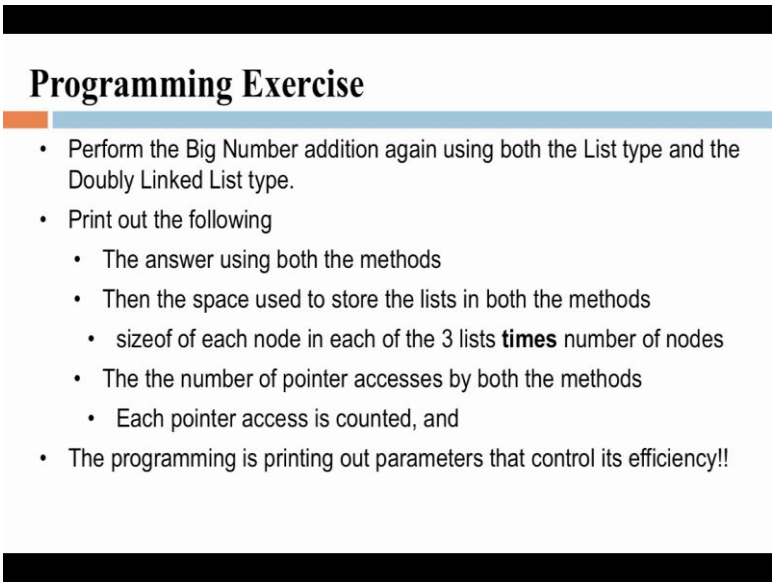
Second modification that we need to perform is that for we need to modify the before function and other functions to take care of the fact that the node data type has one additional piece of information, which ensures a we can avoid some search. Therefore, two major modifications, one we change the structure of the node data type; we keep one additional field in sided. Second, we use this additional field to simplify the implementation of different functions, these are two conceptual modifications.

Here is an implementation modification, which means that because we are going to reuse the code, which was return for list, we should be very careful and ensure that the argument types and the return types of different functions are changed. The arguments in functions like size of which are library functions are also carefully changed. Then, they also need to ensure that the names of the functions are also appropriately changed, that is C does not provide us the feature of using the same name for a function, which performs same task on different data types, we should have different function names.

So, I repeat this, we are limited by the features of C, the programming language. So, for example, if you have a function which is performed on the linked list data type and the same function is performed the doubly linked list data type, we must ensure that the two functions have different names, especially, if you are going to use the two data types in the same program. So, this is another major modification that we must take care of when we reuse code.

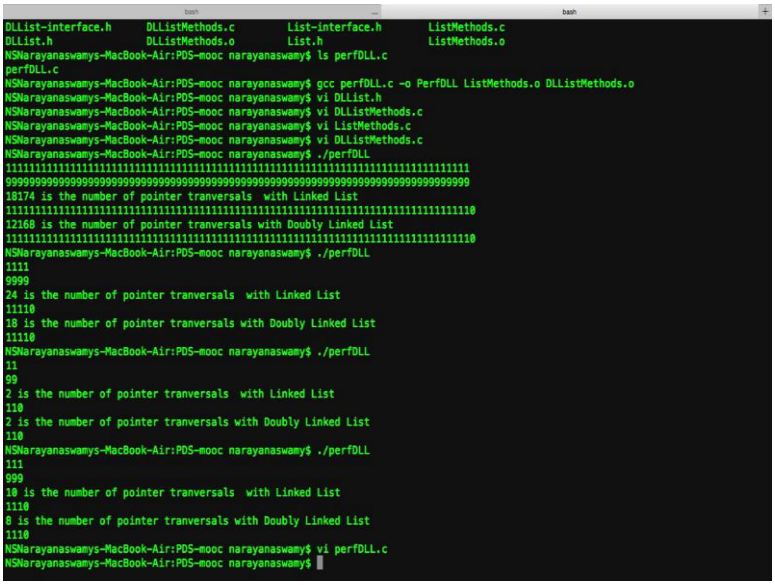
In general, whenever you reuse code, one must reuse code in a very careful manner, otherwise, one may not be able to predict the kind of errors that will happen.

(Refer Slide Time: 13:09)



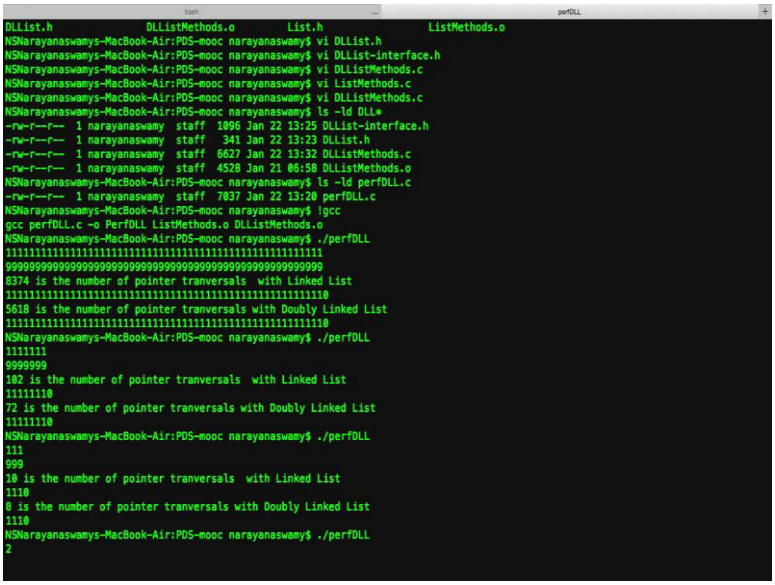
Then, we will look at a programming exercise, but for the moment, let us look at our implementation of the doubly linked list data type.

(Refer Slide Time: 13:28)



So, let us this look at the doubly linked list data type and let us see the kind of files that we are going to deal with.

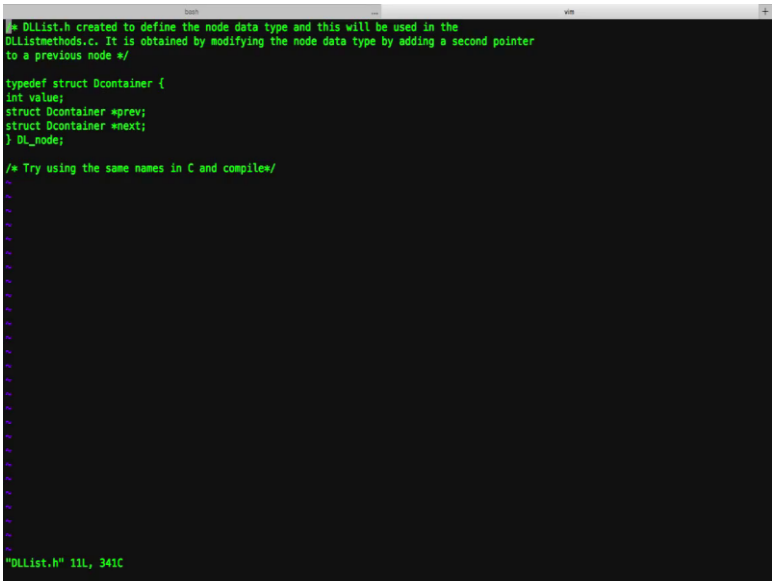
(Refer Slide Time: 13:34)



So, you will be familiar from previous lecture about the three files list dot h, list interface dot h and list methods dot h, these are the three source files and the object file was obtain by compiling list methods dot c and it is called is methods dot o. Similarly, we create three source files for the doubly linked list; these are obtain by extending the list implementation itself. So, we have three additional files now, which are relevant to today's lecture, one is called DL list dot h, then DL list methods dot c and DL list interface dot h.

So, the DL list dot h contains the data type definition, that is the modify node type, the DL methods is dot c contain modifications of the list methods, appropriate modifications which have carefully done. The DL list method dot o is obtain by compiling the DL list methods dot c file at a programmer, who is programming by using the appropriate, these data types will introduced will include the DL list interface dot h on the DL list and the list interface dot h.

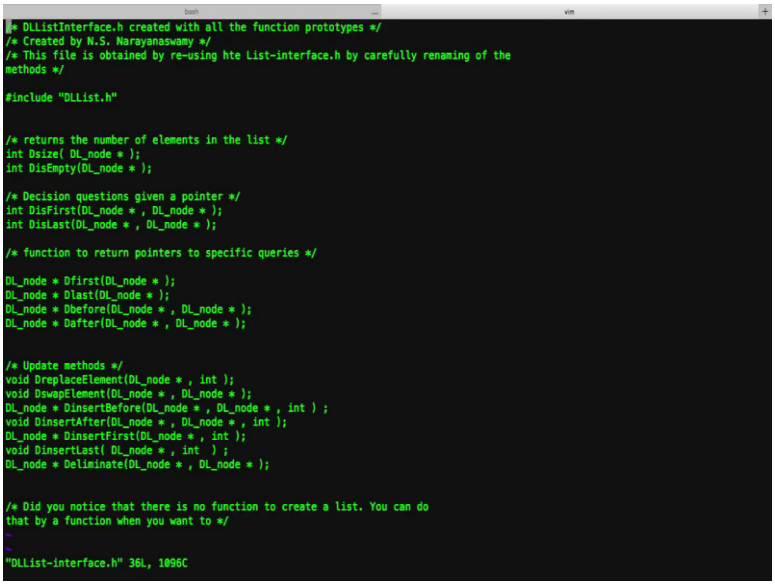
(Refer Slide Time: 15:15)



So, let us look at these files one at a time, as you can see the modification is that the type name has change, it was struct container, now it is struct d container, d stands for doubly linked list, the first letter. Then, the additional pointer previous field has been added and the type name has also been changed to DL underscore node from node. Now, it is important to note that all these type names need to be carefully changed, especially when we are planning to use both the data types in the single program.

A small exercise that you should do is, try using the same names in c and compile and you will see the kind of errors that proper. So, then you should ask yourself, what is a right programming language for ensuring that you can use the same name to mean two different things, just a small pointer to get you thinking ((Refer Time: 16:19)). Now, let us go to the DL list interface dot.

(Refer Slide Time: 16:27)

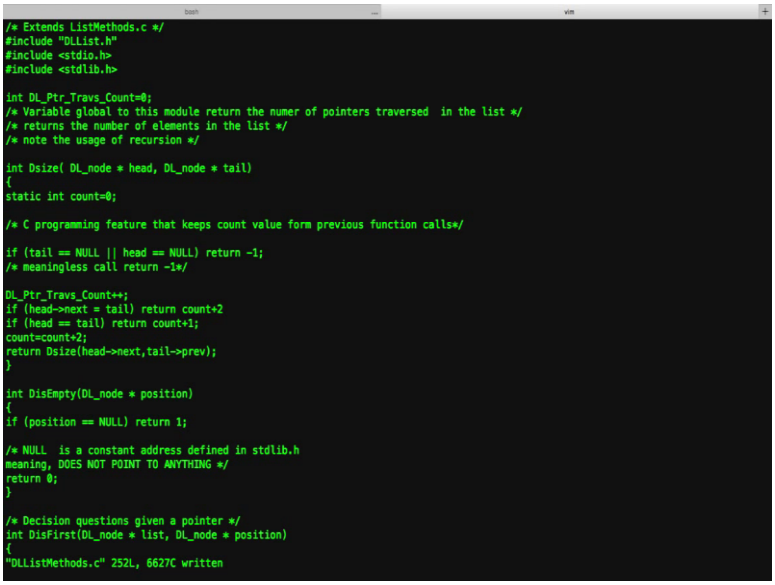


So, again now you have included the DL list dot h, this is the interface file, this contains methods that program was will use, this also tell us what kinds of methods we have implemented. So, let us just see, if one compares the files associated with the linked list and the doubly linked list, you will find that uniformly we have change the argument types from node to DL node.

Secondly, observe that we have changed all the method names, size has been change to D size, is empty has been change to D, is empty is first has been change to D, is first, first has been change D first observe that have carefully added the additional character capital D. To ensure that, there is no confusion for the compiler, because of this mistake of using the same function name in two different abstract data type implementations. That is, I am avoiding the situation, where I use the same function name in two different abstract data type implementations; especially in c, this is not permitted.

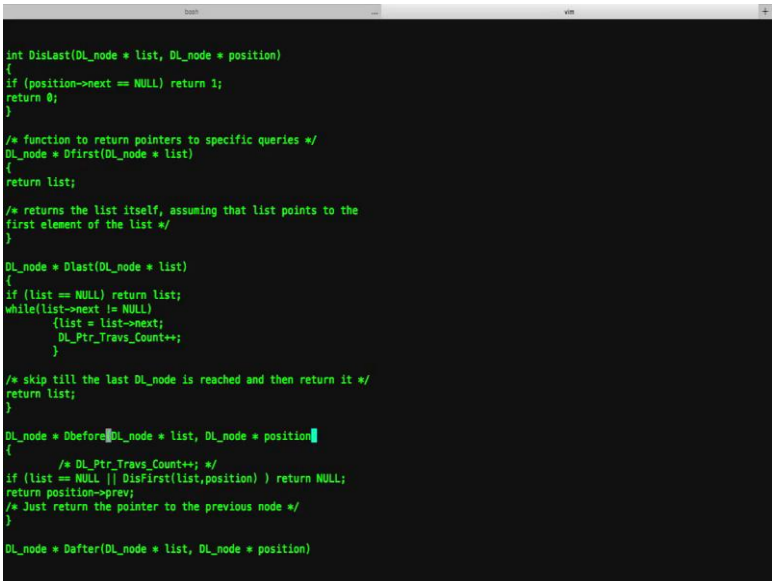
So, this is basically the list of all the methods that are implemented in the list methods dot c and this is what a programmer who wants use a doubly linked list data type will include ((Refer Time: 18:04)). So, now, let us go to the list methods dot c which contains the implementation.

(Refer Slide Time: 18:11)



So, this is inherited from... So, this extends list dot c. So, conceptually really nothing new happens here in terms of the different functions. So, all the functions for example, t size now return the length of a list D is empty. So, the concept associated with each function is exactly the same, the implementation details change and implementation details change in the following way, let us just look at the way in which implementation detail changes.

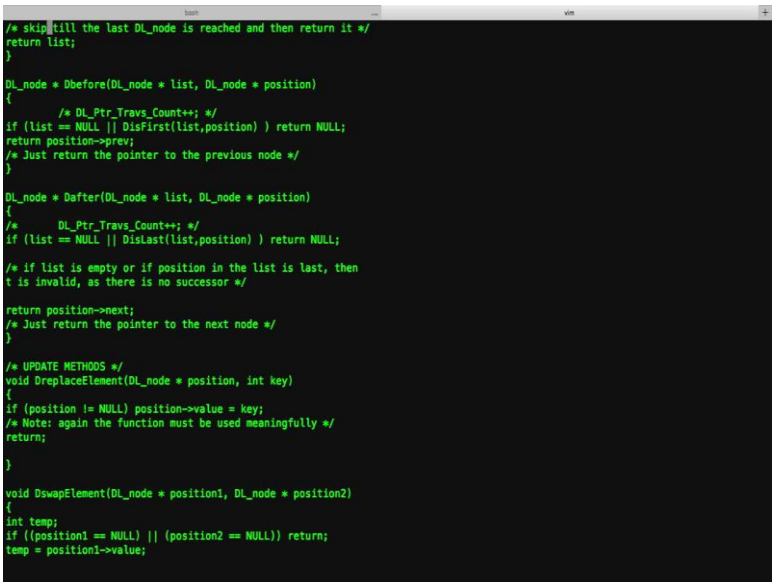
(Refer Slide Time: 19:06)



Wherever so for example, let us look at the before function. So, recall that the before function takes it as argument list and a pointer and returns a pointer to the previous note of the given pointer in that particular list, in the given list. And recall that when we implemented this in the list we add to traverse the whole list cross multiple pointers, before we get to the desired node, which us the predecessor of the given position.

In this case it is extremely simple, you just return the value of the previous field associated with the position node. So, therefore, you take at particular node the current node look at the previous field and it points to the previous node in the list and wants just returns that value, observe that would be null also.

(Refer Slide Time: 20:20)



So, this the one major change, the other major change that we have is that the way we are going to in today's lecture ((Refer Time: 20:21)) we are also not just going to implement this data type, we are also going to compare the big int addition program by considering two implementations, in one implementation the numbers are stored in a singly linked list and in the second implementation, the numbers are stored in a doubly linked list.

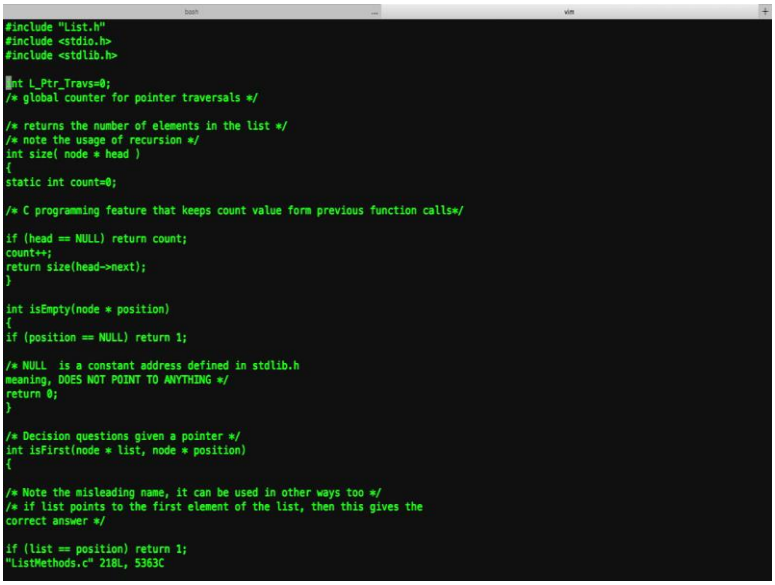
And the way we compare the two implementations is by keeping track of the number of times, pointers have been traversed by trying to perform the addition operation of the two integers. Now, we have added an additional variable which keeps track of any pointer access by calling this the we have added this additional variable called DL pointer traps count it is a variable that is called pointer traversals count. And it is pointer traverse

count in the doubly linked list implementation, it is a variable which is global to this particular module which means it every in any function it can be changed and we have initialized to 0.

So, let us see where all we changed, every time a pointer is been traverse we increment this variable to recognize a fact that a pointer has been traverse. So, where are all the places where the pointers get traverse, recall that when you want to compute the size of the list, then you have to essentially... So, when you keep track of the size of the list, you essentially have to traverse a pointer to go from one node to another node and therefore, we are incrementing the value of the count.

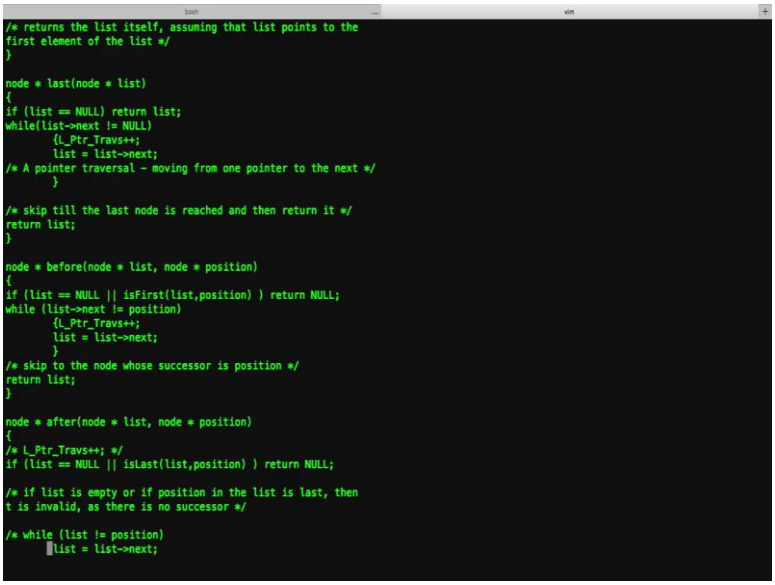
Another place is when we have to find the last element in the doubly linked list, we traverse the whole list, we skip pointer after pointer. And every time the pointer is skipped we actually we increment the counter, when these are the only two places where this counter is changed. Let us just look at the list methods dot c ((Refer Time: 22:49)) here also we do the same thing.

(Refer Slide Time: 22:53)



So, we have introduce a new counter call L underscore pointer underscore traps, this counter keeps track of the number of pointer traversals in the linked list method in the linked list implementation. Again this is a global variable, global to this module which means it can be modified in any the function inside this particular file.

(Refer Slide Time: 23:21)



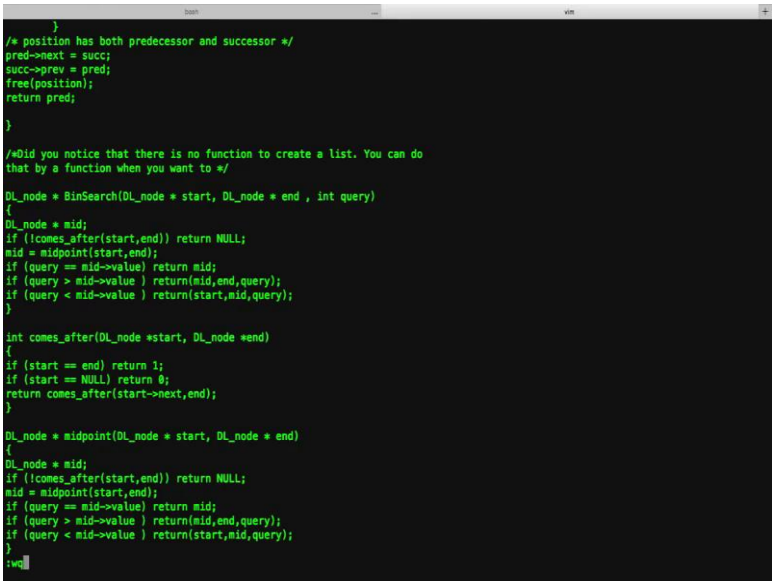
So, again when you want to find the last element of the list, we skip the pointers one after the another, one after the other and then the pointer traps counters incremented. And another places here it is very important, when you want to find the predecessor of a given node, again you have to skip the pointers starting from the head of the list, right up to the point when we find the appropriate node that we want. So, again here the pointer traversal count is incremented.

And therefore, these are the two places at which the pointer traversals count is incremented in the list implementation. Therefore, we have modify the list implementation in a small way by adding one counter, which is global to that particular list implementation. And this counter basically the value is incremented, whenever the pointer traversal instruction is executed, just after it or just before it.

So, these are the only changes that we have made and took and observe that we have taken the list implementation made a copy of it and then modified it by just updating the appropriate arguments very carefully for all the methods. But, you would also observe that we change the function names carefully. So, this completes the implementation of the doubly linked list abstract data type.

I am not going to compile the ((Refer Time: 24:59)) doubly linked list abstract data type implementation, because subsequently I have added additional method which are relevant for the next lecture.

(Refer Slide Time: 25:06)



For example, I have introduce some functions called midpoint, comes after and binary search and so on these are not compiled yet. So, therefore, I am not going to compile it and show it you ((Refer Time: 25:18)). However, you can see that I have already compile the doubly linked list implementation quiet recently and it is available for me to use in the programming exercise that we have planned.

So, now let us go to our programming exercise and before that... So, observe that so far we have completed the definition of the doubly linked list abstract data type, which is very important thing. We also found the motivation for why we want to come up with a new data type. So, essentially we want to gain some efficiency and then we saw that we can reuse code that has already been written.

For example, whatever code you have written for the list abstract data type, we reuse it for the doubly linked list abstract data type and we carefully modified it, conceptually they were not major modifications. Because, essentially both are list you only have to keep track of I mean pointers carefully you need to ensure that the previous field in the doubly linked list nodes are appropriately set and so on and so forth.

So, therefore, the modifications were very simple, we just made a copy of the linked list methods dot c and change the data type names at all the arguments, the return values were changed, the function definitions are appropriately changed and care was taken to ensure that the values of the previous field in each node was meaning fully set. So,

encourage I look at the code, we will make the code available to you on Google drive as a PDF and you can take a look at it.

So, now, let us go to the short description of a programming exercises, which is a comparison between the efficiency of the doubly linked list based implementation and linked list based implementation of adding two large integers. So, that is a focus now, so let us just go to the description of the exercise which is in the power point sequence ((Refer Time: 27:29)). Question is the same; one has to write perform big number addition.

But, the goal is to compare two implementations of big number addition, the first implementation is based on the list data type and a second implementation is based on the doubly linked list data type. However, implementation of this programming exercise will printout the answer, it will also print out how many times the pointers were access by both the methods. And in a subsequent lecture we will also show you, how much additional space has been used by the doubly linked list data type.

Therefore, you will understand that trade of between using more space to gain, some more efficiency in terms of time. So, now let us go to the implementation of this performance monitor program ((Refer Time: 28:27))

So, the performance monitor program is called perf DLL dot c, it is an application program which uses both the data types, which means are if you look at the c code, you are going to see that both the data types, both the header files have been included. And the program uses the methods from methods associated with both the data types and c does not permit you to have same function names.

And therefore, we have also ensure that functions which performs similar operations have different names in the implementations of the abstract data type. Before we look at the code associate with the abstract data type, let us just take a look at one or a few runs of the abstract data type. So, let us compile it first, so let us thus look at it gcc the compiler, you compile this particular program, the output goes into this executable perf DLL and you link the object files link method dot o and the doubly linked list methods dot o.

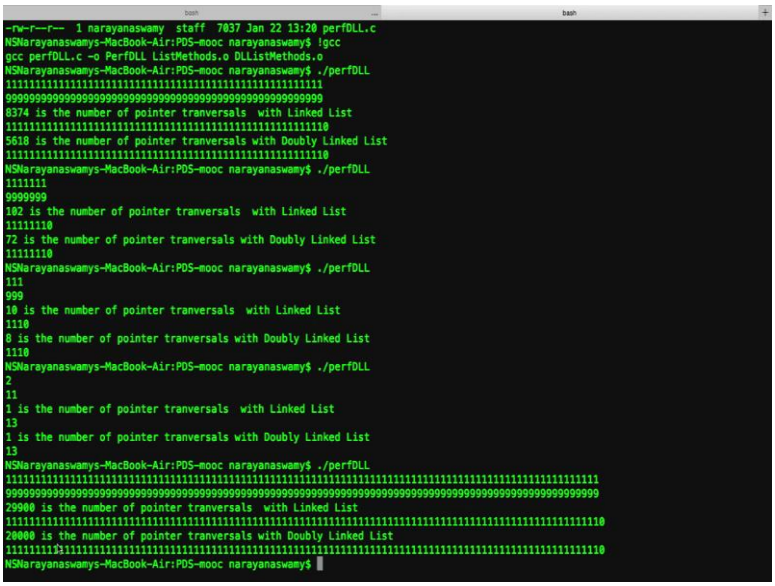
Now, the executable is ready let us write, let me give some really large number which you can verify immediately will cannot be stored in any variable of that data types provided by the C programming language. So, let us just see this, these are the two numbers it is very easy for you to verify that the sum which is in the fourth line is correct. But, trial to the fourth line that is in the third line observe that I have printed out or the program has printed out.

The number of pointer traversals with the linked list implementation that is when the two numbers were stored in two separate linked list an added, the number of pointer traversals will 8374. But, when you use the doubly linked list a number of pointers is about 60 percent with a little more than 60 percent of the number of pointer traversals, when 60 percent less than the number of pointer traversals made when a user linked list, can let me say this again correctly.

So, the number of pointer traversals made when you use a doubly linked list is only 60 percent of the number traversals made, when you perform the implementation with the doubly linked list. Let us look at even smaller numbers this came of course, sit in a appropriate an integer variable, but they not worried about it. But, again you can see that it is close to like 67, 68 percent the number of pointer traversals made with linked list is 102 and the number of pointer traversals made by doubly linked list is about 72.

Let us for even smaller numbers and as you can see now, it is almost 80 percent of the total number of , the doubly linked list users at most 80 percent of the number of pointer traversals as used by a linked list implementation let us do this for two just to see.

(Refer Slide Time: 32:05)



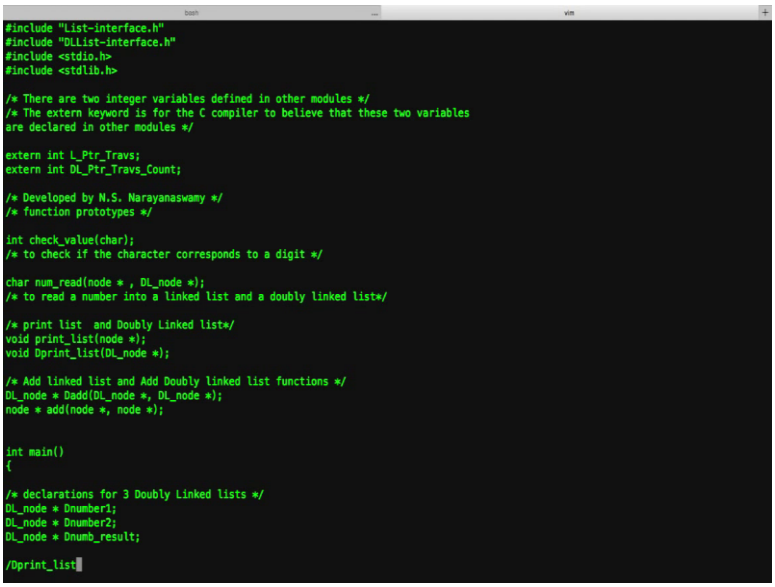
So, this is a interesting observe the number of pointer traversals in both the implementations is just see. Therefore, as the size of the number keeps reducing, observe that the implementation with the linked list and doubly linked list seems to be more or less the same, I am in it is getting close as a numbers reduce, it reduces from 65 percent onwards to something like 68 percent to 80 percent to 100 percent that is the use exactly the same number of pointer traversals.

And observe this is an experimental validation of our analysis of the qualitative effort. So, recall that we started this discussion by say, how much effort is put into traversing pointers and we did not assign values to it. But, here is an experimental validation, your essentially saying that it looks like the doubly linked list implementation users only about 60 percent of the number of pointer traversals made by the singly linked list implementation.

And it is not clear, what the exact percentages from a analysis and that is typically what is the content of a formal analysis, if you study a data structures and algorithms course such concepts are carefully analyzed in a course. But, in this course we are focusing more on the experimental evaluation. So, let us try even a larger number to see what the percentage looks like, so it almost look likes 60 percent, as you can see the number of pointer traverse it is made with the linked list implementation is about 30,000 on the other hand with the doubly linked list it is 20,000.

So, it is about 66 percent that seems to be the right number, I leave that as an exercise for you to exactly quantified that looking at the implementation. What is the percentage relationship between the number of pointer traversals

(Refer Slide Time: 34:55)



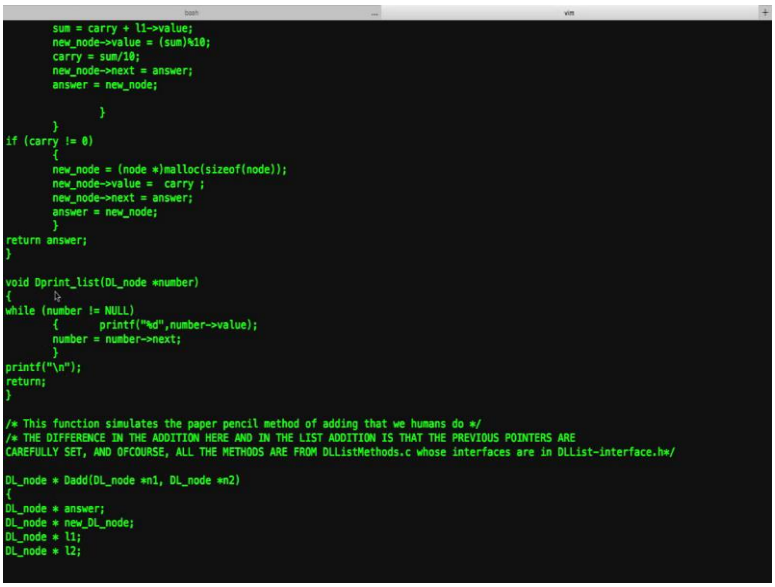
Now, let us go to the program itself, the source file as you can see, we have included both the list interface and doubly linked list interface dot h. Observe that the two variables which are the counters, which keep track of number pointer traversals and are define to be a extern variables here. This means that the compiler uses this declaration in the compilation process and observe that the variables are not initialize here, the variables are initialize in the appropriate modules.

Similarly, both these variables are defined as extern, extern is a key word which basically tells the compiler that this variable is defined elsewhere for this data item is defined elsewhere. And most slightly the good programming practices to ensure that the modification is data item is also done elsewhere, I repeat this the two counters we are using the two counters which are present in the list methods and the doubly linked list methods dot c files and they are included here with the extern keyword and this is mainly for the compiler to go ahead with the compilation.

The function prototypes are essentially unchanged, with check whether the character is a meaningful character that is be only one to deal with numbers, characters which correspond to digits that function does not change. We have are implemented the print

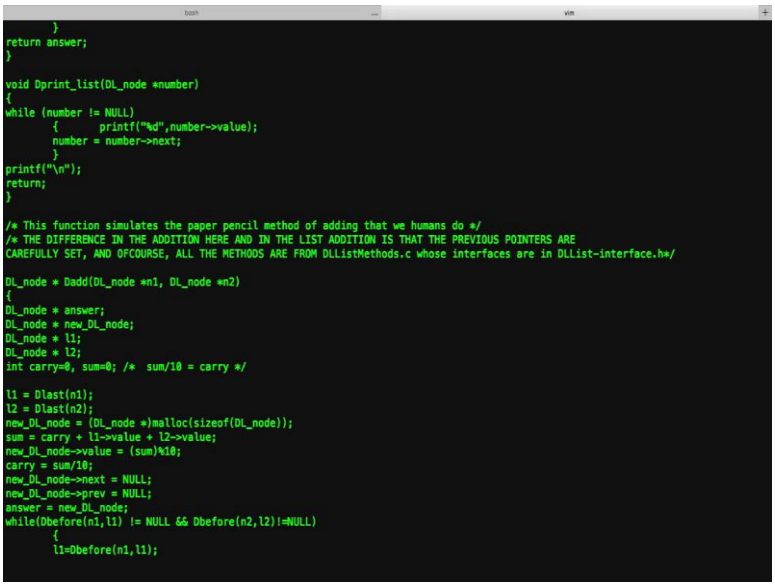
list function and we called it the d print list function and the only changes is the data type as the argument has changed and the code inside this function also is modified to reflect, let us just take a look at it.

(Refer Slide Time: 36:57)



So, if you look at D print list implementation, the argument is change and otherwise really nothing much has change, essentially we reusing the code fairly efficiently ((Refer Time: 37:14)). Similarly, the D add method conceptually where we are adding the two list, in this case the numbers are shown into doubly linked list the D add method is show add the two doubly linked list or add the two numbers in the two doubly linked list that is go to the particular code, the name of the function has been carefully changed, because of the limitations in the C programming language.

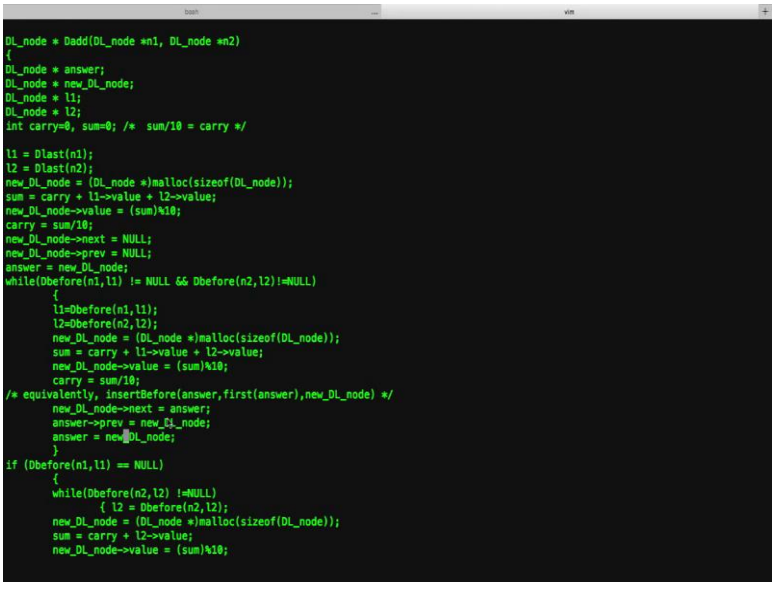
(Refer Slide Time: 37:41)



```
        }
return answer;
}

void Dprint_list(DL_node *number)
{
while (number != NULL)
        {       printf("%d",number->value);
        number = number->next;
        }
printf("\n");
return;
}

/* This function simulates the paper pencil method of adding that we humans do */
/* THE DIFFERENCE IN THE ADDITION HERE AND IN THE LIST ADDITION IS THAT THE PREVIOUS POINTERS ARE
CAREFULLY SET, AND OFCOURSE, ALL THE METHODS ARE FROM DLListMethods.c whose interfaces are in DLList-interface.h*/

DL_node * Dadd(DL_node *n1, DL_node *n2)
{
DL_node * answer;
DL_node * new_DL_node;
DL_node * l1;
DL_node * l2;
int carry=0, sum=0; /*  sum/10 = carry */

l1 = Dlast(n1);
l2 = Dlast(n2);
new_DL_node = (DL_node *)malloc(sizeof(DL_node));
sum = carry + l1->value + l2->value;
new_DL_node->value = (sum)%10;
carry = sum/10;
new_DL_node->next = NULL;
new_DL_node->prev = NULL;
answer = new_DL_node;
while(Dbefore(n1,l1) != NULL && Dbefore(n2,l2)!=NULL)
        {
        l1=Dbefore(n1,l1);
```

So, let us just look at the implementation of D add, really it is exactly the same implementation as the add function itself, except that significant care has been taken to ensure that the pointers are all appropriately set. For example, whenever a new node of the DL node type as allocated both the next pointer in the previous pointer are set to null.

(Refer Slide Time: 38:11)

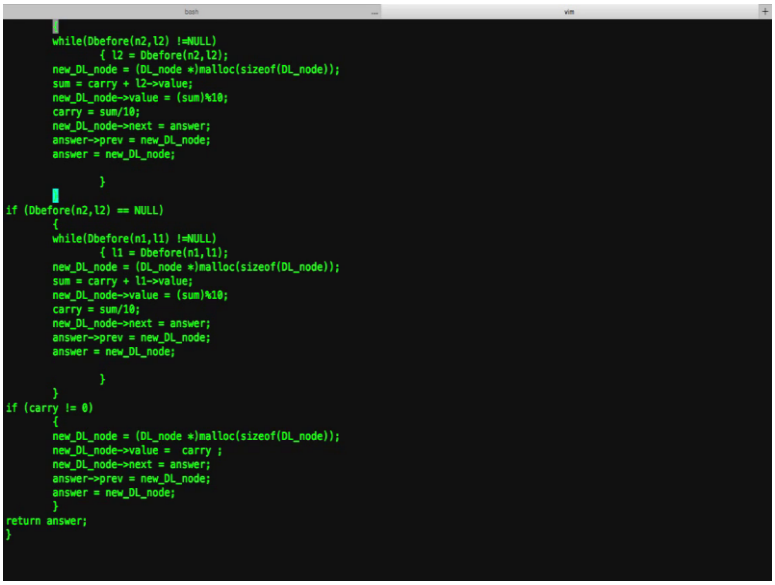


```
DL_node * Dadd(DL_node *n1, DL_node *n2)
{
DL_node * answer;
DL_node * new_DL_node;
DL_node * l1;
DL_node * l2;
int carry=0, sum=0; /*  sum/10 = carry */

l1 = Dlast(n1);
l2 = Dlast(n2);
new_DL_node = (DL_node *)malloc(sizeof(DL_node));
sum = carry + l1->value + l2->value;
new_DL_node->value = (sum)%10;
carry = sum/10;
new_DL_node->next = NULL;
new_DL_node->prev = NULL;
answer = new_DL_node;
while(Dbefore(n1,l1) != NULL && Dbefore(n2,l2)!=NULL)
        {
        l1=Dbefore(n1,l1);
        l2=Dbefore(n2,l2);
        new_DL_node = (DL_node *)malloc(sizeof(DL_node));
        sum = carry + l1->value + l2->value;
        new_DL_node->value = (sum)%10;
        carry = sum/10;
/* equivalently, insertBefore(answer,first(answer),new_DL_node) */
        new_DL_node->next = answer;
        answer->prev = new_DL_node;
        answer = new_DL_node;
        }
if (Dbefore(n1,l1) == NULL)
        {
        while(Dbefore(n2,l2) !=NULL)
                { l2 = Dbefore(n2,l2);
        new_DL_node = (DL_node *)malloc(sizeof(DL_node));
        sum = carry + l2->value;
        new_DL_node->value = (sum)%10;
```

Similarly, if you look at this piece of code when a node is inserted into the list that is when one more number is inserted into the list that becomes the head of the list. The

previous head of the list node points to the new node that has been created, this is another thing that has been taken that has been implemented very carefully.

(Refer Slide Time: 38:43)



Otherwise, conceptually the addition algorithm really does not change, the other things that has change is that whenever we needed to use the before function in the list implementation it is been change the D before and so on and so forth. Therefore, the code is just been change by ensuring that the methods associated with the doubly linked list implementation are used, what these methods actually do there is no major conceptual change.

So, now let us look therefore, we convince ourselves that the methods have an drastically changed, the implementation algorithms are just the same. And it just the careful reuse of code careful in the shames that the names of the arguments have been changed, the names of the functions have been changed. And wherever the previous pointers have to be set carefully in a doubly linked list methods, we have done that also very carefully, these are the only changes, algorithm at the there is no change between the add method or the print list method in either the list implementation or the doubly linked list implementation of this program.

So, let us just look at the top level structure of this program, the flowcharts show to speak. So, the first number is read into number 1 which is the linked list, the same number is read into D number 1 which is the doubly linked list, the first node of the

doubly linked list. Then, the num read function has been modified, it takes two arguments, one is a pointer to the first number which is a singly linked list and it takes a pointer to the doubly linked list, which is suppose to contain the same number.

And returning from this function call, the pointer number 1 contains the first number and the pointer D number 1 contains a same first number, except that in the D number 1 pointer by a points to doubly linked list, the number 1 pointer points to a singly linked list. Same with number 2, conceptually there is nothing we now have two sets of list, one set for number 1 and number 2 these are singly linked list, the other set for d number 1 and d number 2 which are doubly linked list.

Now, we calculate the result using the two functions, which is add and D add, note that D add takes the argument the two doubly linked list and add takes as the argument two singly linked list. Now, what we do is, we print the number of pointer traversals, which is the counter variable which is associated with the list methods dot c, then we print the result, then we print the total number of pointer traversals made by the doubly linked list implementation and then we print the result.

The more interesting thing for you do notice that, observe that the counters are not modified in this particular program, the program has been carefully designed. So, that the counters are modified only in the appropriate modules, this is a good programming practice. So, whenever you uses a certain module, you ensure that the module is designed very carefully to ensure that all the necessary data items associated with the module or modified only by the module and not by any other calling function.

So, this completes the implementation of the doubly linked list data type and the performance analysis of the linked list implementation and the doubly linked list implementation and a comparison between the two of them, this more or less completes the lecture now. So, observe what we have done, so for we define the abstract data type doubly linked list, we argued about it is importance and why it is important, we came up with the qualitative reasoning must.

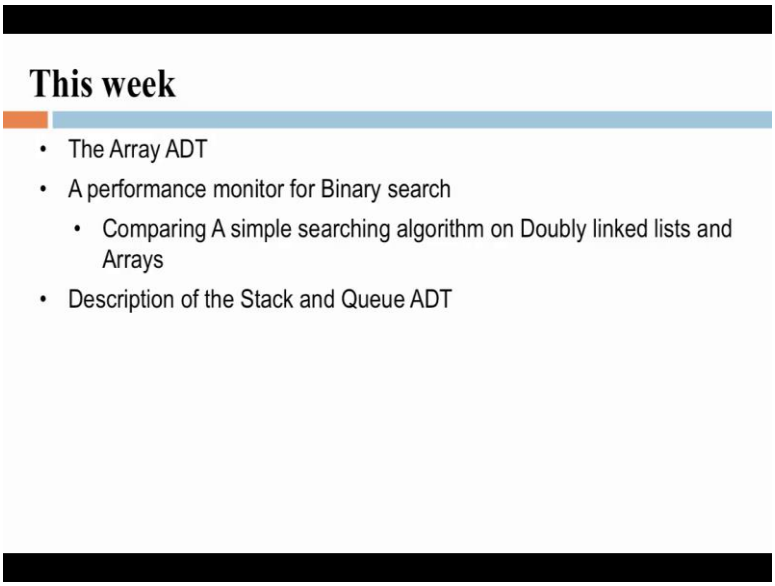
Why we need the doubly linked list data type, then we went about and disc and implemented the doubly linked list data type by reusing code from the list data type and we made the modifications very carefully. And then, we tested the implementation by making a performance analysis program, which implements the big number addition in

two ways. The first implementation source the given numbers in a singly linked list adds them and the second implementation stores the given two number in the doubly linked list and adds them and we calculated our program actually kept track of the number pointer traversals and printed them out.

That way we got a scientific method of comparing the two implementations, I means scientific because we do not do a formal proof, we only do a experimental validation. What we concluded experimentally is that, that the doubly linked list implementation traversals a lesser number of pointer than a singly linked list implementation. And it looks like that it traversals only a 60 percent of the number of pointer traversals made by a singly linked list implementation, this is a very interesting fact.

And this also shows that a doubly linked list implementation is very useful for such arithmetic purposes for large numbers. However, we did not measure how much space additional space was used, this we will definitely do in the next set of lectures this week itself.

(Refer Slide Time: 44:41)



So, what are we planning it do on the next week? The plan for the next week is that we will talk about the array abstract data type very quickly we will not go into the array abstract data type implementation. Because, the C programming language provides us features for manipulating an accessing an arrays, what we will do is, we will use the array associate with the C programming language, we will use our doubly linked list

implementation and we will come up with the performance monitor for the binary search algorithm.

We will count very carefully a certain interesting parameter which will tell us which of the two implementation is more efficient is a binary search on in array or is a binary search on a doubly linked list. Again it will only be an experimental validation and not a validation by a formal through or a mathematical counting argument. After that we will quickly described the stack and queue abstract data type in this week's lectures. And next week we will apply the stack and queue abstract data types to solve certain interesting problems.

So, that brings to end today's lecture, so I have some request from my side and the TS side to all of your attending this course. We appreciate the interaction on the forums and if we are happy to trigger such an interaction, but many of your questions requesting for extra time cannot be really honor, we have already given extra time for the first programming exercise. So, I would encourage you to take your time and do the programming exercises and not request for additional time, if you are not able to finish it within the time.

We may probably give you a approximately 2 or 3 days extra for every assignment. But, we cannot give you a large amount of time, because the course itself launch only for about 2 months and we cannot give you an inordinate amount of time, we have to move onto the next assignment. But, I do appreciate your concern that you also want to learn from the materiel and participate in the certification exam, I appreciate that.

My advice to you on that matter is that you could probably do the course now and become stronger programmers, do the programming assignments yourselves, work through the details take your own time. And appear for the certification exam at a later date, this is something that I could suggest. As one of the approaches to ensure that you get the best out of this course. And as you will see that the second assignment has been released and we will ensure that the programs that I am using to explain different concepts to you in these lectures, are available to you in appropriate documents for you to read through them and use them.

So, thank you very much and we will be uploading this lecture shortly and the next two lectures will follow shortly that brings to the end of today’s lecture.