

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

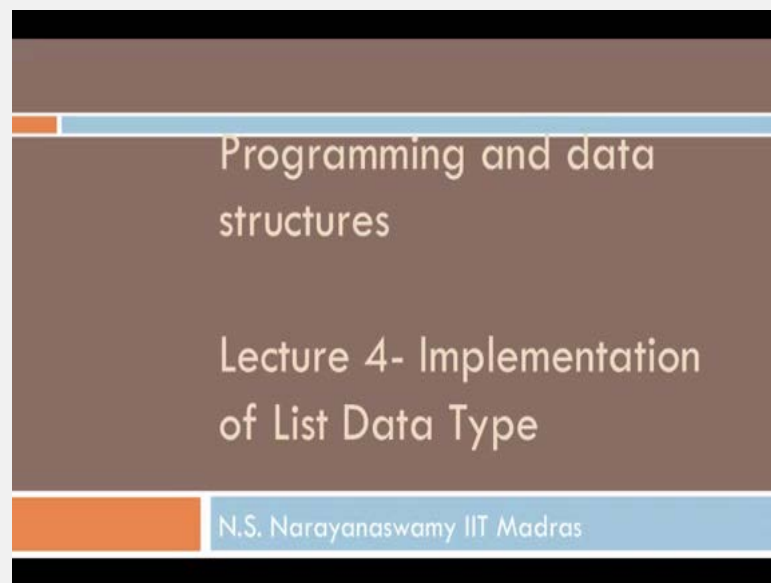
Lecture - 05
Implementation of List Data Type

Hi, welcome to lecture number 5 of this course on programming and data structures. If you recall the material that was covered in the last lecture, we talked about the formal definition of abstract data types, we talked about how to implement the abstract data types in the C programming language. In particular, we looked at the abstract data type definitions of two common data structures, we looked at the list abstract data type and the array abstract data type.

Just you recall what an abstract data type is, it is an abstract specification of a data type, it tells you what kind of data is represented in this data type and what kind of operations are associated with this data type and all these are specified without alluding into any implementation details. We will revisit this formal definition in the first few slides of today's lecture and then quickly move on to looking at an implementation of the list abstract data type and it will only be a partial implementation. We will only look at certain methods associated with the list abstract data type and they will really not be an execution today. But, we will look at the creation of the appropriate files associated with the implementation and we will also look at the compilation process and what kind of files are created as a consequence of this compilation process..

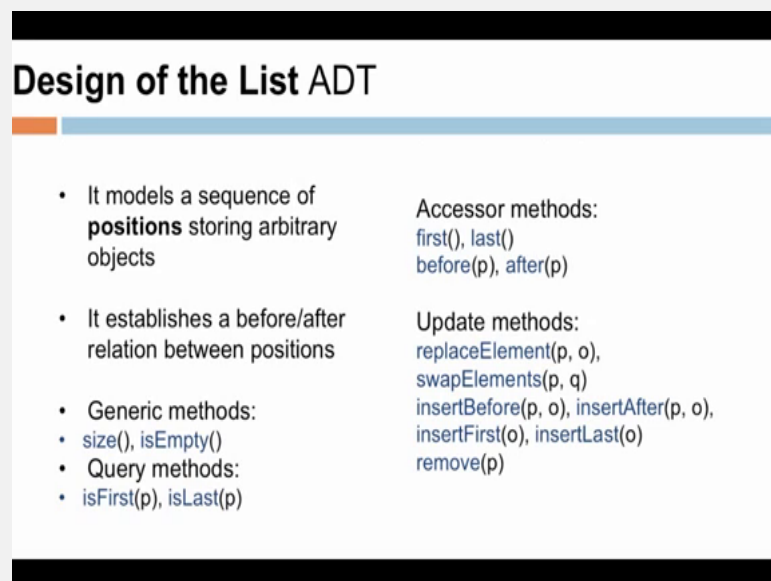
In the next lecture, we will complete the implementation of the list abstract data type and use the implementation in a programming exercise which will involve large integer arithmetic. So, now let us move on to the slides.

(Refer Slide Time: 02:06)



So, the focus of today's lecture is the Implementation of the List Data type.

(Refer Slide Time: 02:14)



Let us just recall what is the list abstract data type, consists of. The list abstract data type is used to models a sequence of positions, storing arbitrary objects, now the word arbitrary can be a bit miss reading, you can assume that it is a sequence of positions storing objects of a certain data type. For example, you can have a list which stores integers, you can have a list which stores characters, a list that stores strings, can also have a list that stores lists.

Therefore, we did not do not restrict the types of the objects, second we ask generic as

you want to imagine. The most important thing with list abstract data type is that the positions which are there in the list have a very well defined before and after relationship. That is, if you looked at any pair of positions there is a well defined before and after relationship between the two of them. The generic methods which we looked at during the last lecture or methods which return the size of the list, a method which returns a Boolean value depending on whether the list is empty or not.

We have query methods which tells us whether a given position is a first position of the list or the last position of the list. Then, we have accessor methods, the accessor methods return pointers to specific locations or specific positions on a list. For example, first and last are two methods which return pointers to the first element of the list and the last element of the list, respectively. Before p and after p are methods which return pointers to the positions, just before p and the position just after p.

Of course, if p is either the first position or the last position before and after may not be well defined and therefore, the implementation of these accessor methods will have to take care of that, after all these methods which are essentially structural queries about the list, we have update queries or update methods, where the methods that we have are the replace element method which takes as an argument a position and a value and replaces the element in position p with a given value o.

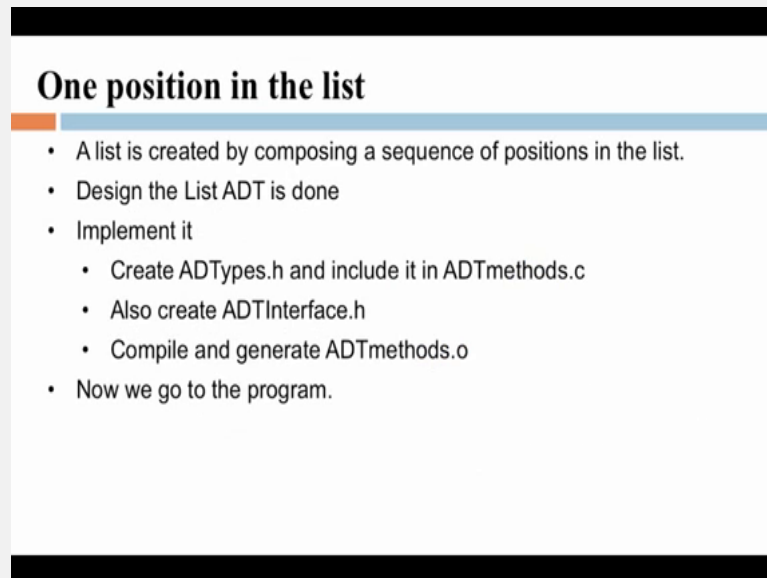
Similarly, we have swap elements which takes it is arguments to position and exchanges the contents of the two of them and insert before and insert after which take a position in an object and inserts the element in a new position just after the given position p or just before the given position p. Insert first and insert last an object creates a new position and makes it the first position of the list or the last position of the list depending on the method.

If you notice all these are methods with that increases a size of the list, the last method is one that can be used to shrink the list when necessary or remove a position, the method remove p, removes a certain position along with the data item and maintains the list. The point notice here is that after removing a certain position, the before and after relationship between the different positions are well defined. These are all the details that implementation has to take care of.

So, what I have done is I listed the different methods and specified to you, what the input output behavior of these methods are and this is exactly the way any abstract data type is

described. Without any implementation details our specification of the data that has maintained in the data type, the basic types of those data, then the appropriate methods for creating the data, creating an instance of the data type, query methods, accessing the data structure kind of methods and update methods.

(Refer Slide Time: 06:26)



One position in the list

- A list is created by composing a sequence of positions in the list.
- Design the List ADT is done
- Implement it
 - Create ADTypes.h and include it in ADTmethods.c
 - Also create ADTInterface.h
 - Compile and generate ADTmethods.o
- Now we go to the program.

So, the most important thing in the list data type, when we look at the implementation is that we are going to create a data type called a position in the list and the methods that are going to be associated with the list abstract data type will help us create the list itself by composing a sequence of position. In other words, we are going to call these positions as known's and we are going to see that the list is obtained by composing a sequence of nodes in the correct order.

Now, as you can see in the previous slide and in the last lecture, we have finished the design of the list abstract data type, now we move on to implementing it. As we discussed in the previous lecture, what we are going to do is we are going to use features of the C programming language and we are going to define appropriate header files and appropriate files where the methods are implemented, then for a programmer to use the data type, we are going to create an interface file and which is the another header file and then by including this interface header files, we will be able to use the abstract data type which is described in types dot h and the appropriate access methods and the maintenance methods which are defined in ADT methods dot c.

This is the details of the implementation at a top level. So, now we go to the program to

see what the implementation of the list abstract data type looks like and like I said before today's lecture will not involve any execution. In particular, we will not be an executable and we will not interact with that execution by giving inputs and getting outputs, what we will do is to create the abstract data type, compiled it and look at what is a kind of outputs that we have got and what it actually means.

How to use it will be presented in the next lecture via an interesting programming exercise which I will solve following that will be an equally challenging programming exercise that you will have to program and solve it. So, now we go to the program.

(Refer Slide Time: 09:06)

```
Week2-pres2.pptx          week1-rec2.cproj
abstract-data-type.pdf    week1-rec3.cproj
brute_force_string_search(1).txt  week2-lec1.mp4
brute_force_string_search.txt  week2-rec1.cproj
NSNarayanaswamy-MacBook-Air:PD5-mooc narayanaswamy$ clear

NSNarayanaswamy-MacBook-Air:PD5-mooc narayanaswamy$ ls List*
List-Interface.h          ListMethods.c          ListTest.c
List.h                    ListMethods.o
NSNarayanaswamy-MacBook-Air:PD5-mooc narayanaswamy$ vi List.h
NSNarayanaswamy-MacBook-Air:PD5-mooc narayanaswamy$ vi List-Interface.h
NSNarayanaswamy-MacBook-Air:PD5-mooc narayanaswamy$ vi ListMethods.c
```

So, let us just looked at the files that I have created all of them start with the name list. So, let we just list only those files, let us just look at the files that we have. So, as you can see there are five different files here and there also named very carefully. So, one is the list dot h and the file list dot h consists of the type definition required to create the single node, the type definition associated with the single node in a list.

The list methods dot c is a file that consists of the implementation of the different methods associated with the list data type that we have defined. Then, the list interface dot h is a file that will be included by programmer's who want to use the list data type. So, in particular the most important files that you will have to create when you create a new data type or the list or the abstract data type methods dot c, the abstract data type interface dot h and the data type dot h itself and we will see further details of it by going to the files themselves. So, let us just look at the header file list dot h.

(Refer Slide Time: 10:59)

```
/* List.h created to define the node data type and this will be used in the  
Listmethods.c */  
  
typedef struct container {  
    int int_data;  
    struct container *next;  
} node;  
  
#endif  
#endif
```

As you can see that is in this case, there is a type definition and let us just look at the type that we have created, we have a created type called a container. We refer to this as a struct container, this a new data type that we have created. The data type struct container has two fields, one is of integer type, the other one is a pointer to struct container and this pointer is given the name next and for the easy of programming, we have given a new type name for struct container and we call this the node type name and this is done by using the typedef keyword associated with the C programming language.

And this is essentially what is there in the list dot h and as a generic principle when you implement an abstract data type, whatever structures that you need to implement the associated methods can go into such a header file. Let me repeat, such a header files will not be included by the programmer, who wants to use your data type. A programmer who wants to use your data type will only include the interface header file as we will see next.

So, this is for the person implementing the details of the abstract data type, in particular he or she is specifies the different kinds of data associated with the data type. So, here we have specified that each object in the list consists of one integer data item and a pointer to another object of the struct container data type. This is what we have specified here. So, therefore, we have specified the kind of data and now you would want to know what are the different kinds of methods and how are the implemented. We would see them of course, in the slides. So, let us take a look at the prototypes of the methods that are provided along with the implementation of this data type ((Refer Time: 13:09)). Let us look at the methods. So, this is in the interface file.

(Refer Slide Time: 13:24)

```
/* ListInterface.h created with all the function prototypes */
#include "List.h"

/* returns the number of elements in the list */
int size( node *);
int isEmpty(node *);

/* Decision questions given a pointer */
int isFirst(node *);
int isLast(node *);

/* function to return pointers to specific queries */
node * first(node *);
node * last(node *);
node * before(node *);
node * after(node *);

/* Update methods */
void replaceElement(node *, int );
void swapElement(node *, node * );
void insertBefore(node *, int );
void insertAfter(node *, int );
void insertFirst( int );
void insertLast( int );
void remove(node *);

/* Did you notice that there is no function to create a list. You can do
that by a function when you want to */
"List-interface.h" 28L, 744C
```

So, this list interface dot h is created with all the function prototypes and let me assert these are the function prototypes and this is essentially the file that a programmer who wants to use your abstract data type, that he or she will look at, that is if I want to use the list data type that we are implementing at the moment, then the file that I will look at to understand what are the function calls that I can make to access features of the list data type, I would look into this particular file.

So, let us just look at what this particular file consists of, remember that it is an interface file, first of all we include list dot h. So, we include the header file list dot h which consists of the definition of the node data type. Let us look at the different kinds of methods that we have. So, we have methods which return the size of the list which returns the Boolean value which is taken to be an integer which is a return value is considered to be of integer data type, the function is called isEmpty, the argument is a pointer to a node.

Similarly, isFirst and isLast are two methods that return a true value, this should of course, have two arguments. The first argument is the pointer to the list itself and the second argument is one's specific location and the query is whether it is the first or the last. Therefore, there must be two arguments here, then they are functions which return pointers on specific queries. So, let us just look at the function first as a representative example, it takes as an argument a pointer to a list, where each element is a node and then returns a pointer to the first element of the list.

Similarly, the last function returns a pointer to a last element of the list, that before function also takes two arguments, one is a pointer to the list itself and a pointer to a specific location and queries whether the queries for a pointer to the location before the given location in the list. Similarly, the after function is a query for a pointer to the location just after the given location in the list. We would look at the implementation of these functions.

Today we will not look at the update methods we will look at them on a next lecture, but as you can see the prototype definitions for the update methods are also here. There have been many request on the forum for these programs, by the middle of next week I will definitely put this programs up on Google derive and ensure that these are accessible to every resistant in this course and the same holds for the slides also.

However, it is important for you not to directly use these programs and write use these pieces of code and write your programs, it is important to write programs from scratch when you are a learner of the subject for the first time. However, they are made available to use. So, that you can read them and understand them at your lecture. So, what is this particular file? This is an interface file and let us recall the programmer who wants to use features of this particular data type implementation uses this interface files by including this file in his or her program.

And look at the most important thing here, the first thing that you include is the appropriate types that tell you what kind of data is manipulated by this particular data type that is what is included So, we have taken a look at two very important files, ((Refer Time: 17:58)) the first of these was for the person was creating the data type, the second of these is for the person who is programming, who is using this particular data types implementation in his or her program, thus the interface file.

Now, the person who creates the data type itself also writes another C program in which all the appropriate methods are defined precisely with a appropriate written types. Remember, that the header files consists of the interface dot h file consists of the function prototypes, this file the list methods dot c consists of the implementations of the different methods which are in the interface file.

And typically a programmer who wants to use the abstract data type will not look at the implementation in this C file, he or she will only use the interface file and program carefully. This is a good programming practice and like I ask in a last lecture, it is an

exercise for you to find out, what is this very popular programming approach.

(Refer Slide Time: 19:16)

```
#include "List.h"
#include <stdio.h>
#include <stdlib.h>

/* returns the number of elements in the list */
/* note the usage of recursion */
int size( node * head )
{
    static int count=0;

    /* C programming feature that keeps count value form previous function calls*/
    if (head == NULL) return count;
    count++;
    return size(head->next);
}

int isEmpty(node * position)
{
    if (position == NULL) return 1;

    /* NULL is a constant address defined in stdlib.h
    meaning, DOES NOT POINT TO ANYTHING */
    return 0;
}

/* Decision questions given a pointer */
int isFirst(node * list, node * position)
"ListMethods.c" 100L, 2154C
```

So, now let us move to the list methods dot c file. What do you will see here is that all this structural functions I mean those that do not update the data structure that we have setting up, all those methods are here. And each of these methods you will see it is a very crisp and short piece of code and that is how programming is made modular. So, you break your programming exercise into small quickly implementable functions and verifiable functions and then put them all together carefully in a separate module.

I repeat, a good programming practice is to understand the problem and break it down into small quickly implementable functions, whose correctness can be independently verified and then you use these functions that you have implemented to create larger programs to solve other related programming exercises. So, let us just look at the c file in which we have implemented all the access methods associated with the list data type, as you would have expected, we have included the file list dot h.

Now, observe that we have separated into two files, the data associated with the list data type and in this file we only have the methods associated with the list data type. This is a good programming practice nothing prevents you from putting in the contents of the list data type here too. However, if your data type is very sophisticated and has many fields and o, on and so, forth, it is a good programming practice to put the data in a separate file and the methods in an associate file. The method implementation in associate file and like we have done the interface functions, the prototypes of the interface function in at

third file.

Now, here we have use the `stdio dot h` and `sdtlib dot h` it really I think though I have put in the `stdlib io dot h`, it is not necessary in this implementation as if not, but I will leave it there. So, let us look at the first function, the first function if you recall was the function that return the size of the list. So, let us just look at the function. So, the return value is very clear, the return value is an integer, the name of the function is `size`, the argument to the `size` function is a pointer to node and this pointer is called `head`.

And let us understand how this functions will be called, though we are not writing the program which is calling it and this is a reason why in today's lecture, we are not going to compile the program, getting an executable and work with it. The motivation is to actually go through the implementation without making function calls and generating and executable. So, now let us just visualize how `size` is going to be called. The argument that `size` will be called will be a pointer to the first element of the list.

So, let me repeat. So, when you want to compute the size of a list you call this function from whatever program you are writing. The argument to the `size` function is a pointer to the first element of the list, whose size you want to query. Now, let us just look at this particular implementation, the implementation is a very crisp and short implementation, you will observe that it is a recursive implementation.

The setup a variable called `count`, `count` keeps track of how many locations are there in the list and let me call this list `head`. So, `head` is the list it is a list of nodes and we want to count the number of nodes which are in the list, which I am going to refer to as the `head`. Now, observe that this a recursive function as I just said, `count` is an integer variable, it is initialize to 0; however, there is an additional keyword called `static int`.

Whose keyword is a C programming feature that keeps the value of `count` in previous function calls to the `size` function and you can increment the value in every function call or you can modify the value in every function call. After you return from a function call, the value of `count` will be appropriately modified. So, in a way you can think that there is a actually just a single `count` variable for all the recursive function calls that we make to `size`.

So, let me repeat this we have a integer variable called `count` it is initialize to 0. So, much is clear. However, there is an additional keyword called `static`, the word `static` means that whenever a function called to `size` is made, the previous value of `count` is retained for use

in that particular function call. Of course, if it is a first function call to size then count get initialize to 0, subsequent function calls to count will have the value that was subsequent function calls to count when the control enters the function will have access to the value of count from the previous function calls.

So, let us look at this recursive function. So, if head points to a null list, null is a keyword in the stdlib dot h is a constant address, the mean of null is that it does not point to any thing. So, if head does not point to any thing, then you just return the value of count, remember there it is a recursive of program. So, you enter this particular function call, you make a function call into size, you enter with this certain value head, if head is a null that is does not point with anything then you return the value count.

So, let us just understand this when the first function call is made, if head is null that is it does in point to anything the observe the count is 0 and the value of count return correctly. If the list has like 20 positions and you have finished your head points to the 20th element or the head points to the 20th element at that time when the control enters this particular function count to the had the value 19 and head is not null because it is points is 20th element. So, count gets incremented.

Now, observe that we make a function call hear, calling size head pointed or next. So, if head is last element of the list, head pointer or next we will ensure very carefully with the head pointer or next will be null, that is we will mention there it will does not point to anything, that case it will be null. So, this recursive call will then return the value of count.

So, tomorrow when we meet we will look at an implementation how we will look at a run by making of a function call to this particular function and we will see how the recursion actually works. Let us go to the next function, the next function basically is empty function takes or pointer to the list and if it is null it returns 1, same there it does not it is empty; otherwise, it returns 0 same there it is not empty, that is a position is point into some element, then it will return a value 0 responding with the answer that the list is not empty.

So, the implementation for this very straight forward, if position is null then position does not point anything meaning full. So, you just say that position points to an empty list and you return one. Otherwise, if that if this condition is false it means that the control does not evaluated this return 1, it comes here and returns 0, which means that

position is not null which means it points to something or other it points to nonempty list.

So, the next queries to the is first function, the argument to the is first function is a list itself, that is we give a pointer to the first element of the list. Otherwise, the head of the list and we have a pointer to some position in the list and the query is, is that is position the first element of this list, that is the query. So, let as just look at this it is very straight forward, when this position the first element of the list. If position will list point to the first element of the list, then position is the first element of the list or other position points to the first element to the list.

Therefore, the check that you do is very simple, you check if list is equal to position in value that is a point to the same thing, then you return one otherwise, you return 0. Therefore, this is the single comparison and if it succeed you return one saying that yes, position is the first element of the list; otherwise, you return 0 saying that, null position is not a first element of the list.

So, now we given a position and a list and we are query and now we are implemented this piece of code to check if position is the last element of the list. Now, let us just take a look at this, if indeed position is a last element of the list, then the position pointer at next is null it should not be pointing to anything and therefore, you return 1. So, let us just understand this position is a last element of the list, if position pointer or next does not point to anything meaning full in otherwise position pointer dot next takes a value null.

If not a position pointer dot next is not null there it means that, there is a well defined successor for position. Let me repeat this, if position pointer dot next is not null, there it means there is a well defined successor for position and he return the value 0.

(Refer Slide Time: 31:50)

```
/* Note the misleading name, it can be used in other ways too */
/* if list points to the first element of the list, then this gives the
correct answer */

if (list == position) return 1;
return 0;
}

int isLast(node * list, node * position)
{
    if (position->next == NULL) return 1;
    return 0;
}

/* function to return pointers to specific queries */
node * first(node * list)
{
    return list;
}

/* returns the list itself, assuming that list points to the
first element of the list */

node * last(node * list)
{
    if (list == NULL) return list;
    while(list->next != NULL)
        list = list->next;
}
```

So, now let us look at functions the whole class of functions that return a pointer on specific queries. So, let us just look at the function called first which takes it is argument a pointer to the first element of a list and we call that pointer itself list. And we want the function to return a pointer to the first element of the list, well this is very easy. Therefore, you just return list itself and one has to be very careful or assumes that list points to the first element of the list.

Of course, if you want to misuse this function you can misuse it, but it is not a very dangerous piece of code. So, it is a very straight forward function it returns the argument itself, there is an underlying assumption that when a programmers calls is function he or she understands it function very precisely and passes pointer to the first element of the list and the return value is the first element of the list itself. The return value for the last element of a list and how is a list given, the list is given by a pointer to the first element of the list.

Now, what is a return value? The return value is the pointer the last element of the list, how does an achieve this, we write this iterative function. So, if list is null then you return list itself. So, this is the well defined exit condition for function calls, where list is when that parameter which is passed is null. So, then you return null itself; otherwise, that is if this function was not, if this statement was not executed then control comes to the next statement, which is a while loop, the next statement is a while loop.

And let us look at the condition that is check here, the condition that is check here is to c

if list pointer of dot next is not equal to null. So, this is an iteration where you can visualize this as the pointer moving right through from one location from the head of the list all the way up to the tail of the list. So, if list pointer dot next is not null, then list is made to point to the next element in the list. When do we exit from this particular while loop, when you are at a node that is when list is pointing to a node, whose next pointer is pointing to null.

In other words when list is the list is pointing to the last element of the list, that is you skip till the last node is reached and here we return. So, what do we have?

(Refer Slide Time: 35:13)

```
/* note the usage of recursion */
int size( node * head )
{
    static int count=0;

    /* C programming feature that keeps count value from previous function calls*/
    if (head == NULL) return count;
    count++;
    return size(head->next);
}

int isEmpty(node * position)
{
    if (position == NULL) return 1;
    /* NULL is a constant address defined in stdlib.h
    meaning, DOES NOT POINT TO ANYTHING */
    return 0;
}

/* Decision questions given a pointer */
int isFirst(node * list, node * position)
{
    /* Note the misleading name, it can be used in other ways too */
    /* if list points to the first element of the list, then this gives the
    correct answer */
}
```

So, for we have queries which returns numeric values like size, then we have queries that return Boolean values as to whether a list is empty or whether a certain position is a first element of a list and whether a certain position is a last element of a list, then we have functions which return pointers to specific locations which are given in the list, that is the first element of the list and returns pointers to the last element of the list.

Now, let us look at two functions, one which returns the predecessor of a position in a list and another which returns the successors of a position with list. So, this one returns a predecessor, the before function and the after function returns a successor of a position inside a given list. So, let us look at the arguments to the before function as an example, it takes as a argument a pointer to the list itself, that is it is a pointer to the first element of the list and then it takes a candidate position, it takes a pointer to a candidate position and it returns a pointer to the position, which is before the given position.

Of course, if the given position is the first element of the list or if the list itself is empty, that is the list is pointing to the nothing, then you return null immediately saying that immediately responding with the answer that your query is meaningless I cannot give you any meaningful pointer on these question. So, if list itself is empty or if the given position is the first element of the list, then return null.

Observe that, we are using the method that we have already implemented in this particular file, then you return null. Otherwise, you skip to the node whose successor is position, that is you do you have a while loop where that condition that is checked is whether, list pointer dot next points to position. If not you skip, in other words list points to the next element, keep repeating this iteration. Till, list pointer dot next is the given position at which point of time, you return list and that could be the position before your given position.

(Refer Slide Time: 38:00)

```
}  
  
node * before(node * list, node * position)  
{  
    if (list == NULL || isFirst(list, position) ) return NULL;  
    while (list->next != position)  
        list = list->next;  
    /* skip to the node whose successor is position */  
    return list;  
}  
  
node * after(node * list, node * position)  
{  
    if (list == NULL || isLast(list, position) ) return NULL;  
  
    /* if list is empty or if position in the list is last, then  
    it is invalid, as there is no successor */  
  
    while (list != position)  
        list = list->next;  
  
    /* skip till you reach position, and then return the next element */  
    return list->next;  
}  
  
/* void replaceElement(node *, int );  
void swapElement(node *, node * );
```

Similar condition holds for the after function which takes in a position, in a list and returns a pointer to a node which occurs after this position in the list. Now, let us look at the boundary conditions, if the list itself is empty or if the position given is the last position on the list. Observe again, that we are using a function that we have implemented in this file of methods associated with the list data type, then you return the value null; otherwise, you keep skipping till the list is the position itself.

So, that is you keep checking if, list is same as the position if not then you skip, that is you make list point to the next element in the list. When you come to look at the position,

then you return the list pointer dot next, that is you would return the successor the next element that list is pointing, that is return list pointer dot next. Of course, you must worry about why is list pointer dot next not null, because if list pointer dot next was null, then is list position would have been true is last in list position that is the given position the last element of the list would have been through.

Therefore, this logical expression would have been true and you would have return the null already. Because, the control did not go through this particular if statement, it means that position is not the last element of the list, nor is the list null therefore, you will return a pointer to a valid node in the list, then more vales brings as to the end of today's lecture.

(Refer Slide Time: 40:09)

```
return list;
}

node * after(node * list, node * position)
{
    if (list == NULL || isLast(list, position) ) return NULL;

    /* if list is empty or if position in the list is last, then
    it is invalid, as there is no successor */

    while (list != position)
        list = list->next;

    /* skip till you reach position, and then return the next element */
    return list->next;
}

/* void replaceElement(node *, int );
void swapElement(node *, node * );
void insertBefore(node *, int );
void insertAfter(node *, int );
void insertFirst( int );
void insertLast( int );
void remove(node *);

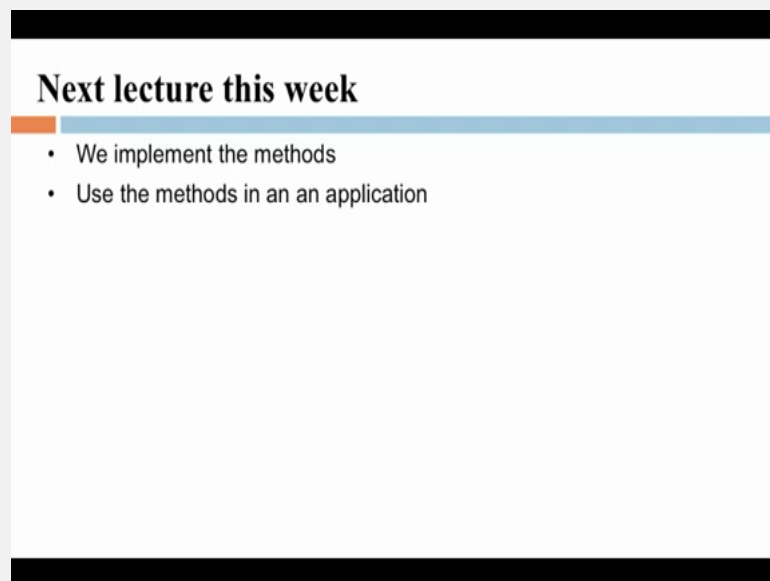
Did you notice that there is no function to create a list. You can do
that by a function when you want to */
```

Let us arbitrarily short and tomorrows lecture, we will look at the implementation of these methods which are use to update the list or modify the list and this is how the data structure grows and strings. So, therefore, the list that we will create will be the data structure which stores certain values and we have the date structure that we have is an incidents of the list abstract data type. And we have implemented sum of the methods that are associated with this data type.

When we meet in the next lecture, we will look at an implementation of the update methods associated with this list data type, there will be two programming exercises, one and data I will that show you on the next lecture and other one where you will have to program essentially with minor modifications to the list data type and both of them will

definitely make you comfortable with programming with the list data type.

(Refer Slide Time: 41:20)



The slide features a black header bar at the top. Below it, the title 'Next lecture this week' is displayed in a bold, black, sans-serif font. A horizontal bar with an orange segment on the left and a light blue segment on the right is positioned below the title. The main content area contains a bulleted list with two items: 'We implement the methods' and 'Use the methods in an an application'. The slide is framed by a black bar at the bottom.

Next lecture this week

- We implement the methods
- Use the methods in an an application

So, what would be in the next lecture we will implement the methods associated with we updates of one list of the list data type or one list data structure and we will use these methods in a interesting application and the application will be or application to multiply large integers. So, that brings as to the end of this lecture I hope things where clear and I look forward to a comings on them soon.

Thank you.