Programming & Data Structures

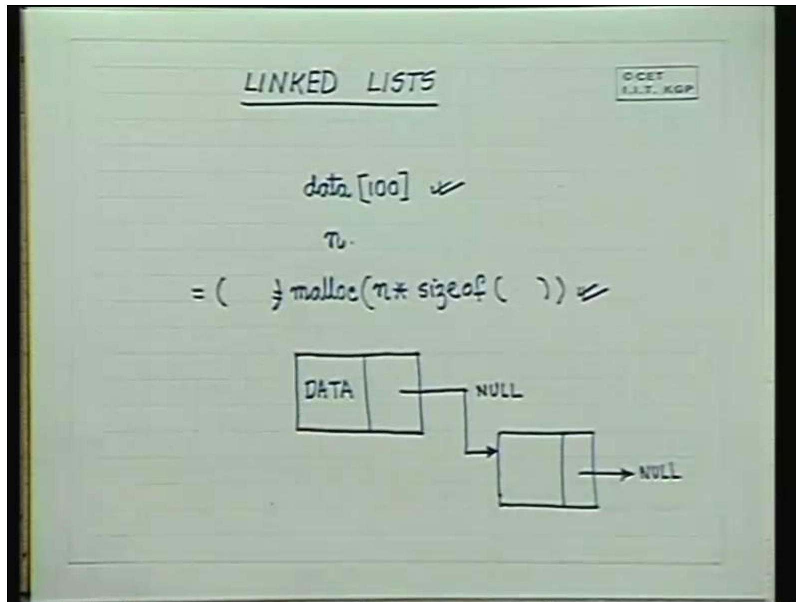
Dr. P.P. Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture 17

Linked Lists -I

The topic of today's lecture is linked lists. We discussed in the previous day that if we have a problem in which we have to read n numbers and just print them and we are initially we know the maximum value of n then we could have read them into an array. So we could have read them into array and declared an array, a data. If we knew a maximum limit we could have declared a data size 100.

(Refer Slide Time: 03:28)

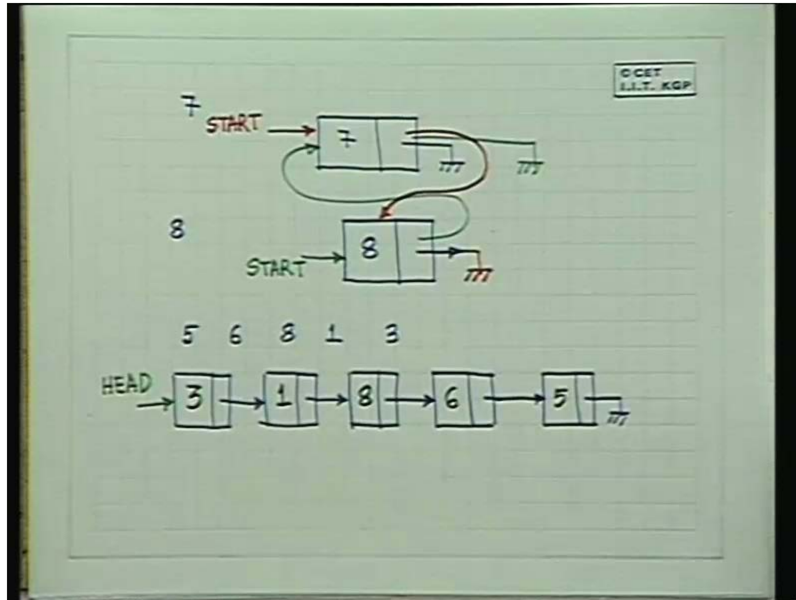


If we did not know a maximum limit and we have to read in the number of numbers which are read say n. Then using dynamic allocation if it is an integer, we could have dynamically allocated n into suppose we are reading in whatever structure we are reading in, you could dynamically allocated this size and read it into a pointer to that size after typecasting.

On the other hand if we now come to the most general saying where you will read in one data and then to ask whether to read in another item or not, neither can you have a pre declared value like this nor can you malloc a block of data dynamically after reading the value of n because the value of n is not known till the end of the data. At that point of time, we have to design a structure in which the data would be read and once this data is read, the next data item would again be have to be read. For that we said that we will have a structure of a node which is like this. A node will have two parts, one part will store the data and the other part will store the next data which comes. If there is no other

data, this other part will be null or grounded or zero or whatever. If there is another data then the other part will be linked to a block which is exactly like this.

(Refer Slide Time: 07:24)



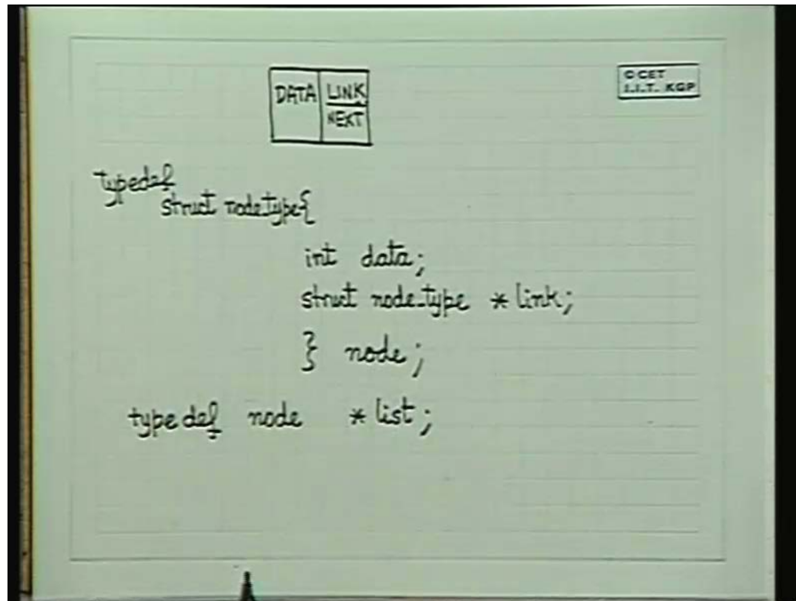
So once we are able to design such a structure, suppose we read in the first number 7 then we will create or we will dynamically allocate a structure like this, put the data value seven and put this to be null. The next item which is read, suppose next item read is 8 then what, we have two options, we can, we will allocate another piece of data make this 8. And we have two options here, one option will be to make this element point here and ground this element. The other option can be to make this element point here and have this grounded, just depends on the order in which you store the elements. And you will maintain a pointer in the situation where the color is green which will be called the start of the set of elements which are linked up.

In this situation this will be called the start of the head, whatever and while you are reading in the data after having read in, suppose you read in a set of 10 elements and then you don't want to read anymore and suppose you have read in the elements 5 6 8 1 3 and then the user says I want to enter no more data, then this linked structure in the green format where it is put in the front, that is the new element is put in the front and in the red structure the new element is put at the end. So if we put it in the front then this whole thing if this comes first, this comes second then after the whole linking is done will look like this. This will be the start of the head of the list and the data elements will be stored because we are putting it in, the new element is put in the front and this will be the structure of the data which is put in. So this will be a linked list that is why the elements are linked up by a set of pointers. So this is called a linked list structures.

There are various forms of linked list structures but this is the basic format and in this basic format, we have to decide what is the declaration of a structure like this. So the first

problem that we have to solve is how to declare such a structure that is point number one and then the second is how to write a program which will read in the data and store up such a structure. Once we are able to store up such a structure then we can manipulate it and we will decide on algorithms to manipulate such a structure.

(Refer Slide Time: 10:54)



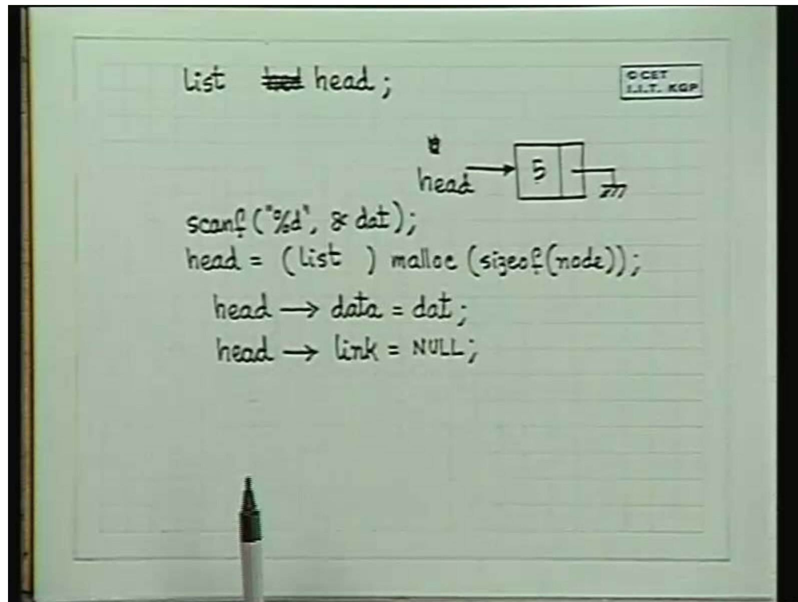
So the first issue that we will tackle is declaring a structure. So our structure conceptually consists of two parts, one is called the data part and the other is called the link part or the next part. So inside the structure definition, we will have int data and corresponding to this link or next or let us call it link or next any one of them. Then this will have a declaration which is a pointer to a structure which is of this type this. This field is a pointer to a structure which is of the same type as this. So now we end up with a recursive definition of such a structure. So this will be struct say node type, if this structure is node type then this will be a pointer to that.

So this is what it will be. It will be a pointer to node type but remember node type is this structure itself. So we have to declare this node type struct node type. So struct node type int data and link is a pointer to node type and you can declare all your variables of this structure. Suppose you declare node, node will be of this structure and if you want a type definition then you can just put in a type def and call this node then node will be a type declaration of this type, alright. So this is the type declaration of this type. Now the link list on the other hand the head, the head of the link list will be a pointer to this node. The head will be a pointer to this node. So any link list is not a node but a pointer to a node. The link list, any variable which defines a link list is not a node but a pointer to a node.

So, if we define the type definition of a link list or if we define a variable also but let us define the type of a link list. So our list is a pointer to type node, so this is the type

declaration for lists. So now once we have got the type declaration for a node and the type declaration for a list, now we should be able to start manipulating our data.

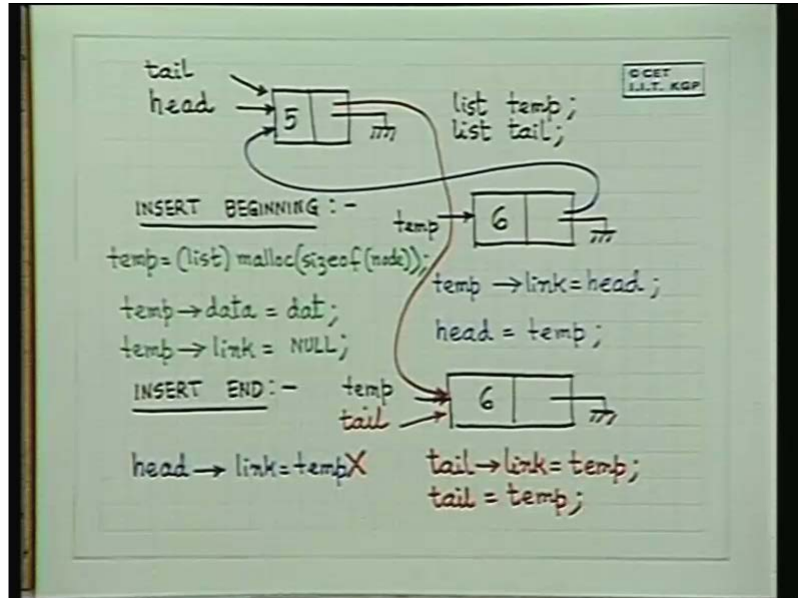
(Refer Slide Time: 14:05)



So let us try and see what we will do. First let us see diagrammatically what we will do. So let us define some variables, let us define list, head of a list. We define a variable head of a list which is the starting of the list. And in the beginning when the first element is read what will we do? We will allocate one element for head of the list. So to allocate an element like this where this points to head and this will be grounded and you will read in your first data here say five whatever, whatever is read in we will be reading like this. So for that we will have to say head is equal to malloc size of we are going to allocate one node of this size. Isn't it, so size of node. And can you tell me what this will be, node start or list we have defined list to be node start type, so either you can write node star or you can write list but since we have made a type of list of this way, we will call it list. But once you do this you have to now initialize these values.

So before this suppose we have read in scanf % d and data, dat into an integer variable. Then what we will have to do? Head pointer dot because head is a pointer. So head pointer if I want to access this data field is equal to and if I want to and I will have to make this ground, make it ground what I have to do is, this was link field. Isn't it? So this is how I allocate my first element. So going back, this is how we define type def struct node type int data struct node type pointer to link node and type def node list. So list is another type, we defined head to be a temporary element and we allocated this.

(Refer Slide Time: 20:26)



Now we will see how we will take in the next element. So at this point of time, our structure is something like this. Now suppose he says yes I want to give another element. No means obviously you have already got your structure and head is the starting of that structure. Suppose somebody wants to read in another element then what we will do? Suppose we will have to read in another element. I will give you a simple way of doing it let us declare another variable temporary variable of type list. So what we can do is suppose we have two options, we can put it at the end or we can put it at the beginning.

So suppose we want to insert in the beginning. If we want to insert in the beginning then we will have to schematically or diagrammatically we will have to allocate one element into say temp. We will ground this, we will put in the next data and then we will do some manipulation. If we want to insert it in the beginning then this link will be put here which means temp pointer dot link is equal to head and the next head will be moved here because now this becomes the head of the list, so head is equal to temp. This inserts it in the beginning.

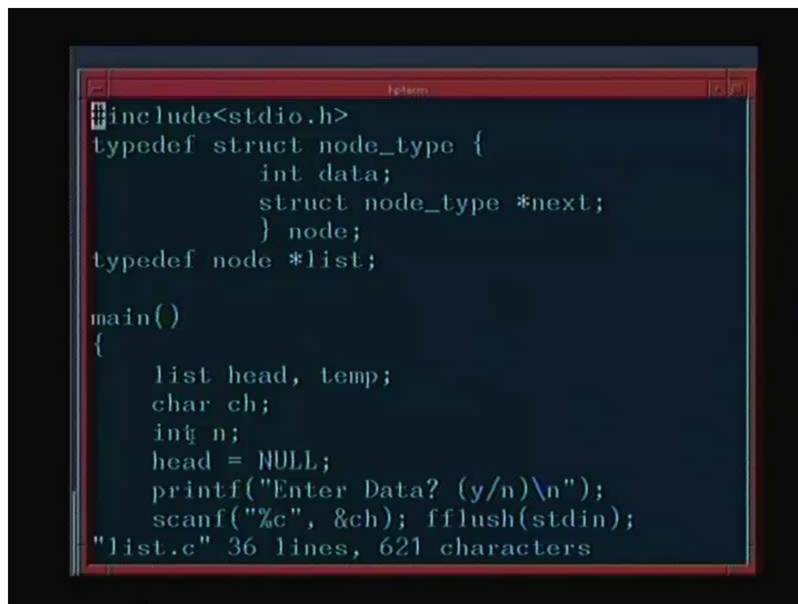
If we want to insert in the end, so before this the common call before this is temp is equal to list malloc size of node temp is equal to list malloc size of node is first point. Temp pointer dot data is equal to dat and initialization is very important, we need not put this to be ground if you know you are going to do this. But it is my advice that you always make it a practice to initialize whatever you are doing. So this is the first thing and then if you want to insert it in the beginning you do this, alright.

And if you want to insert at the end and you have got a structure like this, you have got a structure like this. You have to, this is the list. If it is only one element there is no problem then head dot link will be assigned temp, that's all. So a very simplistic way of arguing is head dot link is equal to temp and you get a structure which is something like

this, in red it will be something like this. But next time suppose you want to allocate another element then head dot next will be assigned, temp will not work. It is actually the end of the list, wherever is the end of the list that should be assigned dot next. So we have to maintain along with head, we have to maintain another variable which marks the end of list. So let us call it then tail of the list. When there is only one element, head and tail are identical. So here what you will do is instead of doing this part, all that you will do is this. This is not will be done. Tail dot link is equal to temp and the next will be tail is equal to temp. So tail dot link will be temp and this will become tail. So whenever the next element comes in, again tail dot next will become temp and tail will become… Here you could have also written tail is equal to tail dot link which is the same as temp because these are both identical. And this you can put into a loop, this part or this part which will continue adding and inserting data.

So once you have added and inserted data like this then you will get a link list of this type with the head element and obviously if you require the tail element will be here. At any point of time, the head and the tail but one will read it in the reversed order, the other will read it in the direct one, alright.

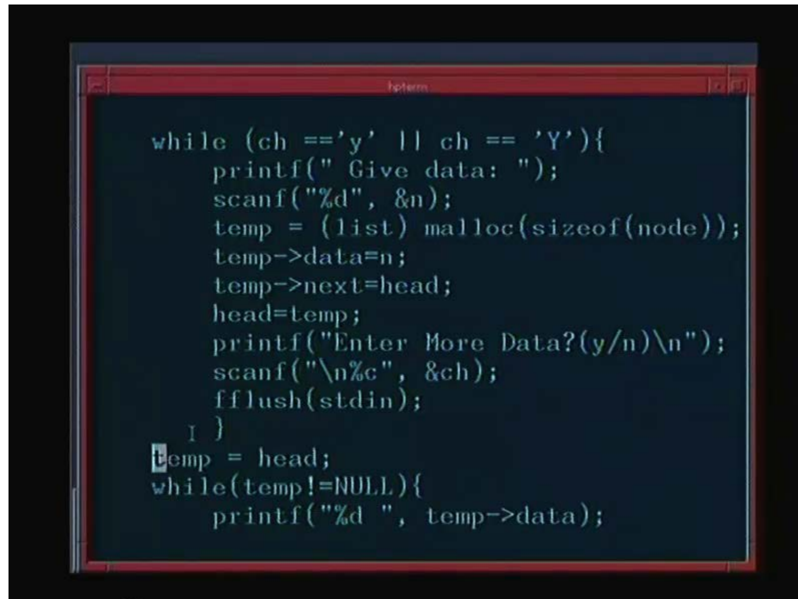
(Refer Slide Time: 21:21)



So today let us see the final programs to do this. This is the program which will read the elements in the reverse order that is insert in the beginning, this program inserts in the beginning. So this is the first one type def struct node type, sorry int data struct node type next pointer to next instead of link equal to next and list is of type pointer to node. And since we are inserting in the beginning, we actually did not require the tail element. So, here we are using just the head element and we are reading in one character to mark whether we want to read in another data or not. So we initialize head to null where no element is read and we ask whether you want to read a data or not. If it is yes, as long as the character read is yes we enter the data. Temp is equal to list malloc size of node, temp

dot data equal to n and temp dot next is equal to head. Here we directly read it, we could have done temp dot next is equal to null and then as we were saying that we will be inserting in the beginning, we will do temp dot next is equal to head and then head is equal to temp, that's all.

(Refer Slide Time: 23:26)



This is sufficient, temp dot data put comes here temp dot next points to the head of the list. So now we have inserted in the beginning of the list and head is now made temp. So this way we are able to insert in the beginning of list, enter more data yes or no, read the data and continue. So while you have finished this, this part builds up the whole link list from here to here this builds up the whole link list, alright. And finally if you want to print the link list, start from the head. So we use this temporary variable temp is equal to head, we don't want to modify it that is one. While temp not equal to null print temp dot data and temp is assigned temp dot next temp pointer dot next. So we will continue to traverse the list and print the data as long as temp does not equal to null.

(Refer Slide Time: 24:02)

```
        scanf("%d", &n);
        temp = (list) malloc(sizeof(node));
        temp->data=n;
        temp->next=head;
        head=temp;
        printf("Enter More Data?(y/n)\n");
        scanf("\n%c", &ch);
        fflush(stdin);
        }
    temp = head;
    while(temp!=NULL){
        printf("%d ", temp->data);
        temp=temp->next;
        }
:q
#
```

So this is just like changing the array index i plus plus, this is temp pointer dot next. So this is traversing and using this program if we run it, we are going to read the list and print it in the reverse order. So, we read in 5 7 51 and 4 and if we print it, this is how the list is stored actually 4 51 7 and 5.

(Refer Slide Time: 24:28)

```
Enter Data? (y/n)
y
 Give data: 5
Enter More Data?(y/n)
y
 Give data: 7
Enter More Data?(y/n)
y
 Give data: 51
Enter More Data?(y/n)
y
 Give data: 4
Enter More Data?(y/n)
n
4 51 7 5
#
```

On the other hand if we want to insert it in the beginning then the program will look like this, just I will make a few. This is the same struct node type int data node type pointer to next node type def list.

(Refer Slide Time: 25:41)

```
#include<stdio.h>
typedef struct node_type {
            int data;
            struct node_type *next;
            } node;
typedef node *list;
void show_list(list);
list reverse(list);
list rec_rev(list);
main()
{
    list head, tail;
    char ch;
    int n;
    head = tail= NULL;
```

We have some functions, the first one is show list, the second and third are reversing of list which we will discuss subsequently and here we require as I mentioned before, the head and the tail because you are inserting at the end, so you will make tail pointer dot next equal to temp. That is what you are going to do because tail points to the last element in the list, character in temp. So, you initialize head and tail to null because there is no element in the list, printf enter data yes. If so the first element is read in an if loop, if statement.

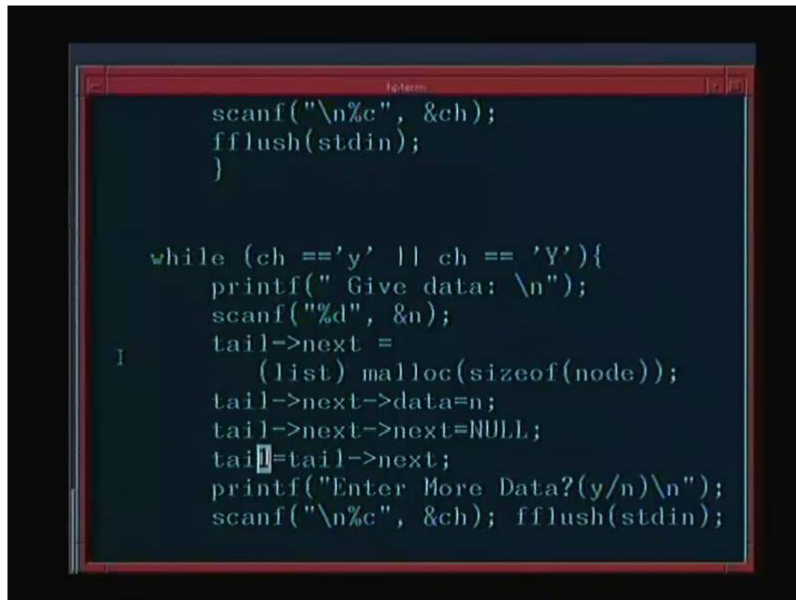
(Refer Slide Time: 26:36)

```
    int n;
    head = tail= NULL;
    printf("Enter Data? (y/n)\n");
    scanf("%c", &ch); fflush(stdin);

    if (ch =='y' || ch == 'Y'){
        printf(" Give data: \n");
        scanf("%d", &n);
        tail = (list) malloc(sizeof(node));
        tail->data=n;
        tail->next=NULL;
        head=tail;
        printf("Enter More Data?(y/n):");
        scanf("\n%c", &ch);
        fflush(stdin);
```

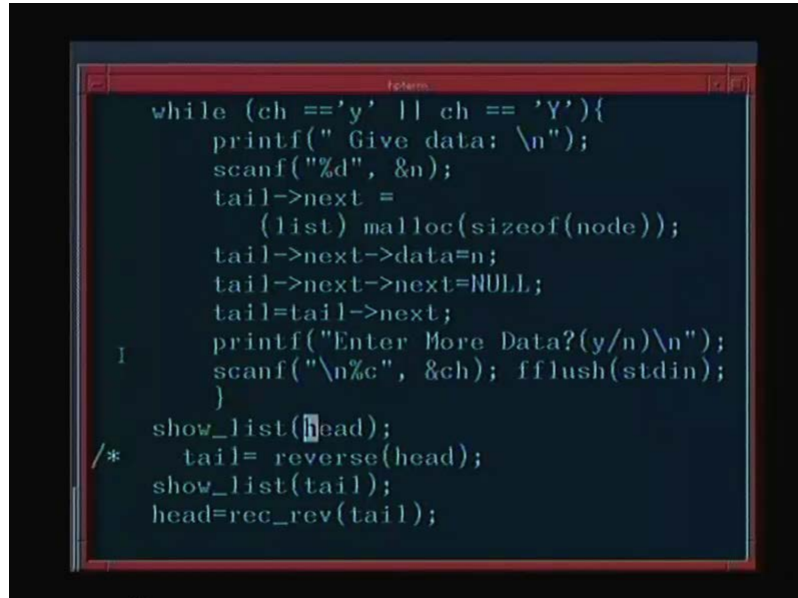
If this is true scanf, tail is equal to we just directly allocate to tail, tail is equal to list malloc tail dot data is equal to n, tail dot next is equal to head. Sorry tail dot next should be equal to null because that's what supposed to be head. And tail are the same thing and if you want to read more data. So this inserts the first element, it initializes, it allocates data to tail, makes tail dot data equal to this, null and head and tail are same and it wants to read more data. If you give more data then in the loop this is what it will be. Tail dot next see look at this, we want to insert a new element at the end of tail.

(Refer Slide Time: 27:40)



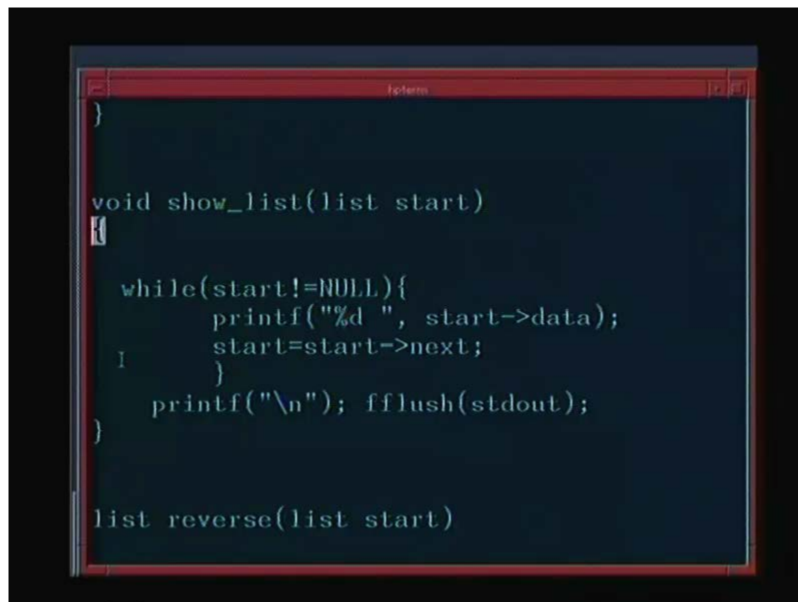
Tail dot next should be null normally, so tail dot next will get the new allocated data. So in tail dot next which is the pointer which is of type list, we will malloc the new element and point it out alright. And tail dot next dot data will be n and tail dot next dot next will be null and tail will move one step ahead to tail pointer dot next, alright. Head does not change because head is initialized to the start and head remains just as it was. And if you want to read another data you can continue to read another data and a function and just call show list with head.

(Refer Slide Time: 27:53)



So let us see how a function show list will be written. This is a function show list which was written inside the main program in the previous function.

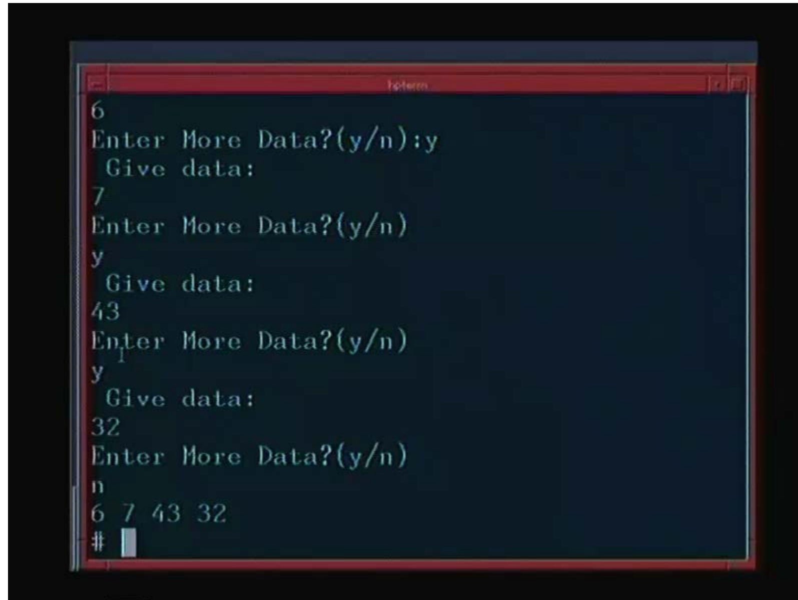
(Refer Slide Time: 28:00)



Show list will take a pointer to a node as argument which is of type list, so it will take a pointer to node as argument and this variable is called start here and while start is not equal to null, it will print start dot data and start will be start dot next and it will continue till that happen and print back slash n. So this is the program to insert in the beginning, insert at the end so that you get the list in the order in which you have entered the

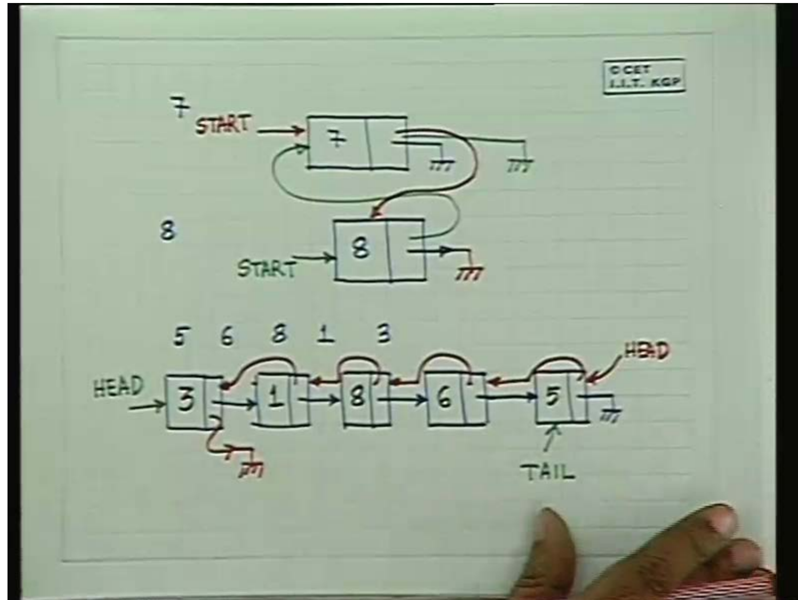
elements. So you have given 6 7 43 and 32 you will get it in that order. So we have seen exactly how we can form lists.

(Refer Slide Time: 29:08)



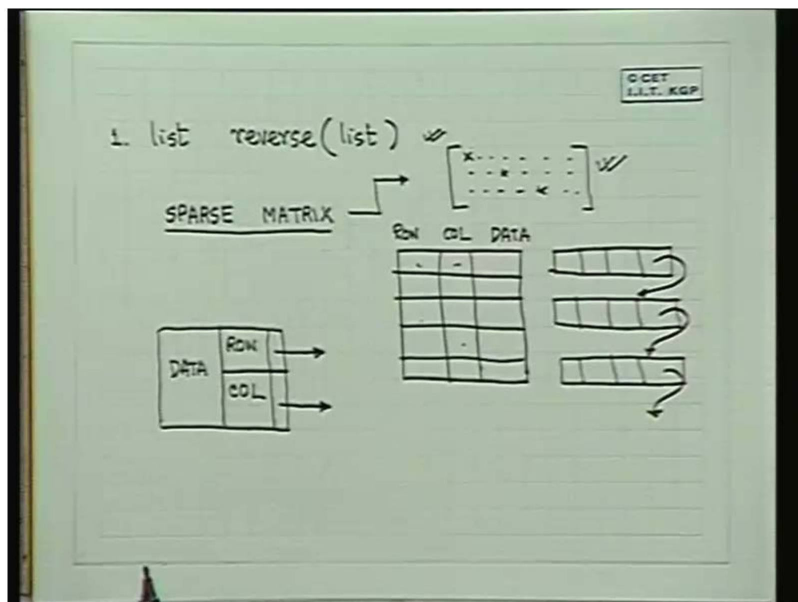
So we have been able to solve our basic problem which was of reading in data continuously and forming up what is a link list. So now we are capable of reading in data as and when required but this linked list structure is quite interesting because its recursive that is a node points to itself. Look at this one, you could not have declared struct node type if it was not a pointer then you would be in trouble because a node cannot have a node inside it. Then the size of that node would become infinite but this is just a pointer to a node. Therefore here self referentiality is through pointers that is possible but direct self-referencing is not allowed. So using our c structures of dynamic allocation and data structure which point, which is capable of pointing to itself, we are now able to generate a simple linear link structure.

(Refer Slide Time: 31:31)



Next we shall come to various manipulation algorithms for this linked structure. That is given a link list like this, could you reverse the link list? That is if I give you this link list as input could you make it such that the head now points here, head points here and all the elements are reversed, all the links that is this one is now here. This one is now here, this one is now here, this one is now here and this one is grounded that is given a list like this I want to reverse a list which means that the head will now be what the tail was and all the pointers will point back to the previous nodes.

(Refer Slide Time: 35:52)



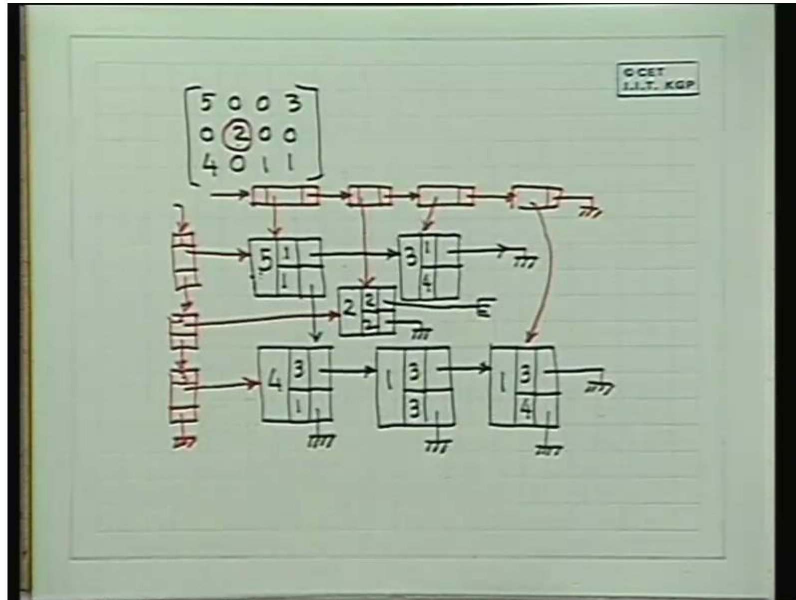
So I want a routine which is like this, reverse it will take in a list type and it will return a list type. It will return the head of the new reversed list. So the problem that, the first problem that we have in hand is reversing, reversing a list. So we will, I will leave it to you to think of how you will write programs to reverse a list, second is to have more complex structures. Suppose I have a sparse matrix and in this sparse matrix may be only 10% of the elements will usually be non-zero and I would like to store up these elements and I will only read in the given number of elements.

So how would I create a data structure for a sparse matrix? So let us see the alternative. The first alternative for declaring a sparse matrix is declare an array of full size and in that array I will put in the elements. The non zero elements I will fill up but that's not what we want to do because if it is a 5000 by 5000 array and I have only 2 or 3 elements then how will I store it. So the other option is to store only the elements. So how will I store only the elements? One method which all of us may know easily is we store up the elements in the following conceptual structure row number, column number and data. We store the row number, the column number and the data.

And if you know how many you are going to store then obviously you can malloc an array and store it and if you don't then obviously other than row, column, data you will have a link field which will point to the next element. You can always create a link list like this but while this is a nice way of doing it for many purposes like may be matrix addition and other things. Suppose you have a problem where you have to traverse the matrix row wise as well as column wise then this representation would have been so nice.

You can traverse it row wise and you can traverse it column wise but this has got too much of space. So how would you put it in this structure? Here you can sort it row wise or you can sort it column wise but if you sort it row wise then the next column element you will not get. Isn't it? The next column element you will have difficulty in obtaining. So, can you suggest what you will do to be able to move row wise as well as column wise? Yes, you can say there is no problem. There is a data, here is the data and you can store row number, column number is okay. So you will store row number, column number anything else. Two pointers can be stored, one to the next element in the row and another to the next element in the column then you will have a structure of an array which looks like this.

(Refer Slide Time: 41:38)



Suppose it is 5 0 0 3 0 4 2 0 0 1 0 1, suppose we take this matrix then this will be, the next element in the row say if this is 5 the row number is 1 1, row and column is 1 1. Then the next element will be 3 1 4, this will be ground then this let us see this will be 4, this will be 3, this will be 1, this will be pointing to this, this will be 1, this will be 3, this will be 3, this will be ground. This will be ground, this will be pointing to 1 3 4, this is grounded, this is grounded fine. But what about this element, what about this element? Poor fellow is left out.

Why is it left out because there is no element two here and there is no element here. In order to solve this problem, we will have header pointers because you will also need require information whether element 2 2 is present, element 2 0 is present. How will you find out? Element 2 0, how will you go to the 2 zero'th element. And we will have header pointers for the columns, for the rows and the columns, alright. So this is the row and this is the column.

This will point here, this will be pointing to that second element 2 2 2. Now this can be grounded and this can also be grounded, there is no difficulty. This will point here, this will point here, this will point here, this will point here. So now let us see if you want to access row wise, now another thing you need not have is you need not have these two fields anymore because these fields are available from this row but let us have it for the time being. And if you want to access row wise, you go this way. Any column you want to access, you access this way. So that is why the row number and the column number have to be stored in these elements somewhere. The row number and the column number 1 2 3 this way.

So for this structure you have got now a linked structure in which you have only elements, you can access it just like you access a matrix. And if you wanted to do a

matrix multiplication this row into this column, it becomes very easy and if you want to add, somebody wants to dynamically add a new element to matrix also you can add. If you want to find out any rows is empty or not this whole thing, if there is no element in a row in a column then this will be ground. And for this you need to define two types of structures. one is this structure, this structure will contain a one data field, three data fields one data, one row number, one column number and two pointers pointing to a structure like this to the same node type and we will have some header pointers which will be of a different structure. They will contain row or column number, a pointer to different structure this structure and a pointer to its own type of structure.

So this will give you a representation for a matrix which is absolutely dynamic where you can store you can add one element of a matrix, you can delete one element of a matrix and the size will be just as it is given. So I ask you to plan out how you will solve a matrix multiplication, matrix addition problem let us say using such a definition and secondly to write out in a simple linked list to write out how you will, given a list how you will reverse its elements. So just try out these two problems as an assignment. We will start with more definitions on linked list and linked structures from the next class.