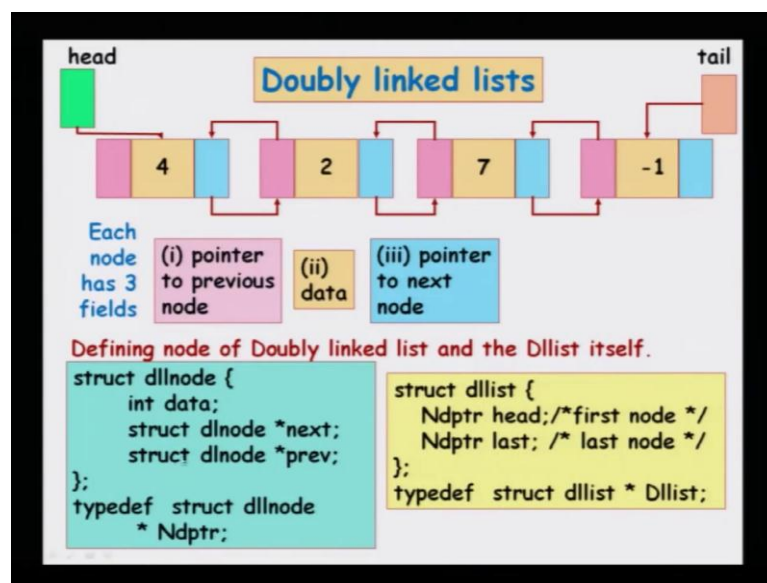


Introduction to Programming in C
Prof. Satyadev Nandakumar
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture – 52

In this lecture, we will see a slightly more advanced data type than a singly linked list. We will briefly go over one or two functions to manipulate the data structure. The principle of manipulating the data structure for the other operations is similar.

(Refer Slide Time: 00:18)



So, in the case of a singly linked list we have seen that every node has one link to its next neighbor. And we have seen this problem in a singly linked list that, if you are at a current node in a linked list, you can always go forward but, there is no way to go back. The only way to get to its previous node is to start all over again from the beginning of the list and traverse until you reach the previous node.

So, we can easily remedy this by thinking of a data structure, a slightly more involved data structure, where every node has two links. So, look at this node 2, it has two links, one is to its successive neighbor, so, it is next node. There is another link which goes back to its previous neighbor. So, in this data structure there are two links per node. Therefore, it is known as a doubly linked list and this list obviously, you can go from a current node, you can go forward or backward so, easily.

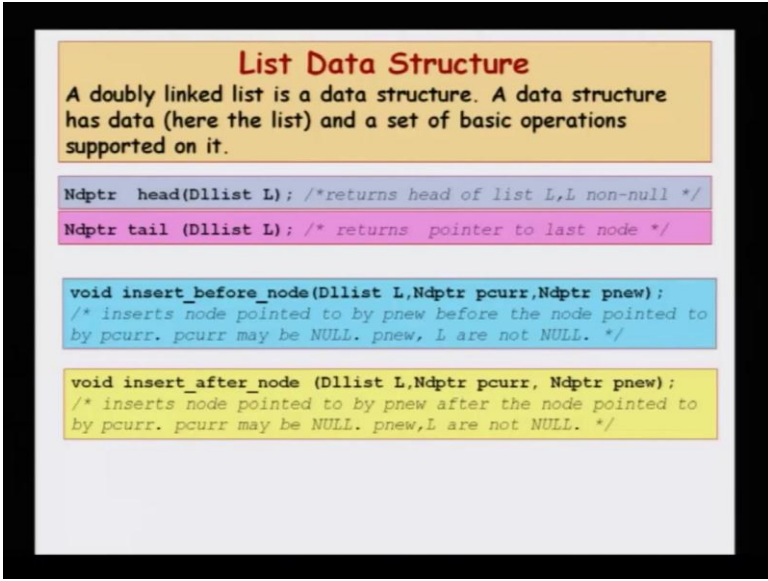
So, now the variation is this and each node has three fields, one is a pointer to the

previous node, the second is the data in the node and the third is the pointer to the next node. So, how will the definition look like? It will say something like struct dllnode, Doubly Linked List node. It will have one field which is data, int data let us say and then two nodes, struct dllnode next and struct dllnode previous.

So, one to go to the next node and another to go to the previous node. Now, we will need two pointers typically for a doubly linked list. One is, the pointer to the beginning of the list, which is usually called the head and then, another to the end of the list which is usually called the tail. So, I will use a typedef in order to shorten the name, I will just say typedef struct dllnode star, node pointer. And then, I will say that the list has two node pointers, node pointer head and node pointer last.

So, a doubly linked list, each node in the doubly linked list has two lists, one to it is previous node and another to it is next node. And the list itself has two pointers, one to the beginning of the list called the head and another to the end of the list called the tail.

(Refer Slide Time: 03:00)



List Data Structure

A doubly linked list is a data structure. A data structure has data (here the list) and a set of basic operations supported on it.

```
Ndptr head(Dllist L); /*returns head of list L,L non-null */
Ndptr tail (Dllist L); /* returns pointer to last node */
```

```
void insert_before_node(Dllist L,Ndptr pcurr,Ndptr pnew);
/* inserts node pointed to by pnew before the node pointed to
by pcurr. pcurr may be NULL. pnew, L are not NULL. */
```

```
void insert_after_node (Dllist L,Ndptr pcurr, Ndptr pnew);
/* inserts node pointed to by pnew after the node pointed to
by pcurr. pcurr may be NULL. pnew,L are not NULL. */
```

So, now a doubly linked list is another data structure. Notice that, we have seen two or three data structures so, for arrays are one, which c already provides, we have already seen a singly linked list. Now, you have seen a third data structure which is a doubly linked list. Now, a data structure has data and a bunch of operations, defined on it. So, let us look at typical operations that can be defined on a doubly linked list. And we will go over the implementation of two or three of them.

So, node pointer head. So, this is our function that should return the head of the list.

Similarly, node point of tail, we should return the tail of the list. Insert before, so, this is like the insert before node in the case of a singly linked list. So, here we are given a current node and we have to insert before a current node in the doubly linked list. Notice this was difficult in a singly linked list, because there was no way to go from a current node to a previous node.

We could always go to the next node. So, if I say that here is a node and insert before that node in a singly linked list, it is a difficult, you need some extra information. But, in a doubly linked list you have the current node and you can use the previous link, in order to go before that. Insert after node also can be done, this could also be done in a singly linked list.

(Refer Slide Time: 04:40)



```
List data structure (contd.)  
  
Dllist make_list(Ndptr pnew); /* create a new list */  
Dllist make_empty_list(); /* creates a new empty list */  
int IsEmpty(Dllist); /* is the list empty? returns 1 if it  
is, otherwise returns 0. */  
  
Dllist copy_list(Dllist L); /* creates a copy of list L */  
Dllist simple_concat(Dllist L1, Dllist L2) ;  
/*concatenate: attach L2 at end of L1. */  
Dllist deep_concat(...);  
/* attach L2 at end of L1. New list is created */  
Dllist append(Dllist L, Dlnode ptr pnew); /*insert at end*/
```

So, and then you can think of several other common like, you can think of a make node, you can think of a make list with a single node pointer 2 by pnew. You can make an empty list, you can check whether a given list is empty. You can write functions to copy a doubly linked list to a new doubly linked list, you can concatenate two doubly linked list, you can do a deep concat, we will see this in a future slide. You can append two link list and so, on.

(Refer Slide Time: 05:21)

The List data structure (contd.)

```
void delete_node(Dllist pL, Ndptr p); /* deletes the node
pointed to by p. p must be a member of list pointed to by
pL. It also frees the storage of node p*/
```

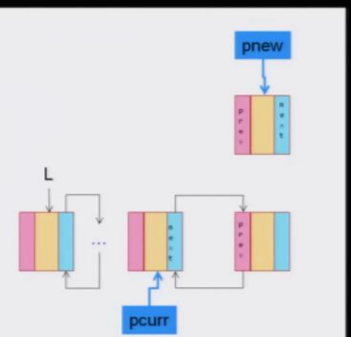
```
void extract_node( Dllist L, Ndptr p); /* removes the node
pointed to by p and fixes the list accordingly. p must be a
member of list L. It does not free the storage of the node
p */
```

```
void delete_list( Dllist L); /* releases storage for all
nodes in the list L and releases storage for the structure
pointed to by L. This completely deletes the list*/
```

Similarly, since we have insert functions, we can also have delete functions, you can delete a particular node, you can extract a node in the sense that... So, delete would take out a node and free the memory allocated to the node. Extract would just take out the node from the linked list. But, you retain the node, you can delete an entire list and so, on.

(Refer Slide Time: 05:50)

```
Dllist insert_before_node
(Dllist L, Ndptr pcurr, Ndptr
pnew){
if (!L)
return make_list(pnew);
if (L->head==NULL){
L->head = L->tail = pnew;
return L;
}
if (!pcurr)
return L; /*error*/
pnew->next = pcurr;
pnew->prev = pcurr->prev;
if (pcurr->prev )
pcurr->prev->next = pnew;
else
L->head = pnew;
return L;
}
```



So, let us look at a couple of these functions, other functions can be written in similar manner. So, suppose let us take insert before node, this was a function that was not easy with the singly linked list. So, I am given a linked list L and given a current node pcurr and a new node to insert before the current node. So, what are the things to check? If the

list is empty, then insert before the current node just means that, you create a new node and return the new list.

Now, if the head of the list is null, then you just say that now the new list contains only one node, L head will point to new, L tail will point to new. So, if the list itself was null, then what you do is, you create a new node. Now, the new list contains only one element. So, the head will point to that and the tail will also point to that and you return that. Now, you come to the non-trivial case. Suppose, there is a list and the list has some elements.

So, if p current equal to null, then you return the L, this is an error, if p current is not equal to null, then what you do is the following. So, now you have to insert p new into the list. So, how do you do this? So, we say that the new nodes next will be... So, we are trying to insert pnew before p current. So, the new nodes next will be p current, p currents previous will go to pnew.

And so, the pnew next will go to p current and p currents previous, will go to pnew. Similarly, we have to say that the previous node, the node before p current it has to point to pnew. So, p currents previous, that nodes next will go point to pnew and then you return the new list. So, this can be done by looking at pointers and handling pointers, carefully.

(Refer Slide Time: 08:23)

```
void delete_list_hdr(Dllist L)
{ if (L) free(L); }

void delete_node(Dllist L, Ndptr p)
{
    if (IsEmpty(L) || !p)
        return;
    if (L->head == p) {
        L->head = p->next;
        L->head->prev = NULL;
    }
    if (L->tail == p) {
        L->tail = p->prev;
        L->tail->next = NULL;
    }
    if (p->next)
        p->next->prev = p->prev;
    if (p->prev)
        p->prev->next = p->next;
    free (p);
}

void extract_node( Dllist L, Ndptr p){
/* same code as above, except for free(p) */
}
```

The slide also contains three diagrams illustrating the deletion process for different cases:

- case 1:** A diagram showing a doubly linked list with a 'head' pointer pointing to the first node. The first node is highlighted in pink and yellow. An arrow labeled 'head' points to this node.
- case 2:** A diagram showing a doubly linked list with a 'tail' pointer pointing to the last node. The last node is highlighted in pink and yellow. An arrow labeled 'tail' points to this node.
- case 3:** A diagram showing a doubly linked list with a middle node highlighted in pink and yellow. Arrows from the nodes immediately before and after it point to this node, indicating the update of pointers.

So, now let us see how to delete a particular node in a list. So, if you have to delete the header of the list, then if there is a list you just delete the header and you just free the entire list. Now, if you have to delete a particular node in the middle of a list, what do

you do? So, let us look at the various cases. So, in case 1 the node that you want to delete is the head of the list. So, in this case suppose you want to delete p, what would you do?

You would make head point to the next element and free p. So, head will be made to point to p next. So, this is the line here, L head will go to p next. Now, this guy's previous will be set to null, because we are going to delete this node. So, this guy's previous will set to null. So, now it does not point to anything and then you will free p. So, this is the first case, where p the node to be deleted was the head of the list.

Similarly, if you want to delete the tail of the list. So, now, what should you do here? The tail should go to p previous. So, in case 2 when p is the end of the list that we want to delete, then tail should go on to p previous. Now, this guy next will now point to null. Because, we are going to delete this node and finally, we will free p. So, L tail will go to p previous, L tail next will be a null and then finally, you will free p.

So, we have seen two easy cases, one is delete the head and the other is delete the tail. And now, you will see the difficult case, where p is an intermediate node. So, in this case what we will do? So, we have to remove this node. So, p previous next node should be the next node of p previous. So, this link should point to the node after p. So, that is the first thing. So, we will make this node point to the node after p and this node previous should point to the node before p. So, we will reset the links.

Now, if you look at the link, this guy's next is the one after p, this guy's previous is the one before p. So, now p can be safely ago. So, this is how you would delete a node in the intermediate list. So, if there is a next node, then p next previous will be p previous that is this backward link. And if there is a previous node, then p previous next will be p next, that is this forward node and finally, after that you will free p.

So, this is how you would delete a node from a doubly linked list. And other operations can be done in a similar manner and some of these operations will be asked in the exercise problem that you will be assigned. Similarly, you can think of an extract node. The code will be exactly identical to before, except at the end, you will instead of freeing p you will return p, you do not free the p node, you will just return the p node.

(Refer Slide Time: 12:07)

```
Dllist append( Dllist L, Ndptr p){
    if (!p)
        return;
    if (!L)
        return make_list(p);
    return insert_after_node(L, L->last,p);
}
```

Now, let us look at one more example, how do you append one node to the end of a list? So, first we will check that the node is pointing to a non null node. If it is pointing to a null node that is nothing to be done. So, there is nothing to be appended,. So, you have returned. Now, if there no need list, what you do is you make a list with only one node which is p. Now, if there is a list you can, in order to append the node at the end what you can do is, call insert after node L, L last p. So, append will be the same as insert the node p at the end of the list. So, you will say insert after L last, what is the node to be inserted, p. So, if you have an insert after node or an insert before node, you can do this to implement other functions. So, this is a brief introduction to doubly linked list which are similar to singly linked list, but facilitate forward as well as backward traveling from a current node. Using that you can implement more functions easier than a singly linked list. At the same time, it has all the advantages of a singly linked list in the sense that, if you want to insert a node, it can be done using a constant number of operations. If you want to delete a node, it can be done in a constant number of operations. So, those advantages are similar to a singly linked list.

At the same time, the disadvantages are also similar to a singly linked list in the sense that, if you want to search through even a sorted doubly linked list, you have to search through all the elements.