**Doubly-Linked List**

1. ***Doubly Linked List* example:**

Here, we will discuss *deleting an element with a specific value* from a doubly-linked list,

and compare it to deleting from a singly-linked list.

Here are some constraints for our example:

* 1. The list holds char's.
  2. Deletion can occur *anywhere* in the list.
  3. The delete operation will not return the element that we delete, it only need remove it from the list.

Now, recall the basic structure of a doubly-linked list:

head

|

v

------------- ------------- -------------

| | | | <---+-- | | | <---+-- | | |

| 0 | a | --+---> | | b | --+---> | | c | 0 |

------------- ------------- -------------

In other words, there is a node for each element, where nodes consist of a part in which the element is stored, a link to the next node, and a link to the previous node. The last node links to nothing i.e., there are no nodes after it. Also, there are no nodes before the first node. There is a link to the beginning of the list called ***head***.

*What do these****links****correspond to in C++?*

**Answer:** Pointers!

We'll consider our elements and nodes to have the following types:

typedef char ItemType;

class DNode {

public:

...

ItemType getData();

DNode \*getNext();

void setNext(DNode \*);

DNode \*getPrev();

void setPrev(DNode \*);

private:

ItemType data;

DNode \*next; // pointer to node after me

DNode \*prev; // pointer to node before me

};

And, the actual linked list will be a type called DList:

class DList {

public:

...

bool search(ItemType);

private:

DNode \*head; // pointer to beginning (1st node) of linked list

};

1. **Deleting from a list:**

Recall that for a singly-linked list there were a few steps to deleting a specific element from the list:

* 1. Find the node with the element (if it exists).
  2. Remove that node.
  3. Reconnect the linked list.
  4. Update the link to the beginning (if necessary).

**Note:** For simplicity, if there are two elements with the value we want, we'll just remove the first one.

Finding the node in question is a matter of traversing the list and looking at each node's element.

Reconnecting the list once a node is to be removed is more interesting. Let's consider at least 3 cases:

* Removing a node from the *beginning*.
* Removing a node from the *middle*.
* Removing a node from the *end*.

**Removing from the beginning**

When removing the node at the beginning of the list, there is no relinking of nodes to be performed, since the first node has no preceding node. For example, removing node with **a**:

head

|

v

------------- ------------- -------------

| | | | <---+-- | | | <---+-- | | |

| 0 | a | --+---> | | b | --+---> | | c | 0 |

------------- ------------- -------------

However, we must fix the pointer to the beginning of the list, and set the previous pointer of the new node at the beginning of the list to 0 (NULL):

head

|

+----------------+

|

v

------------- ------------- -------------

| | | | | | | | <---+-- | | |

| 0 | a | --+---> | 0 | b | --+---> | | c | 0 |

------------- ------------- -------------

**Removing from the middle**

Removing a node from the middle requires that the preceding node *skips over* the node being removed, and the following node to skip over in the reverse direction. For example, removing the node with **b**:

head

| +---------------------+

v v |

------------- ------------- | -------------

| | | | <---+-- | | | +--+-- | | |

| 0 | a | --+--+ | | b | --+---> | | c | 0 |

------------- | ------------- -------------

| ^

+---------------------+

Note that in contarst to the singly-linked list case, here it is straight forward to refer to *the node****before****the one we want to remove*byusing its previous pointer.

**Removing from the end**

Removing a node from the end requires that the preceding node becomes the new end of the list (i.e., points to nothing after it). For example, removing the node with **c**:

link

|

v

------------- ------------- -------------

| | | | <---+-- | | | <---+-- | | |

| 0 | a | --+---> | | b | 0 | | | c | 0 |

------------- ------------- -------------

Note however that in contrast to the singly-linked list case, here the last two cases (middle and end) cannot be combined because there is no node after the last node.

**IMPORTANT NOTE:** When there is only one node, and it is the one to be removed, are we in the case of removing at the **beginning** or **end**?! We should make sure that one of these cases handles removing this single node ELSE we should create a new case.

**Deletion member functions**

We'll write a List member function to delete an element from a list *iteratively*:

class DList {

public:

...

bool deleteIter(ItemType);

private:

DNode \*head;

};

The deletion function will return a bool representing whether the element was deleted (i.e., was present in the list).

1. **Iterative implementation:**

Let's look at how to delete iteratively (i.e., with a loop) via a member function:

bool deleteIter(ItemType);

We'll need to be able to change the pointer to the list, link, in the case that the first node in the list is removed. Since deleteIter() is a member function of List, and link is one of the data member of that class, we can access it in deleteIter() when necessary.

We will iterate (i.e., loop) through the list to find the item to delete. It's easy to see how to start a pointer at the beginning and move it from one node to the next:

------------- ------------- -------------

| | | | <---+-- | | | <---+-- | | |

| 0 | x | --+---> | | y | --+---> | | z | --+--->

------------- ------------- -------------

^

|

currP

In other words:

* + Start at the beginning:
  + currP = head
  + Advance to the next node when necessary:
  + currP = currP->getNext()

by calling the member function to get the next node.

* + Stop when there are no more nodes, i.e., make sure that:
  + currP != NULL

It's easy to combine these into a for loop:

for (currP = link; currP != NULL; currP = currP->getNext()) {

**...**

When we find the one to remove, we'll also need a pointer to the previous node, but this is trrivial; we simply use the previous pointer, so no extra pointer is neede to traverse the list !!!

To complete the function, we'll have to:

* Have some way to indicate when there is no *previous* node.  
  Remember that removing the first node is a special case because...
  + It requires no relinking of nodes.
  + We have to update the pointer to the beginning of the list.
* Have some way to indicate when there is no *next* node.  
  Remember that removing the last node is a special case because...
  + It requires no relinking of nodes.
  + We have to update the pointer to the end of the list.
* Relink the list (i.e., skip over the node to be deleted).
* Remove the node.

One implementation of this function might be:

bool List::deleteIter(ItemType item)

{

DNode \*currP;

/\*

\* Visit each node

\*/

for (currP = head;

currP != NULL;

currP = currP->getNext()) {

if (currP->getData() == item) { /\* Found it. \*/

if (currP->getPrev() == NULL) { /\* Remove from beginning \*/

/\* Fix beginning pointer. \*/

head = currP->getNext();

} else if(currP->getNext() == NULL) { /\* Remove from end \*/

currP->getPrev()->setNext(NULL);

} else { /\* Remove from middle \*/

/\*

\* Fix previous node's next to

\* skip over the removed node.

\*/

currP->getPrev()->setNext(currP->getNext());

/\*

\* Fix next node's prev to

\* skip over the removed node.

\*/

currP->getNext()->setPrev(currP->getPrev());

}

delete currP; /\* Deallocate the node. \*/

return true; /\* Done searching. \*/

}

}

return false; /\* Not in the list. \*/

}

This function can be called as:

dlist.deleteIter(item);

Finally, note that by using a ***sentinel object***- a dummy node at the beginning of the doubly-linked list, the delete operation can be greatly simplified; more specifically, all three cases (removing from the beginning, middle and end) can be combined. The insert operation can be simplified in the same manner. All this comes at the expense of memory: additional memory is needed for the sentinel object, and more significantly, twice the memory is needed for a doubly-linked list compared to a singly-linked list, because the additional previous pointer.

*BU CAS CS - Doubly-Linked List   
This page created by Joni Alon <*[*jalon@cs.bu.edu*](mailto:jalon@cs.bu.edu)*>   
Material adapted for doubly-linked list from*[*Deleting from a Singly-Linked List*](http://www.cs.bu.edu/teaching/cs112/spring-2000/linked-list-delete/)*(C++ version by Jisoook Youn <*[*jisook@cs.bu.edu*](mailto:jisook@cs.bu.edu)*>, and*[*Deleting from a Linked List*](http://www.cs.bu.edu/teaching/c/linked-list/delete/)*(C version by*[*Robert I. Pitts*](http://cs-people.bu.edu/rip/)*<*[*rip@cs.bu.edu*](mailto:rip@cs.bu.edu)*>).*