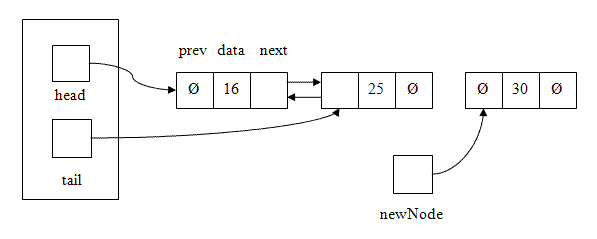
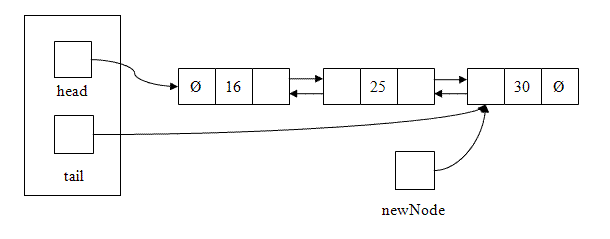
**Insertion at the Rear of a Doubly-Linked List**

**Case 1: List is not empty**

Assume that our linked list contains one or more nodes and that we have allocated a new list node using the pointer newNode. A diagram of the list might look like this:



To insert the new node at the rear of the list, we have to set three pointers: the prev pointer in newNode,\ the next pointer in the current last node in the list, and tail, which needs to be updated to point to the new last node in the list. (The next pointer in newNode has already been set to NULL by the constructor, so we don't need to change that.) Once we've finished, the list should look like this:



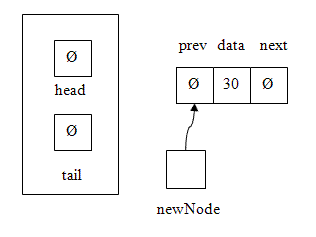
Here is the C++ code to perform these three steps:

1. newNode->prev = tail;
2. tail->next = newNode;
3. tail = newNode;

The order of these steps is important. We can code Steps 1 and 2 in either order with no problems, but Step 3 must be done last. (Why?)

**Case 2: List is empty**

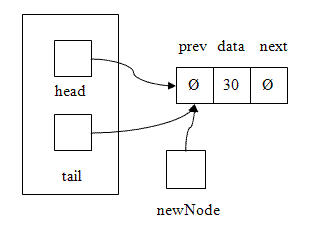
The steps above work as long as there is at least one node in the linked list. But what if the list is empty?



If we use the same steps as above:

1. newNode->prev = tail;
2. tail->next = newNode; // Since tail == NULL, this step causes a segmentation fault
3. tail = newNode;

To insert the new node at the rear of an empty list, we once again have to set three pointers: the prev and pointer in newNode, head, which needs to point to the new first node in the list, and tail, which needs to be updated to point to the new last node in the list. Once we've finished, the list should look like this:



So, for an empty list, the correct C++ code to perform these three steps is:

1. newNode->prev = tail; // Since tail == NULL, newNode->prev will be set to NULL as well
2. head = newNode;
3. tail = newNode;

To combine the two cases and minimize repetition of code, we can

* Perform Step 1
* Decide which version of Step 2 to perform based on whether or not the list is empty
* Perform Step 3

**Insertion at the Front of a Doubly-Linked List**

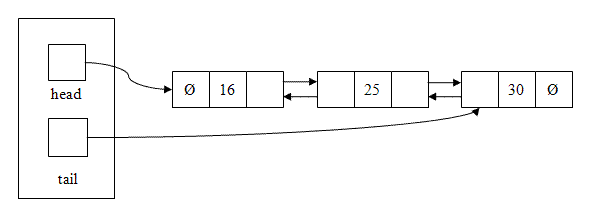
The steps for insertion at the rear of a doubly-linked list and the steps for insertion at the front of a doubly-linked list are *symmetric*. This means that to write the code for push\_front(), take the code you've written forpush\_back() and

1. change every occurrence of head to tail, and vice versa
2. change every occurrence of next to prev, and vice versa

**Deletion at the Rear of a Doubly-Linked List**

**Case 1: List contains more than one node**

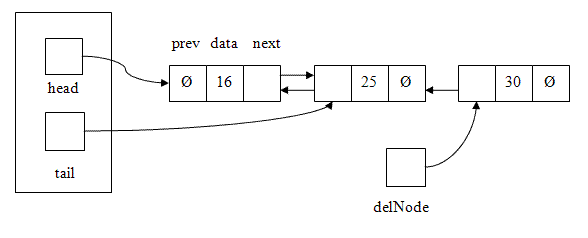
Assume that our linked list contains two or more nodes:



The steps in C++ to remove the last node in the list look like this:

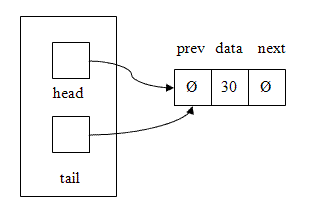
1. LNode\* delNode = tail; // Save address of node to delete in a pointer
2. tail = delNode->prev; // Point tail at the new last node in the list
3. tail->next = NULL; // Set the new last node's next pointer to NULL
4. delete delNode;

Here's a diagram of the list just after Step 3:



**Case 2: List contains one node**

The steps above work as long as there are at least two nodes in the linked list. But what if the list only contains one node?



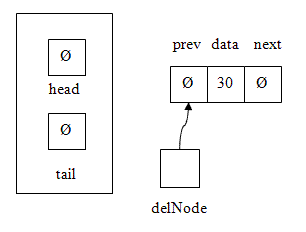
If we use the same steps as above:

1. LNode\* delNode = tail; // Save address of node to delete in a pointer
2. tail = delNode->prev; // This makes tail NULL, which is what it should be
3. tail->next = NULL; // Segmentation fault!
4. delete delNode;

Once again, this is a special case that needs to be handled a bit differently. The correct sequence of steps in this case is:

1. LNode\* delNode = tail; // Save address of node to delete in a pointer
2. tail = delNode->prev; // This makes tail NULL
3. head = NULL; // If tail == NULL, head should be as well since the list is now empty
4. delete delNode;

Here's a diagram of the list just after Step 3:



As with insertion, to combine the two cases and minimize repetition of code, we can

* Perform Steps 1 and 2
* Decide which version of Step 3 to perform based on whether or not the list is now empty (i.e., tail == NULL)
* Perform Step 4

**Deletion at the Front of a Doubly-Linked List**

The steps for deletion at the rear of a doubly-linked list and the steps for deletion at the front of a doubly-linked list are also *symmetric*. This means that to write the code for pop\_front(), take the code you've written forpop\_back() and

1. change every occurrence of head to tail, and vice versa
2. change every occurrence of next to prev, and vice versa