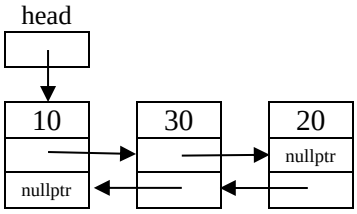
Supplement to Linked List Lecture

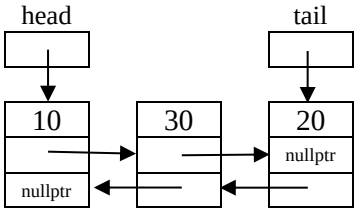
First some notation: When we show a pointer pointing to an object, it's irrelevant where the arrowhead of the pointer touches the object. Thus, these two pictures of a situation where two pointers point to the same struct/class object with three data members mean the same thing:

If the object were at memory address 1000, we assume both of these pictures mean that the two pointers both contain memory address 1000. (Under a notational convention we're *not* using, one might consider the pointers in the right picture to hold different values, neither being memory address 1000.) The reason we use this convention is that when drawing a doubly-linked list, fewer lines cross each other in the picture.

A straightforward way of representing a doubly-linked list of integers containing 10, 30, and 20 in that order would be



This gives us direct access to the first item in the list, and from there we can reach every other item. But for many applications, a lot of activity occurs at the end of the list, so rather than starting at the beginning of a long list and traversing it to the end every time we needed to do something at the end of the list, it would be faster to have direct access to the end of the list:



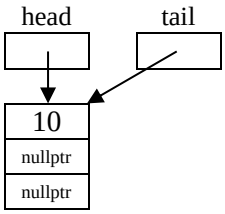
This works, but it makes the implementation of many typical operations a bit tricky. For example, to remove the first item from a list, we might think we could do this:

```
Node* oldHead = head;
head = head->m_next;
head->m_prev = nullptr;
delete oldHead;
```

But this code is wrong. While it works in the general case, it fails when the list is empty (i.e., when head and tail are null pointers). We should have thought of that: How can you remove the first item from a list if there are no items in the list? So let's fix that:

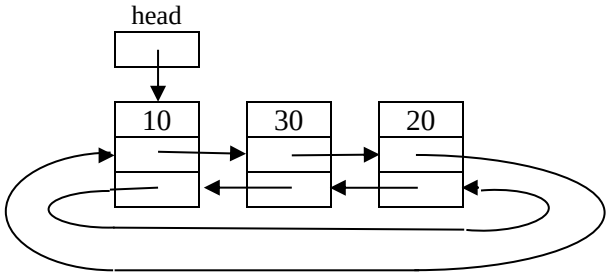
```
if (head != nullptr)
{
      Node* oldHead = head;
      head = head->m_next;
      head->m_prev = nullptr;
      delete oldHead;
}
```

But this code is flawed, too. What if the list has just one element?



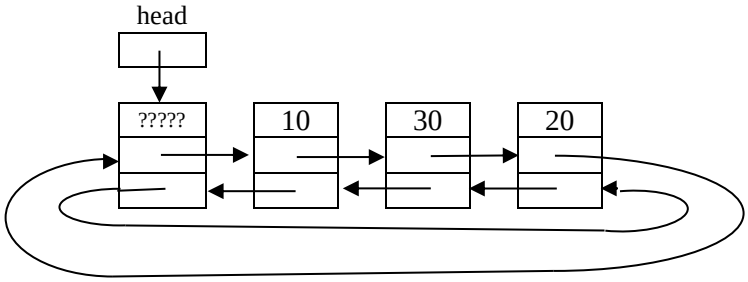
Exercise for the reader: Fix the two bugs.

The representation of a list that we've chosen will lead to a lot of special case checking in code that manipulates the list. One way to reduce some of that checking is to change the representation to one that, for example, will not have the problem of potentially following a null pointer:



This is a *circularly linked list*, or more specifically, a *circular doubly-linked list*. (Note that singly-linked lists can also be circularly-linked.) Notice that we no longer need to keep a separate tail pointer; the tail of the list will be at `head->m_prev`.

While a circular list eliminates some of the special case checking, we'll still have some issues with lists with zero or one element. A technique that might simplify our code is to ensure that there is always at least one node in the linked list:

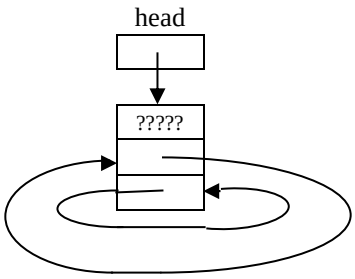


The first node is a *dummy* node, so called because we don't consider its value to be part of the list; in fact we never bother initializing that value, since we will never examine it. The first item we consider to be in the list is at `head->m_next`; the last item we consider to be in the list is at `head->m_prev`. Because there is now *always* a node before and after every item in the list, we've eliminated the special cases in code that manipulates the list.

Here's an example of using a circular doubly-linked list with a dummy node:

```
  // 10 30 20
for (Node* p = head->m_next; p != head; p = p->m_next)
     cout << p->m_data << endl;
  // 20 30 10
for (Node* p = head->m_prev; p != head; p = p->m_prev)
     cout << p->m_data << endl;
```

If the picture above shows what a three-element list looks like with this representation, then what does a zero-element list look like?



If you try writing code to insert and remove items from the list at various positions, you'll find that such code is easiest to write with this representation, because there are no special cases.

Advice

Here are some tips about working with code involving linked lists:

- Draw pictures! In many places in the code, you will be creating or deleting nodes and retargeting where a pointer variable points to. If you look at just the code, it's hard to understand what's going on, so it's easy for mistakes to go unnoticed. Draw a picture of the list and trace through the code carefully, updating the picture as you go along in accordance with what the code actually says to do, not what you wish it did. It's amazing how many bugs you catch this way.

- Anytime you write `p->something`, make sure that you can prove to yourself that no matter how you get to that point,
    - p has previously been given a value
    - p's value is not the null pointer
    - p does not point to an already-deleted object

- Make sure that code that inserts or removes or examines items in a list works correctly
    - for activity in the typical case: somewhere in the middle of the list
    - for activity at the front of the list
    - for activity at the end of the list
    - for an empty list
    - for a one-element list (usually, but not always, this case is covered by some of the others)

- When doing something that changes several pointers (e.g., inserting, removing, or rearranging nodes), make sure you execute the statements changing the pointers in the right order; don't lose a pointer value that you need later. When creating a new node, one thing that helps is setting its pointer members before changing other pointers: Those members have no old value that you might need later, and by setting them first, you reduce the number of ways to mis-order the remaining pointer manipulations.