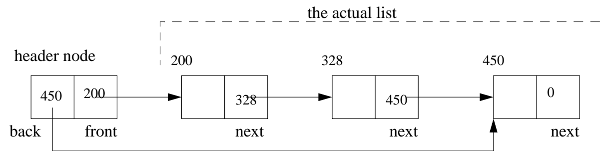
Linked Lists

It is sometimes useful to add a special “header” node to the implementation of a linked list class. The header node is a special node that does not contain an element of the list, but is used to facilitate managing the list. For example, the header node might contain a pointer to the front node and the back node in a singly linked list, which makes it a different type of node than a regular list node:



The header node is a tradeoff of some extra space for faster access time, especially to the last node in the list. Without a header node, inserting a new node at the end of the list would require iterating over the entire list until the last node was found, then inserting at the end. If the list is very long, this can be quite time particularly if you are inserting a large number of items into the end of the list. By wasting space in the header node (8 bytes), you can access the last node in the list in one step. So, both the insert front/back and remove front/back operations are trivial.

Q: How do you tell if you have an empty list? How might you implement a “size()” operation for a list, which reports the number of nodes in the list?

Linked List Implementation

When designing the implementation of a list class, you can define a small set of basic methods that do all the list manipulations,and implement the interface methods in terms of those methods. In addition, a list iterator will provide you with a way to easily, and safely traverse a list.

So, the primary operations that need to be implemented are:

1. insert node
2. remove node
3. iteration

All the other operations can be implemented in terms of these basic list manipulations. Pointer manipulations can be very error prone, so if you can encapsulate the basic list manipulations in a few operations that can be called by other List class methods, then it is easier to implement the List with less chance for error.

For example, the insert_node method might be overloaded so that if one node pointer is given, it inserts into the front and if two node pointers are given, a new node is inserted after the node pointed to by the second arg.

```
void List<T>::insert_front(const T& x)
{
        insert_node (new Node(x));  // special case, meaning insert as header->front
}
void List<T>::insert_back(const T& x)
{
        Node* p = new Node(x); insert_node (new Node(x), header->back); header->back = p;
}
```

Q: How would remove_front and remove_back be implemented?

Linked List Implementation

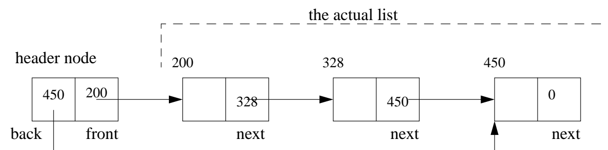
Removing the back node of a singly linked list, even with a header node, requires that you iterate over the list, until you find the node just before the last one just so you can set the next pointer of that node to NULL and update the back pointer in the header node.

```
void List<T>::remove_node(Node* node, Node* prev);

void List<T>::remove_back()
{
        if (header->front == NULL || header->back == NULL)
           return;

        Node* p;
        for (p = header->front; p != NULL && p->next != header->back; p = p->next)
           ;
        remove_node(header->back, p);
        header->back = p;
}
```

The remove_node method removes the node pointed to by the first argument, and sets the next field of the second argument to be the value of the next field of the first argument.

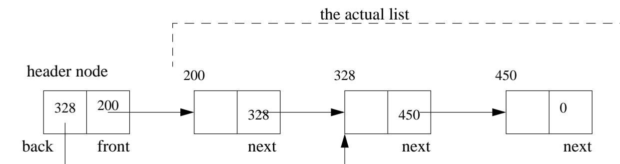


Linked List Implementation

How can the previous problem of removing the back node be solved in a way that does not require iteration through the list?

We could try making the header back pointer really point at the next to the last node instead of the last node. This makes it easy to insert after the last node or remove the last node, as long as we remember that the “back” of the list is really found at header->back->next.

.

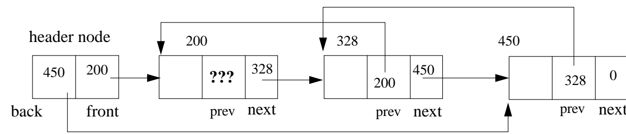



```
void List<T>::remove_back()
{
        if (header->back == NULL)
           return;
        remove_node(header->back->next, header->back);
        header->back = ????;
}
```

Problem! What do we assign to header->back after we remove the last node?

A Doubly Linked List

To avoid having to traverse the list for the remove_back case, we have to tradeoff space for time. We can introduce an additional pointer into each node that points to the previous node in the list, which makes it trivial to access the previous node for any given node.

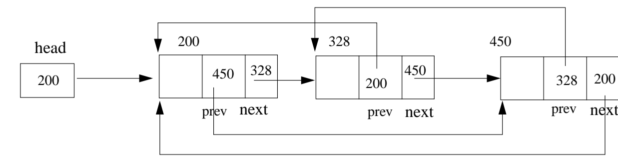


Q: What should the previous pointer of the first node contain?

We could also implement a *circular* doubly linked list and have the first node's previous pointer point at the last node, and the last node's next pointer point at the first node. **If we were to do this, could we eliminate the header node?**

A Circular Doubly Linked List

The first node's previous pointer points at the last node and the last node's next pointer points at the first node.



With this list structure, the remove_back function is trivial:

```
void
List<T>::remove_back()
{
        if (head == NULL)
           return;

        if (head->prev == NULL) {  // special case when only one node in the list
           remove_node(head);
           head == NULL;
        }
        else
           remove_node(head->prev, head->prev->prev);
}
```


Node Deletion

The examples up to this point have assumed that the remove_node operation is a (private) member function of the List class. A more interesting way to implement node insertion and removal is to have the Node constructor insert the new Node into the list after a given node and the Node destructor remove itself from the list:

```
struct Node {

        Node* prev;
        Node* next;
        ...
        Node(Node* p = NULL) : prev(NULL), next(NULL) {
           if (p) {
              prev = p;
              next = p->next;
              p->next = this;
              next->prev = this;
           }
        }

        ~Node() {
           if (prev) {
              prev->next = next;
           if (next)
              next->prev = prev;
        }
};
```


Iteration

The reason for having a separate iterator class is so that you can have multiple instances of an iterator iterating over the list. Each iterator has its own private "current" pointer that points to the current item in the list being iterated. If you implemented list iteration as part of the List class, then it would only be possible to have one iteration in effect at one time since the List object would be responsible for keeping track of the current pointer. Remember that an Iterator<T> is for a List<T> and is a friend of the List.

The implementation of the Iterator depends on how you implement the list: singly linked list with a head pointer, singly linked list with a header node, doubly linked list, etc. The following is a simple iterator implementation for a singly linked list with a head pointer. The end of the list is indicated by a NULL current pointer.

```
template<class T> class Iterator {
        List<T>::Node* first;
        List<T>::Node* current;
public:
        Iterator(List<T>& list) { first = list.head; current = first; }

        begin() { current = first; return first->elem; }
        end() { return NULL; }
        next() { List<T>::Node* p = current; current = current->next; return p; }

        // overload operator() as a way to access the element of a list
        T& operator()() { return *current->elem; }
};

List<int> list;
Iterator<int> i = list;

for (i.begin(); !i.end(); i.next())
        i() = 0;// calls i.operator()(), which returns a ref the current element, which is set to 0
```