Today’s Class: Linked Lists Part 1

- Introductory example on linked lists.

- Basic linked list operations:
  - Stepping through a list
  - Push back
  - Insert
  - Remove

- Common mistakes

- Our study will eventually lead to our own simplified implementation of the standard list class, as well as toward more sophisticated link-based data structures.

**Objects with Pointers / Linking Objects**

- The two fundamental mechanisms of linked lists are
  - creating objects with pointers as one of the member variables, and
  - making these pointers point to other objects of the same type.

- These mechanisms are illustrated in the following short program.

```cpp
#include <iostream>
using namespace std;

template <class T>
class Node {
public:
    T value;
    Node* ptr;
};

void main() {
    Node<int>* ll; // ll is a pointer to a (non-existent) Node.
    ll = new Node<int>; // Create a Node and assign its memory address to ll
    ll->value = 6; // This is the same as (*ll).value = 6;
    ll->ptr = 0; // The value 0 indicates a "null" pointer.

    Node<int>* q = new Node<int>
    q->value = 8;
    q->ptr = 0;

    ll->ptr = q; // ll’s ptr member variable now has the same value
```
// as the pointer variable q

cout << "1st value: " << ll->value << "\n"
   << "2nd value: " << ll->ptr->value << endl;

• We will make the Node class templated throughout our discussion. The functions that
  manipulate the nodes will therefore need to be templated as well.

• The following picture illustrates the structure of memory at the end of the program.
Definition: A Linked List

- The definition is recursive: A linked list is either
  - Empty, or
  - Contains a node storing a value and a pointer to a linked list.

- The first node in the linked list is called the header node and the pointer to this node is called the head pointer. The pointer’s value will be stored in a variable called head.

Visualizing Linked Lists

- The head pointer variable is drawn with its own box. It is an individual variable.
- The objects (nodes) that have been dynamically allocated and stored in the linked lists are shown as boxes, with arrows drawn to represent pointers.
  - Note that this is a conceptual view only. The memory locations could be anywhere, and the actual values of the memory addresses are not usually meaningful.
- The last node MUST have a 0 for its pointer value — you will have all sorts of trouble if you do not ensure this!
- You should make a habit of drawing pictures of linked lists to figure out how to do the operations.

Basic Mechanisms: Stepping Through the List

- Write a function:

```cpp
template <class T>
bool is_there( Node<T>* head, const T& x );
```

  to determine if a particular value, stored in x, is also in the list.

- We can access the entire contents of the list, one step at a time, by starting just from the head pointer.
  - We will need a separate, local pointer variable to point to nodes in the list as we access them.
    - This pointer will play the role of the iterator, although we will not yet have all of the iterator operators.
  - We will need a loop to step through the linked list (using the pointer variable) and a check on each value.

- Writing a templated function is essentially the same as writing a function where the T is replaced by an int — except that it is not.
Basic Mechanisms: Pushing on the Back

- Goal: place a new node at the end of the list.
- We must step to the end of the linked list, remembering the pointer to the last node.
  - This is an $O(n)$ operation and is a major drawback to the ordinary linked-list data structure we are discussing now. We will correct this drawback by creating a slightly more complicated linking structure in our next lecture.
- We must create a new node and attach it to the end.
- We must remember to update the head pointer variable’s value if the linked list is initially empty.
  - Hence, in writing the function, we must pass the pointer variable by reference.
- The function prototype is

\[
\text{template <class T>}
\text{void push_back( Node<T>* & head, T const& value )}
\]

Exercise

Write push_back.

Basic Mechanisms: Inserting a Node

- There are two parts to this: finding the location where the insert must take place, and doing the insert operation.
• We will ignore the find for now. We will also write only a code segment to understand
the mechanism rather than writing a complete function.

• The insert operation itself requires that we have a pointer to the location before the
insert location — in other words, the new value is placed after the value the pointer
refers to.
  
  – Note that this differs from the use of `std::list<T>::insert`, where the value
to be inserted is placed before the node referred to by the iterator.

• If `p` is a pointer to this node, and `x` holds the value to be inserted, then the following
code will do the insertion:

```cpp
Node<T> * q = new Node<T>; // create a new node
q -> value = x; // store x in this node
q -> next = p -> next; // make its successor be the current
                        // successor of p
p -> next = q; // make p’s successor be this new node
```

• Can you draw a picture to illustrate what is happening here?

• This code will not work if you want to insert the value stored in `x` in a new node at
the front of the linked list.

Basic Mechanisms: Removing a Node

• There are two parts to this: finding the node to be removed and doing the remove
operation.

• The remove operation itself requires a pointer to the node before the node to be
removed.

• Removing the first node is an important special case.

Exercise

Suppose `p` points to node that should be removed from a linked list, `q` points to the node
before `p`, and `head` points to the first node in the linked list. Write code to remove `p`, making
sure that if `p` points to the first node that `head` points to what was the second node and
now is the first after `p` is removed.
Basic Mechanisms: Common Mistakes

Here is summary of common mistakes. Read these carefully, and read them again when you have problem that you need to solve.

- Allocating a new node to step through the linked list; only a pointer variable is needed.
- Confusing the \_ and the \=> operators.
- Not setting the pointer from the last node to 0 (null).
- Not considering special cases of inserting / removing at the beginning or the end of the linked list.
- Applying the delete operator to a node (calling the operator on a pointer to the node) before it is removed. Delete should be done after all pointer manipulations are completed.
- Pointer manipulations that are out of order. These can ruin the structure of the linked list.

Looking Ahead to Lecture 11 — Our Own List Class

- Changing the structure of the linked list:
  
  - Nodes will be templated and have two pointers, one going “forward” to the successor in the linked list and one going “backward” to the predecessor in the linked list.
    
    template <class T>
    class Node {
    public:
        Node( ) : next_(0), prev_(0) {}
        Node( const T& v ) : value_(v), next_(0), prev_(0) {}
        T value_;  
        Node<T>* next_;  
        Node<T>* prev_;  
    };

  - We will have a pointer to the beginning and the end of the list.

- All of the mechanisms described above will be reimplemented in a class.

- We will define list iterators as a class inside a class.