CSCI-1200 Data Structures — Fall 2024
Lecture 9 — Linked Lists & List Implementation

Review from Lecture 8

- Unfortunately, erasing items from the front or middle of vectors is inefficient.
- Introduction to iterators: for access, increment, decrement, erase, & insert
- Differences between indices and iterators
- Introduction to STL's `list` class
- *Being a user* of the STL `list` class & `list` iterators (Lab 5, Homework 4, Lab 6)

Today's Class

- Review of differences between STL `list` and STL `vector`
  - Big O Notation comparison of core `vector` & `list` operations
  - Implementation of iterators in our homemade `Vec` class (mimicking STL `vector`)
  - Syntax and functionality of insert & erase on STL `vector` & `list`
  - Situations for iterator invalidation
- *Implementing our own* linked list data structure from scratch:
  - Stepping through a list, searching for an element
  - Push front and push back
  - Insert in the middle
  - Singly-linked vs. Doubly-linked lists!
  - Next Lecture: Finishing the complete `ds_list` class implementation (mimicking STL `list`)

9.1 Compare & Contrast: STL `vector` vs. STL `list`

- Same: Both are templated, sequential *containers*.
- Different: Only `vector` can be accessed using subscript (a.k.a. random-access).
  (Note: Implementing a similar operation for `list` would be inefficient.)

```
std::vector<double> v(10, 3.14);
std::cout << v[4] << std::endl;
v[5] = 6.02;
```

- Same: Elements of both can be accessed by iterators, using the dereference operator. The syntax for iterators with `vector` and `list` was intentionally designed to be similar to pointers with arrays.

```
// std::vector<double> container(10, 3.14);
// std::vector<double>::iterator itr = container.begin();
std::list<double> container(10, 3.14);
std::list<double>::iterator itr = container.begin();
for (itr = container.begin(); itr != container.end(); itr++) {
  if (*itr < 0.0) {
    *itr = 0.0;
  }
}
```

- Same: Iterators can be incremented or decremented to visit all elements in order within the container.

```
++itr;   itr++;   --itr;   itr--;
```

  These operations move the iterator to the next and previous locations in the vector, list, or string.
  The operations do not change the contents of container!

- Same: We can use `==` and `!=` with vector, list, and string iterators.
- Different: We can use `<`, `<=`, `>`, and `>=` vector and string iterators, but not with list iterators. Why not?
  We'll talk about that in the next section...

- Different: Only `vector` iterators can jump forward (or backward) by an arbitary integer number of "slots". (Note: Implementing a similar operation for `list` would be inefficient.)

```
std::vector<double>::iterator v_itr = v.begin();
v_itr = v_itr + 5;
```

- Same: Both have `push_back` and `pop_back`. These operations are constant time for `list`, and constant time on *average* for `vector`.
- Different: Only `list` has `push_front` and `pop_front`. These are constant time, $O(1)$, operations. (Note: Implementing similar operations for `vector` would be inefficient.)
- Same: Both have `erase` and `insert`.

  Different: . . . however, while they are constant time, $O(1)$, operations for `list`, they are linear time, $O(n)$, operations for `vector`.
- Same: Both have a built in `sort` that runs in O(n log n), with an optional comparison function.

  Different: The syntax is slightly different.

```
std::sort(my_vector.begin(),my_vector.end(),optional_comparison_function);
my_list.sort(optional_comparison_function);
```

- Different: Situations in which iterators are *invalidated*.
  - Iterators positioned on an STL `vector`, at or after the point of an `erase` operation, are invalidated.
  - Iterators positioned anywhere on an STL `vector` *may be* invalid after an `insert` (or `push_back` or `resize`) operation. Why? Because the array might need to be resized & reallocated (re-located) on the heap.
  - Iterators attached to an STL `list` are not invalidated after an `insert` or `push_back/push_front` or `erase/pop_back/pop_front`. (Except iterators attached to the erased element!)

9.2 Implementing `Vec<T>` Iterators

- Let's add iterators to our `Vec<T>` class declaration from Lecture 6:

```
public:
  // TYPEDEFS
  typedef T* iterator;
  typedef const T* const_iterator;

  // MODIFIERS
  iterator erase(iterator p);

  // ITERATOR OPERATIONS
  iterator begin() { return m_data; }
  const_iterator begin() const { return m_data; }
  iterator end() { return m_data + m_size; }
  const_iterator end() const { return m_data + m_size; }
```

- First, remember that `typedef` statements create custom, alternate names for existing types.

  `Vec<int>::iterator` is an iterator type defined by the `Vec<int>` class. It is just a `T *` (an `int *`). Thus, internal to the declarations and member functions, `T*` and `iterator` **may be used interchangeably**.
- Because the underlying implementation of `Vec` uses an array, and because pointers *are* the "iterator"s of arrays, the implementation of vector iterators is quite simple. *Note: the implementation of iterators for other STL containers is more involved! We'll see how STL `list` iterators work in a later lecture.*
- Thus, `begin()` returns a pointer to the first slot in the `m_data` array. And `end()` returns a pointer to the "slot" just beyond the last legal element in the `m_data` array (as prescribed in the STL standard).
- Furthermore, dereferencing a `Vec<T>::iterator` (dereferencing a pointer to type `T`) correctly returns one of the objects in the `m_data`, an object with type `T`.
- And similarly, the `++`, `--`, `<`, `==`, `!=`, `>=`, etc. operators on pointers automatically apply to `Vec` iterators. We don't need to write any additional functions for iterators, since we get all of the necessary behavior from the underlying pointer implementation.

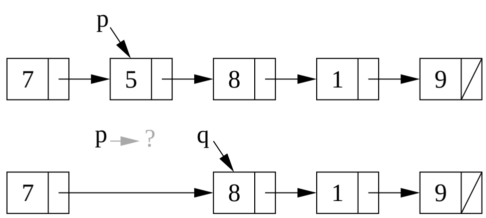
Finishing some Material from Lecture 8...

8.14 STL list (and STL vector) has an erase member function

- STL `list`s and `vector`s each have a special member function called `erase`. In particular, given list of ints `s`, consider the example:

```
std::list<int>::iterator p = s.begin();
++p;
std::list<int>::iterator q = s.erase(p);
```

- After the code above is executed:
  - The integer stored in the second entry of the list has been removed.
  - The size of the list has shrunk by one.
  - The iterator `p` does not refer to a valid entry.
  - The iterator `q` refers to the item that was the third entry and is now the second.



- To reuse the iterator `p` and make it a valid entry, you will often see the code written:

```
std::list<int>::iterator p = s.begin();
++p;
p = s.erase(p);
```

- Even though the `erase` function has the same syntax for vectors and for list, the vector version is $O(n)$, whereas the list version is $O(1)$.

8.15 Insert

- Similarly, there is an `insert` function for STL lists that takes an iterator and a value and adds a link in the chain with the new value immediately before the item pointed to by the iterator.
- The call returns an iterator that points to the newly added element. Variants on the basic insert function are also defined.

8.16 Example – *common confusion/mistake with STL iterators*

- NOTE: The example syntax below is the same for STL vector and STL lists.

```
std::vector<int> data;
std::vector<int>::iterator itr,itr2,itr3;
//std::list<int> data;
//std::list<int>::iterator itr,itr2,itr3;

data.push_back(100);  data.push_back(200);
data.push_back(300);  data.push_back(400);  data.push_back(500);

itr = data.begin(); // itr is pointing at the 100
++itr;              // itr is now pointing at 200
*itr += 1;          // 200 becomes 201

// itr += 1;      // NOTE: this syntax only works for vector/vector iterator
                  //       but it does not compile for list/list iterator
                  //       list iterators cannot be advanced like this

itr = data.end(); // itr is pointing "one past the last legal value" of data
itr--;            // itr is now pointing at 500;
```

```
itr2 = itr--;     // itr is now pointing at 400, itr2 is still pointing at 500
itr3 = --itr;     // itr is now pointing at 300, itr3 is also pointing at 300

// dangerous: decrementing the begin iterator is "undefined behavior"
//  (similarly, incrementing the end iterator is also undefined)
//  it may seem to work, but break later on this machine or on another machine!
itr = data.begin();
itr--;   // dangerous!
itr++;
assert (*itr == 100);  // might seem ok...  but rewrite the code to avoid this!
```

Now we can return to Lecture 9..

9.3 Definition of a Linked List

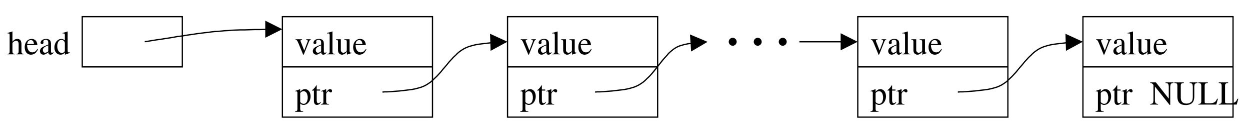
- A linked list is either:
  - Empty, or
  - Contains a node storing a value and a pointer to a linked list.
- Yes, the definition is recursive.
  And similarly, our implementation of many linked list functions can be recursive as well!

9.4 Visualizing & Implementing Linked Lists

- A linked list is made of `Node` objects. The `Node`s are be templated to allow linked lists of different types:

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

- The first node in the linked list is called the *head* node. In the example below, we store a pointer to the head node in a variable on the stack named `head`. The `Node`s of the linked list are dynamically allocated on the heap.



- It is important to have a separate variable pointer to the first node, since the "first" node may change as we edit the data stored in the list.
- Note that the diagram above is a conceptual view only. The memory locations could be anywhere – they aren't necessarily arranged in in this order, in adjacent memory locations. The actual values of the memory addresses aren't usually meaningful.
- The last node MUST have NULL for its pointer value — you will have all sorts of trouble if you don't ensure this!
- You should make a habit of drawing pictures of linked lists to figure out how to do the operations.

9.5 Stepping Through the List, Searching for a Value

- We'd like to write a function to determine if a particular value, stored in `x`, is 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.
  - We will need a loop to step through the linked list (using the pointer variable) and a check on each value.

9.6 Exercise: Write is_there

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

- If the input linked list chain contains n elements, what is the Big O Notation of our `is_there` function?

9.7 Overview: Adding an Element at the Front of the List

- We must create a *new* node.
- We must permanently update the `head` pointer variable's value.
  Therefore, we must pass the pointer variable ***by reference*** .

9.8 Exercise: Write push_front

```
template <class T> void push_front(Node<T>*& head, const T& value) {
```

- If the input linked list chain contains n elements, what is the Big O Notation of our `push_front` function?

9.9 Overview: Adding an Element at the Back 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 simple 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.

9.10 Exercise: Write push_back

```
template <class T> void push_back(Node<T>*& head, const T& value) {
```

- If the input linked list chain contains n elements, what is the Big O Notation of our `push_back` function?

9.11 Inserting a Node into a Singly-Linked List

- With a singly-linked list, we'll need a pointer to the node *before* the spot where we wish to insert the new item. *NOTE: This is not how STL list's insert function works!*
- Let's say that `p` is a pointer to this node, and `x` holds the value to be inserted. First, draw a picture to illustrate what is happening.

- Then, write a *fragment of code* that will do the insertion.

- Note: This code will not work if you want to insert `x` in a new node at the *front* of the linked list. Why not?

9.12 Limitations of Singly-Linked Lists

- We can only move through it in one direction
- We need a pointer to the node *before* the spot where we want to insert or erase.
  This requirement does not match the STL list specification!
- Appending a value at the end requires that we step through the entire list to reach the end.
  The Big O Notation does not match the STL list specification!

9.13 Generalizations of Singly-Linked Lists

- Three common generalizations (can be used separately or in combination):
  - Doubly-linked: allows forward and backward movement through the nodes
  - Circularly linked: simplifies access to the tail, when doubly-linked
  - Dummy header node: simplifies special-case checks

9.14 Transition to a Doubly-Linked List Structure

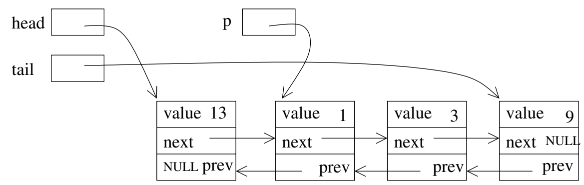
- The revised `Node` class has two pointers, one going "forward" to the successor in the linked list and one going "backward" to the predecessor in the linked list. We will have a `head` pointer to the beginning *and* a `tail` pointer to the end of the list.

```
template <class T> class Node {
public:
    Node() : next_(NULL), prev_(NULL) {}
    Node(const T& v) : value_(v), next_(NULL), prev_(NULL) {}
    T value_;
    Node<T>* next_;
    Node<T>* prev_;
};
```

- The tail pointer is not strictly necessary to access the data, but it facilitates efficient push-back operations.
- **Question:** If we have the tail pointer, do we still need the list to be doubly-linked?

9.15 Inserting a Node into the Middle of a Doubly-Linked List

- Suppose we want to insert a new node containing the value 15 following the node containing the value 1. We have a temporary pointer variable, `p`, that stores the address of the node containing the value 1.



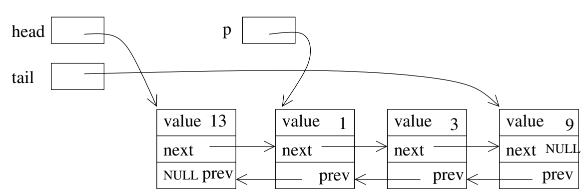
- What must happen? Editing the diagram above...
  - The new node must be created, using another temporary pointer variable to hold its address.
  - Its two pointers must be assigned.
  - Two pointers in the current linked list must be adjusted. Which ones?

  Assigning pointers for the new node MUST occur before changing pointers of the existing linked list nodes!

- **Exercise:** Write the code as just described. Focus first on the general case: Inserting a new into the middle of a list that already contains at least 2 nodes.

9.16 Removing a Node from the Middle of a Doubly-Linked List

- Now instead of inserting a value, suppose we want to remove the node pointed to by `p` (the node whose address is stored in the pointer variable `p`)
- Two pointers need to change before the node is deleted! All of them can be accessed through the pointer variable `p`.
- **Exercise:** Edit the diagram below, and then write this code.



9.17 Special Cases of Remove

- If `p==head` and `p==tail`, the single node in the list must be removed and both the `head` and `tail` pointer variables must be assigned the value `NULL`.
- If `p==head` or `p==tail`, then the pointer adjustment code we just wrote needs to be specialized to removing the first or last node.