CSCI-1200 Data Structures — Fall 2024
Lecture 10 — Doubly-Linked Lists & List Implementation

Review from Lecture 9

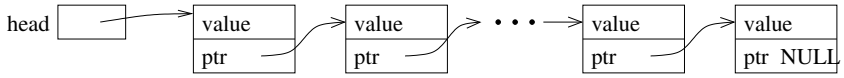
- Review of iterators, implementation of iterators in our homemade `Vec` class
- Starting *our own* basic linked list implementation
- Implementation of a few functions with a *single chain of forward links*
- Discussed how *singly-linked* lists don't match the STL list implementation.

Today's Lecture

- Review limitations of singly-linked lists
- Our own version of the STL `list<T>` class, named `dslist`,
- Implementing list iterators
- Writing `push_back`, `insert`, `erase`, and `copy_list`,
- STL List w/ iterators vs. "homemade" linked list with Node objects & pointers
- Common mistakes, unfortunately necessary `typename` syntax

10.1 Limitations of Singly-Linked Lists

- We can only move through it in one direction

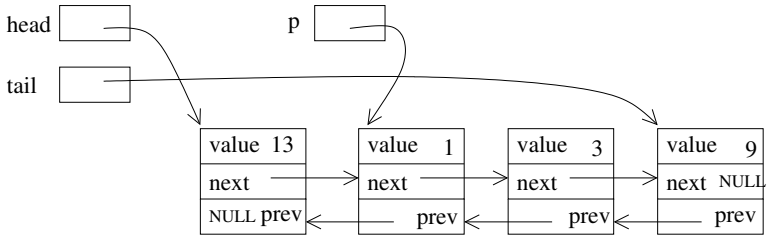


- We need a pointer to the node *before* the spot where we want to insert or erase.
- Appending a value at the end requires that we step through the entire list to reach the end.
- So what can we do? There are three common generalizations (can be used separately or in combination):
  - Doubly-linked: 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

10.2 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.

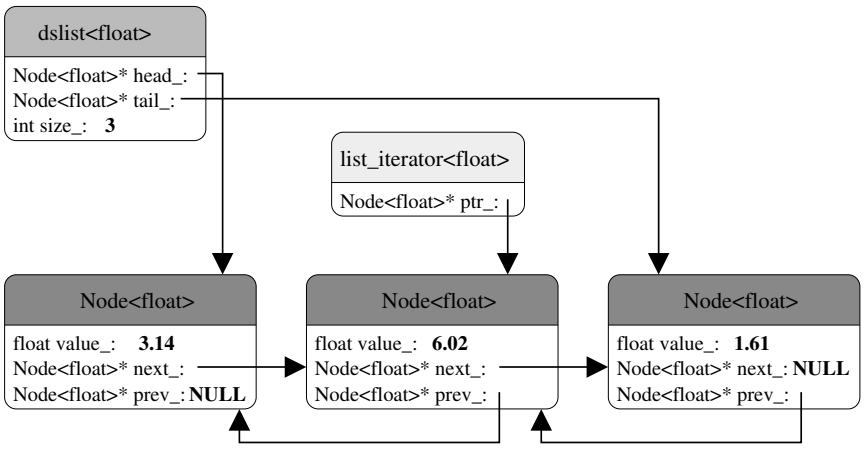
```
template <class T>
class Node {
public:
    T value_;
    Node<T>* next_;
    Node<T>* prev_;
};
```



- 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?

10.3 The `dslist` Class — Overview

- Our templated class `dslist` implements much of the functionality of the `std::list<T>` container and uses a doubly-linked list as its internal, low-level data structure. *The code is attached at the back of this handout.*
- Three classes are involved: the node class, the iterator class, and the dslist class itself.
  Below is a basic diagram showing how these three classes are related to each other:



- For each list object created by a program, we have one instance of the `dslist` class (the manager), and multiple instances of the `Node` (one for each value). For each iterator variable (of type `dslist<T>::iterator`) that is used in the program, we create an instance of the `list_iterator` class.

10.4 The Node Class

- It is ok to make all members public because individual nodes are never seen outside the list class.
  (`Node` objects are not accessible to a user through the public `dslist` interface.)
- Another option to ensure the `Node` member variables stay private would be to nest the entire `Node` class inside of the private section of the `dslist` declaration. We'll see an example of this later in the term.
- Note that the constructors initialize the pointers to NULL.

10.5 The Iterator Class

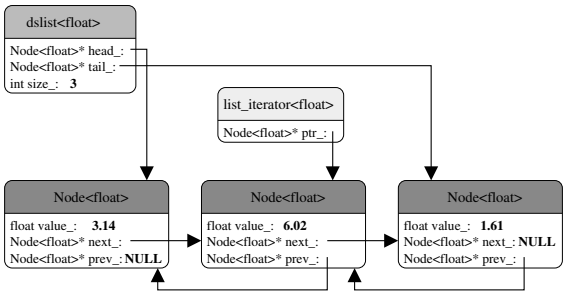
- Unfortunately, unlike our `Vec` class and the STL `vector` class, we can't simply typedef the `iterator` as just a pointer and get the desired functionality for free.
- We have to define a separate class that stores a pointer to a node in a linked list.
- The iterator constructor initializes the pointer. It is only called from `dslist<T>` class member functions.
- Stepping through the chain of the linked-list is implemented by the increment and decrement operators.
- The dereference operator, `operator*`, gives access to the contents of a node.
  (The user of a `dslist` class is never given full access to a `Node` object!)
- Two comparison operations: `operator==` and `operator!=`. Other comparisons (e.g., < or >) are not allowed.
- `dslist<T>` is a *friend class*, which allows access to the iterator's `ptr_` pointer variable
  (needed by `dslist<T>` member functions such as `erase` and `insert`).

10.6 The dslist Class

- Manages the actions of the iterator and node classes. Interfaces with the user through member functions.
- Maintains the head and tail pointers and the size of the list.
- Typedef for the `iterator` name.
- Prototypes for member functions, which are equivalent to the `std::list<T>` member functions.
- As a class managing dynamically-allocated memory, it must implement the "big 3": copy constructor, assignment operator, and destructor. These are implemented with private `copy_list` and `destroy_list` helper functions.

10.7 Exercise: Implement `dslist::push_back`

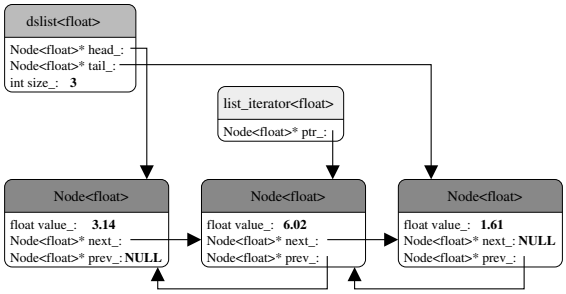
- Edit the diagram below to *push back* a new value to the list.



- Now write the code for `push_back`. Make sure to handle the corner case starting with an empty list.

10.8 Exercise: Implement `dslist::insert`

- Suppose we want to insert a new node containing the value 7.5 before the node containing the value 6.02. We have an iterator, which is a wrapper around a Node pointer, attached to the Node containing 6.02.



- What must happen? First, let's make the edits to 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?

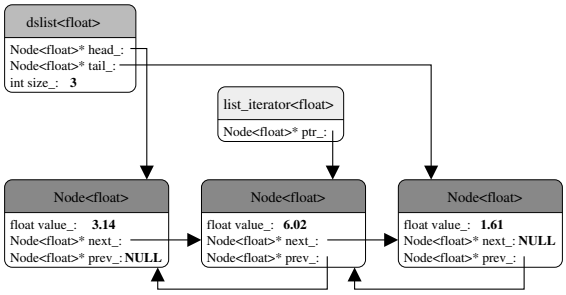
  Assigning pointers for the new node MUST occur before changing pointers of the existing linked list nodes!

- Now 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.

- Then let's think about the corner cases. Do we need to write any additional code to handle the special cases of empty list, one element lists, inserting at the front or back of the list?

10.9 Exercise: Implement `dslist::erase`

- Now instead of inserting a value, suppose we want to remove the node pointed to by the iterator?
- Two pointers need to change before the node is deleted! All of them can be accessed through the iterator's pointer variable. First edit the diagram below:



- Now write the code:

- Again, we must consider all corner cases / special cases:
  - 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.

10.10 Exercise: Implement `dslist::copy_list`

- Now let's write the helper function `copy_list`.

10.11 Basic Linked Lists Mechanisms: Common Mistakes

Here is a summary of some common mistakes. Review this list when you get stuck on HW or a Test.

- Allocating a new node (using `new`) to step through the linked list – when only a pointer variable is needed!
- Confusing the `.` and the `->` operators. (The compiler will help you – except on a test!)
- Not setting the pointer from the last node to NULL.
- Not considering all special cases: inserting / removing, first or last element, only 1 element, empty list, etc.
- Applying the `delete` operator to a node (calling the operator on a pointer to the node) before it is appropriately disconnected from the list. Delete should be done only after all pointer manipulations are completed.
- Pointer manipulations that are out of order. These can ruin the structure of the linked list.
- Trying to use STL iterators to visit elements of a "home made" linked list chain of nodes.
  (And the reverse.... trying to use `->next` and `->prev` with STL list iterators.)

10.12 Unfortunate C++ Template Implementation Detail - Using `typename`

- The use of typedefs within a templated class, for example the `dslist<T>::iterator` can confuse the compiler because it is a *template-parameter dependent name* and is thus ambiguous in some contexts. (Compiler asks: Is "iterator" a value or is it a type?)
- If you get a strange error during compilation (where the compiler is clearly confused about seemingly clear and logical code), you will need to explicitly let the compiler know that it is a type by putting the `typename` keyword in front of the type. For example, inside of the `operator==` function:

```
typename dslist<T>::iterator left_itr = left.begin();
```

- Don't worry, we'll never test you on where this keyword is needed. Just be prepared that you may need to use it when implementing templated classes that use typedefs.

```
#ifndef dslist_h_
#define dslist_h_
// A simplified implementation of the STL list container class,
// including the iterator, but not the const_iterators.  Three
// separate classes are defined: a Node class, an iterator class, and
// the actual list class.  The underlying list is doubly-linked, but
// there is no dummy head node and the list is not circular.
#include <cassert>

// ---------------------------------------------------------------
// NODE CLASS
template <class T>
class Node {
public:
  // CONSTRUCTORS: set the pointers to NULL
  Node() : next_(NULL), prev_(NULL) {}
  Node(const T& v) : value_(v), next_(NULL), prev_(NULL) {}
  // REPRESENTATION
  T value_;
  Node<T>* next_;
  Node<T>* prev_;
};

// A "forward declaration" of this class is needed
template <class T> class dslist;

// ---------------------------------------------------------------
// LIST ITERATOR
template <class T>
class list_iterator {
public:
  // default constructor, copy constructor, assignment operator, & destructor
  list_iterator(Node<T>* p=NULL) : ptr_(p) {}
  // NOTE: the implicit compiler definitions of the copy constructor,
  // assignment operator, and destructor are correct for this class.

  // The dereferencing operator gives access to the value at the pointer.
  T& operator*()  { return ptr_->value_;  }
  // pre-increment, e.g., ++iter
  list_iterator<T>& operator++() {
    ptr_ = ptr_->next_;
    return *this;
  }
  // post-increment, e.g., iter++
  list_iterator<T> operator++(int) {
    list_iterator<T> temp(*this);
    ptr_ = ptr_->next_;
    return temp;
  }
  // pre-decrement, e.g., --iter
  list_iterator<T>& operator--() {
    ptr_ = ptr_->prev_;
    return *this;
  }
  // post-decrement, e.g., iter--
  list_iterator<T> operator--(int) {
    list_iterator<T> temp(*this);
    ptr_ = ptr_->prev_;
    return temp;
  }
```

```
  // Comparions operators are straightforward
  bool operator==(const list_iterator<T>& r) const {
    return ptr_ == r.ptr_; }
  bool operator!=(const list_iterator<T>& r) const {
    return ptr_ != r.ptr_; }

  // the dslist class needs access to the private ptr_ member variable
  friend class dslist<T>;

private:
  // REPRESENTATION
  Node<T>* ptr_;    // ptr to node in the list
};

// --------------------------------------------------------------
// LIST CLASS DECLARATION
// Note that it explicitly maintains the size of the list.
template <class T>
class dslist {
public:
  // default constructor, copy constructor, assignment operator, & destructor
  dslist() : head_(NULL), tail_(NULL), size_(0) {}
  dslist(const dslist<T> &old) { copy_list(old); }
  dslist& operator= (const dslist<T>& old);
  ~dslist() { destroy_list(); }

  // typedef allows use of 'dslist<int>::iterator' syntax
  typedef list_iterator<T> iterator;

  // simple accessors & modifiers
  unsigned int size() const { return size_; }
  bool empty() const { return head_ == NULL; }
  void clear() { destroy_list(); }

  // read/write access to contents
  const T& front() const { return head_->value_;  }
  T& front() { return head_->value_; }
  const T& back() const { return tail_->value_; }
  T& back() { return tail_->value_; }

  // modify the linked list structure
  void push_front(const T& v);
  void pop_front();
  void push_back(const T& v);
  void pop_back();
  iterator erase(iterator itr);
  iterator insert(iterator itr, const T& v);
  iterator begin() { return iterator(head_); }
  iterator end() { return iterator(NULL); }

private:
  // private helper functions
  void copy_list(const dslist<T>& old);
  void destroy_list();

  //REPRESENTATION
  Node<T>* head_;
  Node<T>* tail_;
  unsigned int size_;
};
```

```
// ---------------------------------------------------------------
// LIST CLASS IMPLEMENTATION
template <class T>
dslist<T>& dslist<T>::operator= (const dslist<T>& old) {
  // check for self-assignment
  if (&old != this) {
    destroy_list();
    copy_list(old);
  }
  return *this;
}

template <class T> void dslist<T>::push_back(const T& v) {
 // We will implement this in Lecture 10
}
template <class T> void dslist<T>::push_front(const T& v) {
  // You will implement this in Lab 7
}
template <class T> void dslist<T>::pop_front() {
  // You will implement this in Lab 7
}
template <class T> void dslist<T>::pop_back() {
  // You will implement this in Lab 7
}

// operator== asks "Do these lists look the same (length & contents)?"
template <class T>
bool operator== (dslist<T>& left, dslist<T>& right) {
  if (left.size() != right.size()) return false;
  typename dslist<T>::iterator left_itr = left.begin();
  typename dslist<T>::iterator right_itr = right.begin();
  // walk over both lists, looking for a mismatched value
  while (left_itr != left.end()) {
    if (*left_itr != *right_itr) return false;
    left_itr++; right_itr++;
  }
  return true;
}
template <class T>
bool operator!= (dslist<T>& left, dslist<T>& right){ return !(left==right); }

template <class T>
typename dslist<T>::iterator dslist<T>::insert(iterator itr, const T& v) {
 // We will implement this in Lecture 10
}
template <class T>
typename dslist<T>::iterator dslist<T>::erase(iterator itr) {
 // We will implement this in Lecture 10
}
template <class T>
void dslist<T>::copy_list(const dslist<T>& old) {
 // We will implement this in Lecture 10
}
template <class T>
void dslist<T>::destroy_list() {
  // You will implement this in Lab 7
}

#endif
```