

CSCI-1200 Data Structures — Fall 2022

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 – forward-only, singly-linked
- Implemented: stepping through, push front, push back, and simple insert

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

- Push back:

```
template <class T> void push_back( Node<T>* & head, T const& value ) {
    if (head == NULL) {
        head = new Node<T>;
        head->value = value;
        head->ptr = NULL;
    } else {
        push_back(head->next, value);
    }
}
```

- Inserting a Node into a Singly-Linked List - *if p points at node before where you want the insertion*

```
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
```

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?

Today's Lecture

- Limitations of singly-linked lists
- Doubly-linked lists: Structure, Insert, & Remove
- Our own version of the STL `list<T>` class, named `dslist`, Implementing list iterators
- Common mistakes, STL List w/ iterators vs. “homemade” linked list with Node objects & pointers

10.1 Exercise: Removing a Node from a Singly-Linked List

- The remove operation itself requires a pointer to the node *before* the node to be removed.
- Suppose `p` points to a 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. Note: Removing the first node is an important special case.
- 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. Draw a picture of each scenario.

10.2 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.

10.3 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

10.4 Transition to a Doubly-Linked List Structure

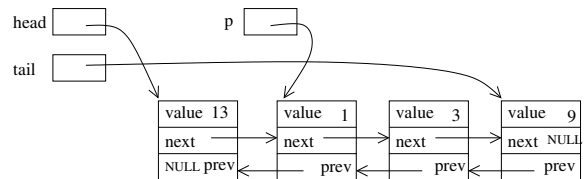
- The revised Node class has two pointers, one going “forward” to the successor in the linked list and one going “backward” to the predecessor in the linked list. We will have a **head** pointer to the beginning *and* a **tail** pointer to the end of the list.

```
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?

10.5 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?

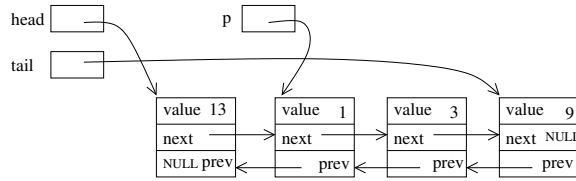
Assigning pointers for the new node **MUST** occur before changing pointers of the existing linked list nodes!

- **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.

10.6 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.

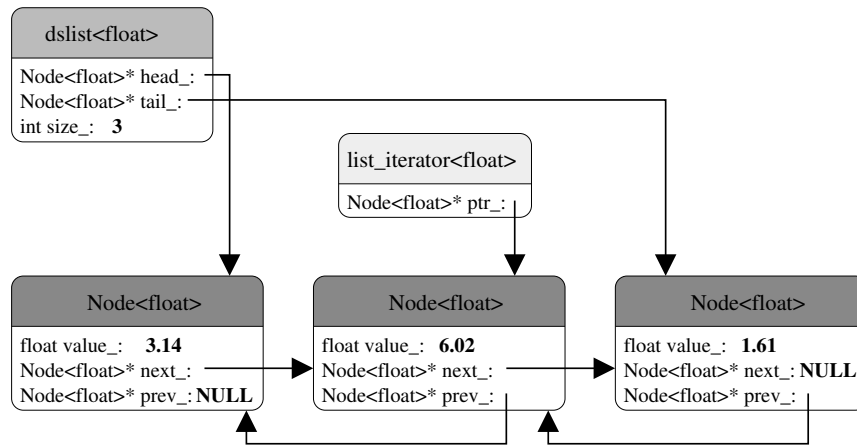


10.7 Special Cases of Remove

- 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.

10.8 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.
- 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.

10.9 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`.

10.10 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. Will only be called from the `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 comparison (e.g., `<` or `>`) not allowed.
- `dslist<T>` is a friend class to allow access to the iterators `ptr_` pointer variable (needed by `dslist<T>` member functions such as `erase` and `insert`).

10.11 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.

10.12 Exercises

1. Write `dslist<T>::push_front`
2. Write `dslist<T>::erase`

10.13 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; 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.)

10.14 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.

```

#ifdef 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:
    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

    // dereferencing operator gives access to the value at the pointer
    T& operator*() { return ptr_->value_; }

    // increment & decrement operators
    // 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;
    }
};

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

// Comparisons 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_; }

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 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>
dplist<T>& dplist<T>::operator=(const dplist<T>& old) {
    // check for self-assignment
    if (&old != this) {
        destroy_list();
        copy_list(old);
    }
    return *this;
}

template <class T>
void dplist<T>::push_front(const T& v) {

}

template <class T>
void dplist<T>::pop_front() {

}

template <class T>
void dplist<T>::push_back(const T& v) {

}

template <class T>
void dplist<T>::pop_back() {

}

// do these lists look the same (length & contents)?
template <class T>
bool operator==(dplist<T>& left, dplist<T>& right) {
    if (left.size() != right.size()) return false;
    typename dplist<T>::iterator left_itr = left.begin();
    typename dplist<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!=(dplist<T>& left, dplist<T>& right) { return !(left==right); }

template <class T>
typename dplist<T>::iterator dplist<T>::erase(iterator itr) {

}

template <class T>
typename dplist<T>::iterator dplist<T>::insert(iterator itr, const T& v) {

}

template <class T>
void dplist<T>::copy_list(const dplist<T>& old) {

}

```