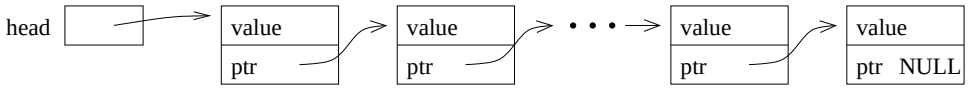
CSCI-1200 Data Structures — Spring 2013
Lectures 10 — Linked Lists, Part I

Review from Lecture 9

- Returning references to member variables from member functions
- Review of iterators and iterator operations
- STL Lists, `erase` and `insert` on lists
- Differences between indices and iterators, differences between lists and vectors
- Introductory linked lists & operations on linked lists

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



  - Stepping through a list

```
template <class T>
bool is_there(Node<T>* head, const T& x) {
  for (Node<T> *p  = head; p != NULL ; p = p->ptr) {
    if (p->value == x) return true;
  }
  return false;
}
```

  - Push back

```
template <class T>
void push_back( Node<T>* & head, T const& value ) {
  // test for empty list
  if (head == NULL) {
    Node<T> *q = new Node<T>;
    q->value = value;
    q->ptr = NULL;
    head = q;
  } else {
    Node<T> *p  = head;
    for ( ; p->ptr != NULL ; p = p->ptr) {
      // do nothing, just walk to the end of the list
    }
    Node<T> *q = new Node<T>;
    q->value = value;
    q->ptr = NULL;
    p->ptr = q;
  }
}
```

- STL List w/ iterators vs. "homemade" linked list with Node objects & pointers

Today's Lecture

- Basic linked list operations, continued: Insert & Remove
- Common mistakes
- Limitations of singly-linked lists
- Doubly-linked lists:
  - Structure
  - Insert
  - Remove

10.1 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.
- If `p` is a pointer to this node, and `x` holds the value to be inserted, then the following code will do the insertion. Draw a picture to illustrate what is happening.

```
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?

10.2 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.

10.3 Exercise: Remove a Node

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. 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.

10.4 Exercise: List Copy

Write a *recursive* function to copy all nodes in a linked list to form an new linked list of nodes with identical structure and values. Here's the function prototype:

```
template <class T> void CopyAll(Node<T>* old_head,  Node<T>*& new_head) {
```

10.5 Exercise: Remove All

Write a *recursive* function to delete all nodes in a linked list. Here's the function prototype:

```
template <class T> void RemoveAll(Node<T>*& head) {
```

10.6 Basic Linked Lists Mechanisms: Common Mistakes

Here is a 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 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 appropriately disconnected from the list. 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.

10.7 Limitations of Singly-Linked Lists

- We can only move through it in one direction
- We need a pointer to the node **before** the node that needs to be deleted.
- Appending a value at the end requires that we step through the entire list to reach the end.

10.8 Generalizations of Singly-Linked Lists

- Three common generalizations:
  - 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
- Today we will explore and implement a doubly-linked structure.

10.9 `ds_list`: Our own implemention mimicing STL's list

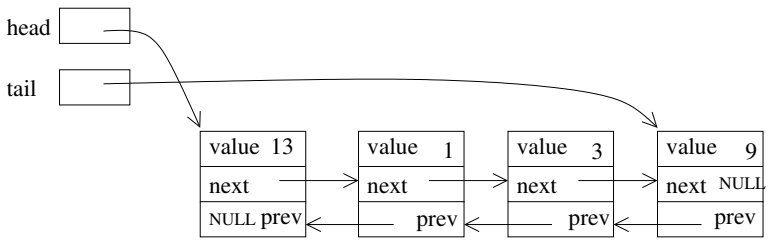
- We will start the implementation of `ds_list`, our own implementation of the STL list. Nodes will be templated and have two pointers, one going "forward" to the successor in the linked list and one going "backward" to the predecessor in the linked list. We will have a pointer to the beginning *and* 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_;
};
```

- First we'll reimplement some of the basic mechanisms we've already worked through for singly-linked lists. In the next lecture we'll build the full `ds_list` class and will define the list iterators as a class inside a class.

10.10 The Structure of Doubly-Linked Lists

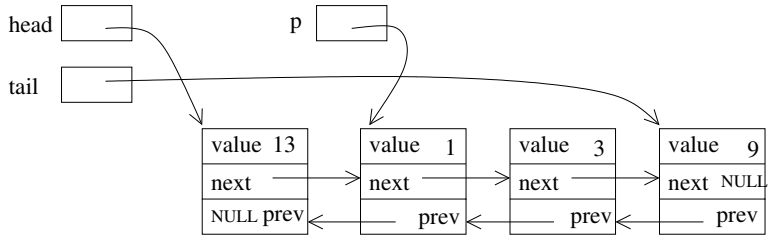
- Here is a picture of a doubly-linked list holding 4 integer values:



- Note that we now assume that we have both a `head` pointer, as before and a `tail` pointer variable, which stores the address of the last node in the linked list.
- The tail pointer is not strictly necessary, but it allows immediate access to the end of the list for efficient push-back operations.

10.11 Inserting in 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. Here's a picture of the state of affairs:



- What must happen?
  - 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 the pointers for the new node MUST occur before changing the pointers for the current linked list nodes!

- At this point, we are ignoring the possibility that the linked list is empty or that `p` points to the tail node (`p` pointing to the head node doesn't cause any problems).

- **Exercise:** write the code as just described.

10.12 Removing from the Middle of a Doubly-Linked List

- Suppose now instead of inserting a value 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:** write this code.

10.13 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.

- All of these will be built into the `erase` function that we write as part of our `ds_list` class.