Tuesday, September 15, 2009
3:31 PM

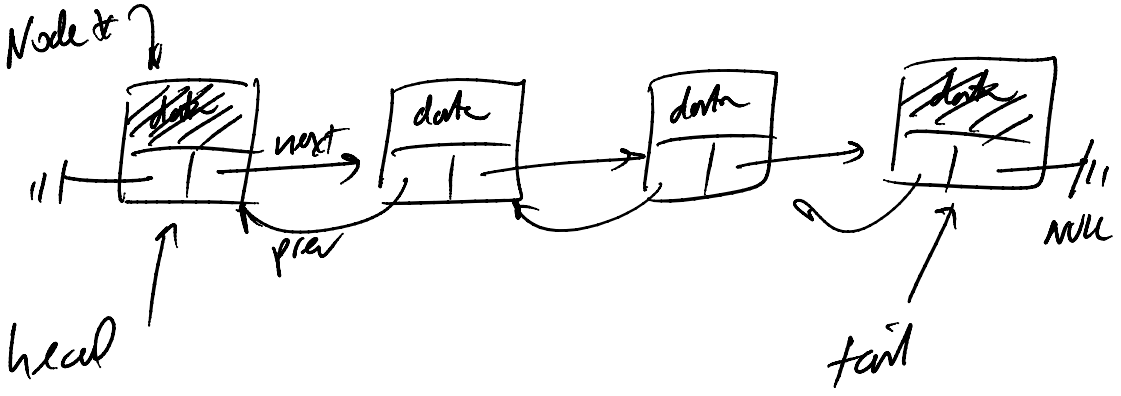
9/15/09

You've seen:

ADTs
- simple array
- templates
- binary heap (array acting as binary tree)
- singly linked list (lab 4)

Today: doubly-linked list
- several implementations
- in lab: Node as a separate class
1. — first thing: put into List class as embedded class
2. — textbook's use/implementation of iterators
3. — textbook bug fixes

- Doubly-linked list as implemented by text
  uses two sentinel nodes



Tuesday, September 15, 2009
3:40 PM

§3.3: types of operations we want List to provide

```
template <typename T>
class List {
  private:
    struct Node {};
  public:
```

"container"

begin(): returns pointer to first node of the list

end(): return pointer to tail (not to node before tail)

insert(Node *, const T&) } on/off list
erase(Node *)

internal (for speed)

size() { return sz; }
empty() { return size() == 0; }
clear() { while (!empty()) pop_front(); }

pop_front() { erase(head->next); }
pop_back() { erase(tail->prev); }

T& front() { return head->next->data; }
const T& front() const { return " }

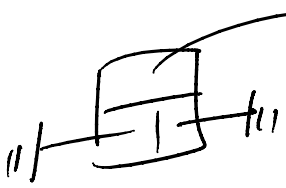
T& back() { return tail->prev->data; }
const T& back() { " }

push_front(const T& o) { insert(head->next, o); }

```
                                    //insert in front of this node
push_back(const Td o){insert(tail,o);}

private:
        int sz;
        Node* head;
        Node* tail;
};
```

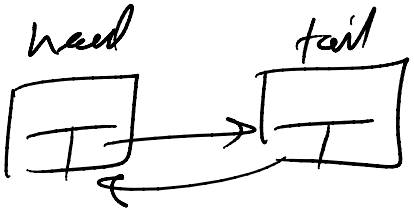
```
struct Node                    // a "class" public: by default
{                              // class : private by "
                               // I also think struct as something
                               //   with no member functions
   T    data;
   Node *prev;
   Node *next;
   Node(const T& d=T(), Node *p=NULL, Node *n=NULL):
        data(d), prev(p), next(n) { };
};
```

By default, Node node;



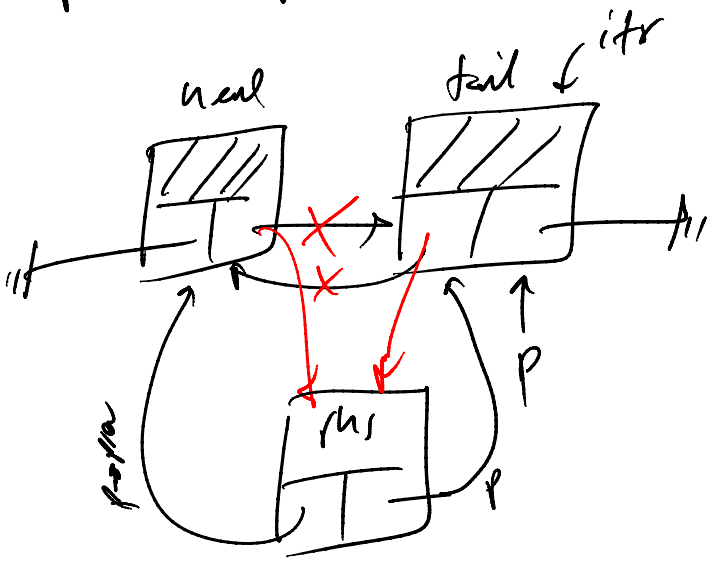
your set

```
template <typename T>
List<T>::List()  // constructor
{
    sz = 0;
    head = new Node;
    tail = new Node;
    head->next = tail;
    tail->prev = head;
}
```



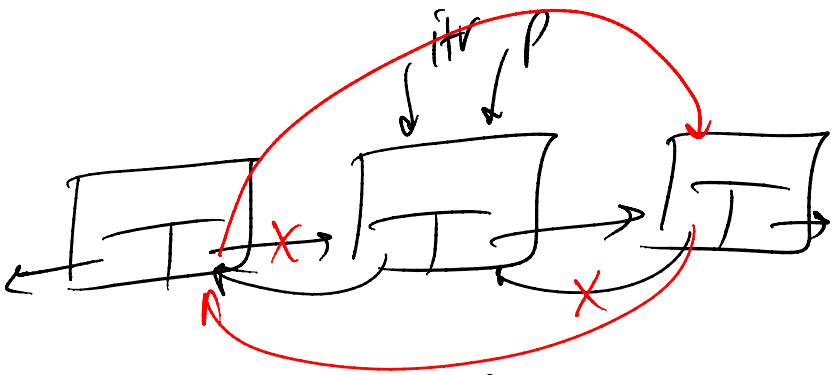

```
template <typename T>
typename List<T>::Node* List<T>::insert(Node* itr,
                                        const T& rhs) {
}
    Node *p = itr;

    sz++;
```

```
    sz++;
    return (p->prev = p->prev->next = new Node(val, p->prev,
                                               p));
}
```

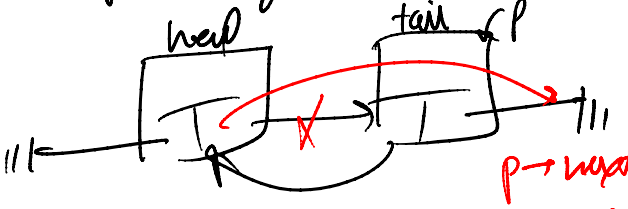



```
erase(Node* itr)
{
    Node* p = itr;
    p->prev->next = p->next;
    p->next->prev = p->prev;
    delete p;
    sz--;
}
```

what about error checking?



* what if we try to delete tail?



p->next->prev

seg fault or bus error

Usage:

```
List<int>  list;

for (int i=0; i<10; i++) {
    num = rand()
    list.push_back(num)
}

while (!list.empty()) {
    std::cout << list.front();
    list.pop_front()
}
```

Tuesday, September 15, 2009
4:21 PM

§ 3.3.1

Iterators

- fancy wrappers for pointers to nodes
- when the "Node List" class is there to iterate over list, I need to "know" something about Node*, local copy.
- if Node is part of List class (as it should be) then its privates should be inaccessible to user
- so, create iterators, to allow this:

```
for( List<int>::iterator itr = list.begin();
        itr != list.end();  ++itr)
    cout << *itr;
```

↑ dereference itr; output user data in Node

- need's
  - a List<T>::iterator class (embedded in List)
  - need List class ↑ to provide begin(), end()
  - need List class to provide operator!=()
    (which needs operator==())
  - need operator++ (infix & postfix)
    ++itr itr++
  - need dereferencing operator*()

- need dereferencing operator*( )

Ann, double all these, one set for iterator (mutator)
& one set for const_iterator (accessor)
(just like our T& operator[] mutator
const T& operator[] const accessor)

to implement these, use const_iterator as base class & iterator as derrived class

inheritance { const_iterator (base class)
↑ is-a
iterator (derived class)

- iterator has one data member: Node* curr;
(current node pointer)

Tuesday, September 15, 2009
4:34 PM

§ 3.5 List implementation

```
template <typename T>
class List {
    private:
    struct Node
    {
        T data;
        Node* prev;
        Node* next;
        Node(const T& ---)   // as before
    };
    public:
    class const_iterator
    {
        public:
        const_iterator() : curr(NULL) { };
        const T& operator*() const
        {
            return retrieve();
        }
        const_iterator& operator++()   // if blank ++itr
        {
            curr = curr->next;
            return *this;
        }
        const_iterator& operator++(int) // itr++
            const_iterator old = *this;
            ++(*this);
            return old;
        }
    see text
```