Announcements

- Program checkpoint due Friday
- Program due Monday
Recap of Vectors:

Idea: extend arrays so that they grow when needed

But keep things efficient
**Running times**

**Constructor:** $O(1)$

**Operator []:** $O(1)$

**Destructor:** $O(1)$

**Insert:** $O(n)$

**Remove:** $O(n)$

**Push_back:** $O(n)$ (but not very often)
Proposition: The running time of making $N$ push-back operations in an empty array is $O(N)$.

Proof: (called Amortized analysis)
Consider virtual dollars—make each operation “pay” for its running time.

$1 - easy$  $N - push-back when capacity = size$

Instead, make each pay $3 to a bank and withdraw only what they need for operation.
\[ N = 2^i \]

\[ \begin{align*}
3(2^{i-1}) - 1 \cdot 2^{i-1} &= 2 \cdot 2^{i-1} \text{ in bank} \\
&= 2^i
\end{align*} \]

How long does it take to double \& copy everything down?

Total work: \((\# \text{ calls})(\text{dollars charged})\)

\[ = N \cdot 3 = O(N) \]
Summary:

Each push-back takes $\frac{m}{n} = O(1)$

Amortized time $\frac{m}{n} = O(1)$
Linked lists

Motivation: The running time of insert in a vector is awful.

Idea: If we know where an element should go, inserting should be faster.
Doubly Linked List: Insert

\( \text{Insert}(\text{LAX}) \)

(saw this code in add function)

\( O(1) \)
Problem: What do we need the user to have in order to implement insert?

Solution: "Wrap" pointers into iterators to write functionality in our iterator class.
An iterator will give the user a “pointer,” but with a heavily controlled structure (so they can’t manipulate the nodes directly). *(no delete)*

Compromise between hiding the underlying data and allowing the user to specify a location directly.
Check out STL functions:

- `insert`
- `erase`
- `begin`
- `end`
- `operator ++`
- `push_back`
- `(lots)`
Usage example

```cpp
// Scoping
List<int> mylist;
List<int>::iterator it;

mylist.push_back(5);
mylist.push_back(7);
mylist.push_back(9);

it = mylist.begin();

it++;
mylist.insert(it, 6);
++it
```
```cpp
template <typename Object>
class List {

protected:

    struct Node {
        Object _data;

        Node* _prev;
        Node* _next;

        Node (const Object &data, Node* next, Node* prev) {
            _data = data;
            _next = next;
            _prev = prev;
        }
    }
};
```
**Iterator class**: What should we code?

```cpp
public: // in list class
class iterator {

private:
    Node* _current;

public:
    iterator () : _current(NULL) {};
    iterator (Node* ptr) : _current(ptr) {};
    iterator (const iterator &other) : _current (other._current) {};
```
// prefix
void operator ++ ( ) { }  
3
  _current = _current -> next ;

// postfix
void operator += ( int dummy ) { }  
3
  // same

Object& operator * ( ) { }  
3
  return ( _current -> _data ) ;

Object& operator -> ( ) { }  
3
  return ( _current -> _data ) ;