Linked lists, continued.
Review from last lecture

- Last lecture we looked briefly at how a linked list could be conceptualized as a “chain” of nodes.
- A Node is simply a “link” in the chain.
Review from last lecture

- Each Node contains a reference to an Object that the user wants to store (node._data).

- Each Node also contains a reference to the next “link” (Node) in the chain (node._next).
**Nodes**

- **Nodes** in a `LinkedList` play an analogous role to the “slots” (elements) of an array in an `ArrayList`.

```java
list.add(o1);
list.add(o2);
list.add(o3);
```

```
ArrayList
Object[] _underlyingStorage

<table>
<thead>
<tr>
<th>o1</th>
<th>o2</th>
<th>o3</th>
</tr>
</thead>
</table>

int _numElements: 3
```

```
LinkedList
_Node head

Node o1 → Node o2 → Node o3 → null
```
Elements of an array

- In an array, there is no need to link the elements using pointers because array elements are always adjacent to each other in memory.

- For an `Object[]` array, the address of element 1 is just 4 bytes more than the address of element 0.
Elements of an array

- Arrays are always stored contiguously in memory (in one big chunk):
  - Addr of element $i = \text{BaseAddr} + i \times 4$
- Easy to jump to a particular index using the [], operator.

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>8192</td>
<td></td>
</tr>
<tr>
<td>8196</td>
<td>o1</td>
</tr>
<tr>
<td>8200</td>
<td>o2</td>
</tr>
<tr>
<td>8204</td>
<td>o3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Nodes of a linked list

- With linked lists, nodes can be allocated anywhere in memory.
  - No need for contiguity; hence, more flexible.
- However, this means that it takes more effort to compute the address of any particular node.
- We must “iterate through” all nodes before it.
Finding a particular node

- Let’s assume we have a linked list containing 3 nodes.
- We have a _head pointer to the first node.
- How do we access the _data contained in the 3rd node?

![Diagram showing a linked list with nodes and pointers](image-url)
Finding a particular node

```java
final Object thirdElement = _head._next._next._data;
```
Finding a particular node

• Alternatively, we could use a for-loop:
  Node cursor = _head;
  for (int i = 0; i < 2; i++) {  // Why only 2?
    cursor = cursor._next;
  }
  final Object thirdElement = cursor._data;
Iterating through the whole list

- Suppose we wish to iterate through the entire list and print out the _data in each node?
Iterating through the whole list

- Suppose we wish to iterate through the entire list and print out the _data in each node?

\[
\text{Node cursor} = \_\text{head};
\]
Iterating through the whole list

- Suppose we wish to iterate through the entire list and print out the _data in each node?

Node cursor = _head;
while ( ) {

}

Node cursor = _head;
while () {

}
Iterating through the whole list

• Suppose we wish to iterate through the entire list and print out the _data in each node?

```java
Node cursor = _head;
while (cursor != null) {
    // Print or process data
    cursor = cursor._next;
}
```

Diagram:
- _head
- Node with _next and _data
- Node with _next and _data
- ... (ellipsis)
- Node with _next and _data
- null
Iterating through the whole list

- Suppose we wish to iterate through the entire list and print out the _data in each node?

```java
Node cursor = _head;
while (cursor != null) {
    System.out.println(cursor._data);
}
```
Iterating through the whole list

• Suppose we wish to iterate through the entire list and print out the _data in each node?

Node cursor = _head;
while (cursor != null) {
    System.out.println(cursor._data);
    cursor = cursor._next;
}
// Done!

Node cursor = _head;
while (cursor != null) {
    System.out.println(cursor._data);
    cursor = cursor._next;
}
// Done!
Iterating through the whole list

- Alternatively, we could use a for-loop:

```java
for (Node cursor = _head; cursor != null; cursor = cursor._next) {
    System.out.println(cursor._data);
}
// Done!
```

[Diagram of a linked list with nodes and arrows indicating traversal]

Monday, July 9, 12
Adding a new node

- The “iteration” code described above assumes that a linked list already exists.
- How is the “chain of nodes” actually constructed?
Before discussing how to implement the \texttt{add(o)} method, let's first “concretify” the linked list class itself.

Let's create a \texttt{SinglyLinkedList} class that implements an (expanded) \texttt{List} interface...
public interface List {
    // Adds o to the "back" of the list, i.e.,
    // o becomes the element with the highest
    // index in the List.
    void add (Object o);

    // Returns the element stored at the specified
    // index.
    Object get (int index)
        throws IndexOutOfBoundsException;

    // Removes the element stored at the specified
    // index.
    void remove (int index)
        throws IndexOutOfBoundsException;

    // Returns the number of elements stored in
    // the List.
    int size ();
}
class SinglyLinkedList

• We will implement the Node class as an inner-class of SinglyLinkedList.

• More on inner-classes later.

• We will use two instance variables:
  Node _head, _tail;
class SinglyLinkedList

• Note the slight inconsistency with previous slides:
  • In our SinglyLinkedList implementation, we will be using “dummy nodes” for the head and tail.
  • These nodes will simplify the implementation.
  • Dummy nodes are nodes whose _data fields are always null -- they contain no data from the “user”.
  • The dummy nodes will always exist, even if the user hasn’t added any data yet.
  • Nodes for the user’s data will be created between the dummy head and tail nodes.
public class SinglyLinkedList implements List {
    class Node {  // Inner-class
        Node _next;
        Object _data;
    }
    private Node _head, _tail;

    SinglyLinkedList () {
        // Instantiate dummy head and tail nodes
        _head = new Node();
        _tail = new Node();

        // Link _head to _tail
        _head._next = _tail;
    }

    void add (Object o) { ... }
    Object get (int index) throws IndexOutOfBoundsException { ... }
    void remove (int index) throws IndexOutOfBoundsException { ... }
    int size () { ... }
}
After construction

- After the constructor has been called, our `SinglyLinkedList` object looks like this:
Let’s consider how to implement the `add(o)` method.

As a “rule” when implementing `add(o)`, we will maintain the invariant that `_head` and `_tail` point to dummy nodes.

- We will never use them to store real user data.
- An invariant is a condition that always holds true.
```plaintext
void add (Object o)

• Given the dummy head and tail nodes, we can add a new node to our chain in 4 steps:

  1. Instantiate a new Node object.

  2. Set its _data field to equal o.

  3. Iterate a "cursor" from the dummy head towards the tail, stopping just before the dummy tail.

  4. Insert the new Node just after cursor.
```
void add (Object o)

1. Instantiate a new Node object.
   final Node node = new Node();
void add (Object o)

2. Set its _data field to equal o.

```java
node._data = o;
```

Monday, July 9, 12
void add (Object o)

3. Iterate from the head towards the tail, stopping just before the tail.

Node cursor = _head;
while (cursor._next != _tail) {  // Why?
    cursor = cursor._next;
}
4. Insert the new node just after cursor.

```java
node._next = cursor._next;
```
void add (Object o)

4. Insert the new Node just after cursor.

```java
node._next = cursor._next;
cursor._next = node;
```
void add (Object o)

Done!

If we pull the chain “taut”...
void add (Object o)

...it will look more like what we started with...

Notice: _head and _tail still point to the dummy nodes, and they contain no “real” data -- as intended.
Reality check

• Why do we need to iterate the cursor to the node just *before* the dummy tail?
Let's add one more node...

1. Instantiate a new `Node` object.

```java
final Node node = new Node();
```
Let’s add one more node...

2. Set its `_data` field to equal `o2`.

```cpp
node._data = o;
```
Let's add one more node...

3. Iterate from the head towards the tail, stopping just before the tail.

```java
Node cursor = _head;
while (cursor._next != _tail) {
    cursor = cursor._next;
}
```
Let's add one more node...

4. Insert the new node just after cursor.

```java
node._next = cursor._next;
```
Let's add one more node...

4. Insert the new **Node** just after cursor.

```java
Node.next = cursor.next;
cursor.next = node;
```
Let’s add one more node...

Done (and pulled taut again)!

Notice: Object o2 is stored just “after” o, as required by add(o) specification in our List interface.
Reality check

- Which objects should \texttt{get(0)} and \texttt{get(1)} return on this list below?

\begin{itemize}
\item \texttt{_next}
\item \texttt{_data}
\end{itemize}

\texttt{Node}

\texttt{_next}

\texttt{_data}

\texttt{SinglyLinkedList}

\texttt{dummy}

\texttt{Node}

\texttt{_next}

\texttt{_data: 0}

\texttt{Node}

\texttt{_next}

\texttt{_data: 02}

\texttt{Node}

\texttt{_next}

\texttt{_data}

\texttt{dummy}
void remove (int index)

• Now let’s consider how to implement the remove(index) method:

1. Iterate a cursor from the dummy head towards the dummy tail until just before the node corresponding to index.

• Index 0 is just after the dummy head.

• Index size-1 is just before the dummy tail.

2. “Unlink” the cursor._next node from the chain.
void remove (int index)

• Now let’s consider how to implement the remove(index) method:

• As an example, let’s show how remove(1) works on the SinglyLinkedList to which we just added two elements.
void remove (int index)

1. Iterate until just before the node corresponding to index.

```java
Node cursor = _head;
for (int i = 0; i < index; i++) {
    cursor = cursor._next;
}
```

Let's assume for now that index is valid.
void remove (int index)

1. “Unlink” cursor._next from the chain.

cursor._next = cursor._next._next;
void remove (int index)

1. “Unlink” cursor._next from the chain.

    cursor._next = cursor._next._next;

Notice that *nothing points to* the Node we just unlinked; hence, the JVM garbage collector will eventually remove it...
void remove (int index)

Done! (You can pull it taut yourself.)

SinglyLinkedList

_node_head _node_tail

Node _node_next _node_data: 0

cursor

Node _node_next _node_data

dummy
dummy

dummy
If you followed the `add(o)` and `remove(index)` methods, then this one should be straightforward.

```java
Object get (int index) throws IndexOutOfBoundsException {
    // TODO: check whether index is valid

    Node cursor = _head._next;
    for (int i = 0; i < index; i++) {
        cursor = cursor._next;
    }
    return cursor._data;
}
```
int size ()

- Finally, we need to implement a simple size() method.

- Two possible strategies:
  1. Add another instance variable int _size to SinglyLinkedList, which we increment/decrement whenever add/remove is called.
  2. Don’t add another variable; instead, count the number of nodes between the head and the tail whenever size() is called.

- Each strategy has its advantages & disadvantages.
int size ()

• On the one hand:
  • Using a _size instance variable is much faster -- whenever size() is called, we can return the result immediately.
  • Without a _size variable, we have to iterate over the whole list -- slow!

• On the other hand:
  • Adding a new variable always creates code complexity. In a sense, _size is redundant -- the size is already implicitly encoded in the number of nodes in the list. Maintaining a “copy” of the size in a _size variable gives us more opportunities to mess up.
In a linked list, updating \_size is fairly easy.

In this case, it’s probably worth adding a \_size variable to reduce the time cost of the size() method, especially if we expect size() to be called frequently by the user.
SinglyLinkedList ADT

• Now that we know how to implement the four operations add, remove, get, and size, we can complete our SinglyLinkedList class.

• We now have two complete implementations of List:
  • ArrayList
  • LinkedList

• The “user” can use either implementation of List by calling the same methods.
List interface

```java
final List list = new LinkedList();

list.add("first");
list.add("second");
list.add("third");
System.out.println(list.get(1));  // "second"
list.remove(0);
System.out.println(list.get(1));  // "third"
```
List interface

```java
final List list = new ArrayList();

list.add("first");
list.add("second");
list.add("third");
System.out.println(list.get(1));  // "second"
list.remove(0);
System.out.println(list.get(1));  // "third"
```

The user can change from a LinkedList to an ArrayList by changing one line of code. None of the remaining code need change at all.
Confetti demo
Dummy nodes, revisited

- Let’s now go back to our SinglyLinkedList ADT and consider how to implement it without dummy nodes.

- In this case, the _head points to the first node, and _tail points to the last node.

- All nodes are “real” -- their _data pointers all point to data the user added.
Dummy nodes, revisited

- But what if the list is empty? What should `_head` and `_tail` point to?
• If the list is empty, let’s just set them both to null.

• Let’s now consider how to implement \texttt{add(o)} without the dummy nodes.

\begin{center}
\begin{tikzpicture}
\node (head) at (0,0) {\_head};
\node (tail) at (1,0) {\_tail};
\node (null1) at (-0.5,-1) {null};
\node (null2) at (0.5,-1) {null};
\draw (head) -- (null1);
\draw (tail) -- (null2);
\end{tikzpicture}
\end{center}

\texttt{SinglyLinkedList}
**add(o) without dummy nodes**

- What if `add(o)` is being called for the first time (i.e., on an empty list)?

- To which node should the new `Node` be linked?

```java
final Node node = new Node();
node._data = o;
```

```plaintext
... // ??
```
add(o) without dummy nodes

• What if add(o) is being called for the first time (i.e., on an empty list)?

• To which node should the new Node be linked?

  • None -- there is no other Node yet.

• We just set _head and _tail to the new Node.
add(o) without dummy nodes

- What if \texttt{add(o)} is being called for the second (or later) time?
- To which \texttt{Node} should the new \texttt{Node} be linked?
- The \textit{tail} -- now it actually exists.
**add(o) without dummy nodes**

- Without dummy nodes, the `add(o)` method must be implemented with an if-statement:

```java
final Node node = new Node();
node._data = o;
if (_head == null) {  // List is empty
    _head = _tail = node;
} else {  // List is not empty
    _tail._next = node;
    _tail = node;
}
```

- The if-statement makes the `add(o)` method more complicated than when using dummy nodes.
Similarly, when implementing `remove(index)` without dummy nodes, there must be an `if`-statement to distinguish two cases:

- Removing a node from a list of size 1.
- Removing a node from a list of size >1.

The dummy nodes require a bit more space (two “wasted” nodes), but they make the programming easier -- a worthwhile trade-off.
Doubly linked lists.
Problems with singly-linked lists

- Singly-linked list ADTs are useful because they:

  1. Grow automatically as the user adds more data.

  2. Do not suffer from the “contiguity” problem like ArrayLists do.

  3. Store only as many nodes as required (maybe +2 dummy nodes, but 2 nodes is not a big cost).
Problems with singly-linked lists

• However, singly-linked list ADT also suffer from a few drawbacks:

  1. Expensive to “jump” to particular element index.

  • Have to iterate from the head towards the tail.
Problems with singly-linked lists

• However, singly-linked list ADT also suffer from a few drawbacks:

1. Expensive to “jump” to particular element index.

• Have to iterate from the head towards the tail.

• “Iterating” to the desired element is fundamental to linked lists -- there’s no real way to avoid this.
Problems with singly-linked lists

2. There's no easy way to iterate backwards.
   - Each node only contains a \texttt{next} pointer.
Problems with singly-linked lists

2. There’s no easy way to iterate backwards.
   - Each node only contains a \texttt{next} pointer.
   - This can be remedied using a \textit{doubly-linked list}.
Doubly-linked lists

- In a doubly-linked list, each Node object has both a _next and a _prev pointer:

```java
class Node {
    Node _next, _prev;
    Object _data;
}
```
Doubly-linked lists

- A doubly-linked list containing 2 “real” nodes, and using 2 dummy nodes, would look like:
Doubly-linked lists

- With doubly-linked lists, it’s very fast to access nodes close to the tail, e.g.:

  ```java
  Object lastElement = _tail._prev._data;
  ```
Doubly-linked lists

• In particular, it is fast to remove an element from either end of the list.
• Just “unlink” the node _tail._prev.
• No need to “iterate through” the list (starting at the head) to get to the tail.
Linked list variants

• There exist other linked-list “variants” as well, e.g., circular lists.

• We will cover these later this week.
• In programming project 1, you must implement a doubly-linked list to implement the \texttt{List12} interface.

• It’s up to you whether you use dummy nodes or not. (I recommend you do because it simplifies the code.)

• Make sure to carefully adhere to the \texttt{List12 interface specification}.
As a specific requirement, your `addToFront()`, `addToBack()`, `removeFront()`, and `removeBack()` methods must operate “efficiently”.

Since you are implementing a doubly-linked list, there is no need to always “iterate through” the list starting at the head.

- If you’re implementing `addToFront()` or `removeFront()`, start at the head.
- If you’re implementing `addToBack()` or `removeBack()`, start at the tail.
One of the requirements of a class implementing the `List12` interface is the `iterator()` method.

But what is an `Iterator`?
Iterators.
Iterating over elements of a data structure.

- Many ADTs offer the user the ability to iterate over all of their elements in some “natural order”.
- With the simple List interface defined during lectures, this is already possible using the get(index) methods:

```java
final int size = linkedList.size();
for (int i = 0; i < size; i++) {
    System.out.println(linkedList.get(i));
}
```
Iterating over elements of a data structure.

- However, that approach will also be very slow:
  
  ```java
  LinkedList.get(0)
  ```

  ![Linked List Diagram](dummy to dummy)

Monday, July 9, 12
Iterating over elements of a data structure.

- However, that approach will also be very slow:
- `linkedList.get(0)`

![Diagram](dummy -> dummy)
Iterating over elements of a data structure.

- However, that approach will also be very slow:
  - `linkedList.get(0)`
  - `linkedList.get(1)`

![Diagram of a linked list with dummy nodes and elements]

Monday, July 9, 12
Iterating over elements of a data structure.

- However, that approach will also be very slow:
  - `linkedList.get(0)`
  - `linkedList.get(1)`

![Diagram of linked list with dummy nodes]

(dummy)
Iterating over elements of a data structure.

- However, that approach will also be very slow:

  ```java
  LinkedList.get(0)
  LinkedList.get(1)
  ```

[Diagram of linked list with dummy nodes]
Iterating over elements of a data structure.

- However, that approach will also be very slow:
  - `linkedList.get(0)`
  - `linkedList.get(1)`
  - `linkedList.get(2)`
Iterating over elements of a data structure.

- However, that approach will also be very slow:
  - `linkedList.get(0)`
  - `linkedList.get(1)`
  - `linkedList.get(2)`

![Diagram of LinkedList with dummy nodes]
Iterating over elements of a data structure.

- However, that approach will also be very slow:
  - `linkedList.get(0)`
  - `linkedList.get(1)`
  - `linkedList.get(2)`
Iterating over elements of a data structure.

• However, that approach will also be very slow:
  
  • `linkedList.get(0)`
  • `linkedList.get(1)`
  • `linkedList.get(2)`

We keep “re-iterating” -- starting from scratch back at the head. This is computationally wasteful. Why can’t we just pick up where we left off?
Iterators: performance benefits

- An “iterator” object helps us to avoid this wasted computation.
- An iterator is a “helper object” with which the user can iterate across all elements in a data structure.
- The iterator will “remember” where it left off.
Iterators: software design gain

• Iterators are also useful because they offer a uniform way of accessing all of a data structure’s elements.

• Even very different data structures -- e.g., graphs and lists -- can both support iterators.
interface Iterator

• In Java, the Iterator interface contains three method signatures:

```java
boolean hasNext();
Object next();
void remove();
```
How Iterators are used

• Here’s how the “user” would use an Iterator to print out every element in a linked list.

```java
final Iterator iterator = linkedList.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```
Here’s how the “user” would use an Iterator to print out every element in a linked list.

User calls `hasNext()` to “ask” the Iterator if there’s another element to fetch.

```
final Iterator iterator = linkedList.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

User calls `next()` to actually fetch the next element from the Iterator.
hasNext() and next()

• Note that the user is not “required” by the Iterator interface to call the hasNext() method.

• next() will still work correctly without previously calling hasNext().

• (But practically speaking, how else will the user know he/she is “done” iterating?)
The Iterator interface also gives the user the ability to remove elements from the linked list while iterating through them.
• E.g., consider a linked list containing 5 objects (o1, o2, o3, o4, o5).

```java
final Iterator iterator = linkedList.iterator();
iterator.next(); // returns o1
iterator.next(); // returns o2
iterator.next(); // returns o3
iterator.remove(); // removes o3
iterator.next(); // returns o4
iterator.next(); // returns o5
```

• If you subsequently called `linkedList.size()`, you would get 4 -- the linked list itself has changed.

• The `Iterator` object returned by `linkedList.iterator()` is “tied” to the underlying `LinkedList` object.
Restrictions on using an Iterator

• Before the user is “allowed” to call the \texttt{remove()} method, he/she \textit{must} first call the \texttt{next()} method.

• If he/she does not, the Iterator \textit{must} throw an \texttt{InvalidStateException}.
Restrictions on using an Iterator

- The `Iterator` interface also specifies that “the behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.”
Iterator interface

*Unspecified* means that the implementor is “absolved of any responsibility” for maintaining correct functionality in the Iterator if the user modifies the `DoublyLinkedList12` while he/she is iterating over it.

- The Iterator interface also specifies that “the behavior of an iterator is *unspecified* if the underlying collection is *modified* while the iteration is in progress in any way other than by calling this method.”

*Modifications* in the case of `DoublyLinkedList12` mean `addToFront()`, `removeFront()`, etc. -- anything that changes the contents of the list.
Interface as a “contract”

- An interface specification serves as a contract between user and implementor of the interface.

- The method signatures specify to the user what each method does, and how it is called (i.e., parameters).

- The comments describe to the implementor what each method must do and what values to return.
Interface as a “contract”

- The comments may also prescribe to the user various constraints on how the methods are called, e.g., “next () must be called before remove () .

- If the user does not adhere to these constraints, then he/she is in violation of contract.

- If the user violates the contract, then the implementor may:
  - Throw an exception (e.g., InvalidStateException).
  - Be “absolved of responsibility” to keep working correctly (“behavior is...unspecified”).
  - E.g., calls to next ()/remove ()/hasNext () may stop working correctly, and this is no longer the implementor’s fault.
Implementing Iterators

• The tricky thing about implementing an Iterator is that “you the implementor” do not get to decide when to traverse from one node to the next (e.g., `node = node._next`) -- the user decides that.

• The `Iterator` objects that your linked-list constructs (and returns in `iterator()`) must remember their position in the linked list -- and pick off where it left off when the user calls `next()` again.
class StudentDatabaseApplication {
    void doSomethingInteresting () {
        List12 list =
            new DoublyLinkedList12();

        ...

        list.add(new Student("Bob"));
        list.add(new Student("Lulu"));

        ...
    }
}
class StudentDatabaseApplication {
    void doSomethingInteresting () {
        List12 list =
            new DoublyLinkedList12();

        ...

        list.add(new Student("Bob"));
        list.add(new Student("Lulu"));

        ...
    }
}

class DoublyLinkedList12 implements List12 {
    {
        static class Node {
            ...
        }

        void add (Object o) { ...
        int size () { ...

        ...

        Iterator iterator () {
            ...
        }
    }
```java
class StudentDatabaseApplication {
    void doSomethingInteresting () {
        List12 list =
            new DoublyLinkedList12();

        ...

        list.add(new Student("Bob"));
        list.add(new Student("Lulu"));

        ...

        Iterator iter = list.iterator();
        while (iter.hasNext()) {
            Student s = (Student) iter.next();
            System.out.println(s._name);
        }
    }
}

class DoublyLinkedList12 implements List12 {
    {
        static class Node {
            ...
        }

        void add (Object o) { ... }
        int size () { ... }

        ...

        Iterator iterator () {
            ...
        }
    }
```
class StudentDatabaseApplication {
    void doSomethingInteresting () {
        List12 list =
            new DoublyLinkedList12();

        ...

        list.add(new Student(“Bob”));
        list.add(new Student(“Lulu”));

        ...

        Iterator iter = list.iterator();
        while (iter.hasNext()) {
            Student s = (Student) iter.next();
            System.out.println(s._name);
        }
    }
}

class DoublyLinkedList12 implements List12 {
    
    static class Node {
        ...
    }

    void add (Object o) { ...
    int size () { ...
    ...

    Iterator iterator () {
        return new Iterator();
    }
}
class StudentDatabaseApplication {
    void doSomethingInteresting () {
        List12 list =
            new DoublyLinkedList12();

        ...

        list.add(new Student("Bob"));
        list.add(new Student("Lulu"));

        ...

        Iterator iter = list.iterator();
        while (iter.hasNext()) {
            Student s = (Student) iter.next();
            System.out.println(s._name);
        }
    }
}

class DoublyLinkedList12 implements List12 {
    
    static class Node {
        ...
    }

    void add (Object o) { ... }
    int size () { ... }

    ...

    Iterator iterator () {
        return new DLL12Iterator();
    }
}
class StudentDatabaseApplication {
    void doSomethingInteresting () {
        List12 list =
            new DoublyLinkedList12();

        ...

        list.add(new Student("Bob"));
        list.add(new Student("Lulu"));

        ...

        Iterator iter = list.iterator();
        while (iter.hasNext()) {
            Student s = (Student) iter.next();
            System.out.println(s._name);
        }
    }
}

class DoublyLinkedList12 implements List12 {
    static class Node {
        ...
    }

    class DLL12Iterator implements Iterator {
        void add (Object o) { ... }  
        int size () { ... }  

        ...

        Iterator iterator () {
            return new DLL12Iterator();
        }
    }
}
class StudentDatabaseApplication {
    void doSomethingInteresting () {
        List12 list =
            new DoublyLinkedList12();

        ...

        list.add(new Student("Bob"));
        list.add(new Student("Lulu"));

        ...

        Iterator iter = list.iterator();
        while (iter.hasNext()) {
            Student s = (Student) iter.next();
            System.out.println(s._name);
        }
    }
}

class DoublyLinkedList12 implements List12 {
    static class Node {
        ...
    }

class DLL12Iterator implements Iterator {
    boolean hasNext() { ... }
    Object next () { ... }
    void remove () { ... }

    }

    void add (Object o) { ... }
    int size () { ... }

    ...

    Iterator iterator () {
        return new DLL12Iterator();
    }
}
class StudentDatabaseApplication {
    void doSomethingInteresting () {
        List12 list =
            new DoublyLinkedList12();

        ...

        list.add(new Student("Bob"));
        list.add(new Student("Lulu"));

        ...

        Iterator iter = list.iterator();
        while (iter.hasNext()) {
            Student s = (Student) iter.next();
            System.out.println(s._name);
        }
    }
}

class DoublyLinkedList12 implements List12 {
    static class Node {
        ...
    }

    class DLL12Iterator implements Iterator {
        boolean hasNext() { ... }
        Object next() { ... }
        void remove() { ... }

        Somewhere in next() will be code
        "cursor = cursor._next;"

        void add (Object o) { ... }
        int size() { ... }

        ...

        Iterator iterator() {
            return new DLL12Iterator();
        }
    }
}
class StudentDatabaseApplication {
    void doSomethingInteresting () {
        List12 list =
            new DoublyLinkedList12();

        ...

        list.add(new Student("Bob"));
        list.add(new Student("Lulu"));

        ...

        Iterator iter = list.iterator();
        while (iter.hasNext()) {
            Student s = (Student) iter.next();
            System.out.println(s._name);
        }
    }
}

class DoublyLinkedList12 implements List12 {
    static class Node {
        ...
    }

    class DLL12Iterator implements Iterator {
        boolean hasNext() { ... }
        Object next () { ... }
        void remove () { ... }

        Iterator iterator () {
            return new DLL12Iterator();
        }
    }

    void add (Object o) { ... }
    int size () { ... }

    ...

    But when this is called is determined by when the user calls "iter.next();".

    Somewhere in next() will be code
    "cursor = cursor._next;"

    void add (Object o) { ... }
    int size () { ... }

    ...

    Iterator iterator () {
        return new DLL12Iterator();
    }
}