CSE 12: Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Four
4 Aug 2011
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).

```
  _head
    ├── Node
    │   └── _next
    │       └── _data
    │           └── Object
    └── Node
          └── _next
              └── _data
                  └── Object

  _tail
    ├── Node
    │   └── _next
    │       └── _data
    │           └── Object
    └── null
```
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>01</th>
<th>02</th>
<th>03</th>
</tr>
</thead>
</table>

int _numElements: 3
```

```
LinkedList
_Node

<table>
<thead>
<tr>
<th>Node</th>
<th>Node</th>
<th>Node</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>02</td>
<td>03</td>
<td>null</td>
</tr>
</tbody>
</table>
```

Thursday, August 4, 2011
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.

```java
ArrayList

_underlyingStorage

| o1 | o2 | o3 |

_numElements: 3
```
Elements of an array

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<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>

- Arrays are always stored contiguously in memory (in one big chunk):
  - Addr of element i = BaseAddr + i * 4
- Easy to jump to a particular index using the [ ] operator.
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?
final Object thirdElement = _head._next._next._data;
Finding a particular node

- Alternatively, we could use a for-loop:

```java
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?

Node cursor = _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 (  ) {
  //...
}

Thursday, August 4, 2011
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) {

}

![Diagram](image_url)
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);
}

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?

```java
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](image)
Adding a new node

• The “iteration” code described above assumes that a linked list already exists.

• How is the “chain of nodes” actually constructed?
class SinglyLinkedList

• Before discussing how to implement the `add(o)` method, let’s first “concretify” the linked list class itself.

• Let’s create a `SinglyLinkedList` class that implements the simple `List` interface from Lecture Two...
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.
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.
   ```java
   final Node node = new Node();
   ```
void add (Object o)

2. Set its \_data field to equal \( o \).

node\._data = o;
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;
}
void add (Object o)

4. Insert the new node just after cursor.

node._next = cursor._next;

SinglyLinkedList
  _head _tail
  node._next = cursor._next;

Node
  _next _data: o
  node
  Node
    _next _data: o
    node
    Node
      _next _data
      dummy
      cursor
      Node
        _next _data
        dummy
void add (Object o)

4. Insert the new Node just after cursor.

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.

```java
node._data = o2;
```

Let’s change o to o2 (just in the slides, not in code) to distinguish from the previous Object 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.

```
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{SinglyLinkedList}
  \item \texttt{Node\_next\_data: o}
  \item \texttt{Node\_next\_data: o2}
\end{itemize}
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.

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.)
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;
}
```
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 \texttt{size} is fairly easy.

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

\begin{verbatim}
int size ()
\end{verbatim}
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.
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?
Dummy nodes, revisited

- If the list is empty, let’s just set them both to null.
- Let’s now consider how to implement `add(o)` without the dummy nodes.
**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;
...  // ??
```
**add(o) without dummy nodes**

- What if \( \text{add}(o) \) 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 add(o) is being called for the second (or later) time?
- To which Node should the new Node be linked?
- The 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.
SinglyLinkedList
without 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 `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:

```
DoublyLinkedList
  .head
  .tail

Node
  .data: o
  .prev
  .next
dummy

Node
  .data: o2
  .prev
  .next
dummy

Node
  .data
  .prev
  .next
dummy
```
Doubly-linked lists

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

```java
final 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 next 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.
• Your `DoublyLinkedList12` class must implement the `List12` interface.

• `List12` is a `subinterface` of `Iterable`.

  • In a similar way that a subclass extends its parent class, a subinterface extends its parent interface.

  • Therefore, your `DoublyLinkedList12` must also be `Iterable`.

• What defines an “`Iterable`” class?

  • It must implement the following method:
    
    ```java
    Iterator iterator();
    ```
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:

- `linkedList.get(0)`

```
    □ ┌─┬─┐ ┌─┬─┐ ┌─┬─┐ ┌─┬─┐ ┌─┬─┐ ┌─┬─┐ ┌─┬─┐
    □ └─┴─┘ └─┴─┘ └─┴─┘ └─┴─┘ └─┴─┘ └─┴─┘ └─┴─┘
        dummy dummy
```
Iterating over elements of a data structure.

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

![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)`

![Linked List Diagram](image)
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]
Iterating over elements of a data structure.

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

```
  □→□→□→□→□→□→□
  □ □ □ □ □ □ □
  dummy dummy
```
Iterating over elements of a data structure.

• However, that approach will also be very slow:

• `linkedList.get(0)`
• `linkedList.get(1)`
• `linkedList.get(2)`

```python
# Diagram of a linked list
```

dummy → □ → □ → □ → □ → □ → □ → dummy

```
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)`

```plaintext
dummy → □ → □ → □ → □ → □ → □ → □ → □
dummy
```
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.

• An “iterator” is one of the fundamental design patterns of software engineering. (More on this later.)
In Java, the `Iterator` interface contains three method signatures:

```java
boolean hasNext();
Object next();
void remove();
```
interface Iterator

• Your DoublyLinkedList12 class must implement the method:
  
  ```java
  Iterator iterator();
  ```

• I.e., it must return an Iterator via the `iterator()` method.

• But `Iterator` is itself an interface, not a class.

• This means that your `iterator()` method is allowed to return an object of *any* class that implements `Iterator`.

• It will be most convenient (and required in P1) to implement your `Iterator` as an inner-class, e.g., `DoublyLinkedList12Iterator`. 
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());
}
```
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());
}
```

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

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?)

Thursday, August 4, 2011
The Iterator interface also gives the user the ability to remove elements from the linked list while iterating through them.
remove()

• 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.
remove() and next()

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

• If he/she does not, the Iterator must throw an InvalidStateException.
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.