Computer Science 501
Data Structures & Algorithms
The College of Saint Rose
Fall 2013

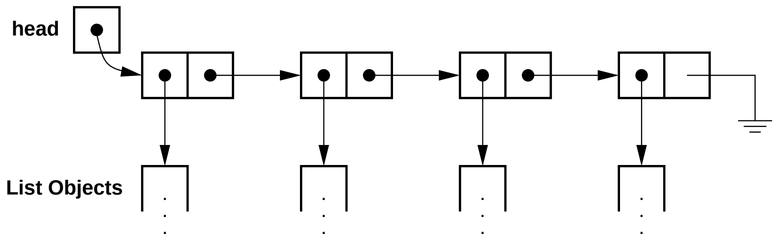
Topic Notes: Linked Structures

So far, all of our structures for holding collections of items have been very simple. We've used only arrays and `Vectors`. These have some pretty significant limitations. `Vectors` are resizeable, but it is an expensive operation. It's also expensive to add or remove objects from the start or the middle of the vector. We can do better.

We will begin our study of more advanced data structures with *lists*. These are structures whose elements are in a *linear* order.

Singly Linked Lists

These came up just briefly last time as a motivation for iterators. Most of you have seen the idea of a *linked list*:



This structure is made up of a pointer to the first list element and a collection of list elements.

The structure that makes up a list element has two fields:

1. `value`: the `Object` which is stored at that list element's position in the list.
2. `next`: a pointer to the next list element, or `null` for the last element.

So the data for a very basic linked structure could look like this:

```
class SimpleListNode<E> {

  protected E value;
  protected SimpleListNode<E> next;
}
```

```
public class SimpleLinkedList<E> {

  protected SimpleListNode<E> head;
}
```

As we saw with the `VectorIterator`, `public` is not specified in the class definition, since we aren't allowing regular users to create one of these, only a `SimpleLinkedList<E>`.

So if we want to create one of these, it's very easy. We just construct a `SimpleLinkedList<E>` and set its `head` to `null`.

```
  public SimpleLinkedList() {
    head = null;
  }
```

How about adding an element? This involves two steps:

1. construct a new list node for the element
2. insert the new list node into the list

Let's think about what this will mean. We add our first element, say a 1, we want this list to go from just an empty `head` reference, to a node pointed at by `head` which has the 1 as its `value` and `null` as its `next`.

Now, we add another element, say 2. We have two choices. We can add at the beginning or at the end.

Now, we add another element, 3. Now we have three choices. Beginning, middle, or end. In general, we can add at position 0, 1, or 2.

Construction of the new list node is easy, once we know what to set its `next` pointer to. Here's a constructor:

```
    public SimpleListNode(E value, SimpleListNode<E> next) {
        this.value = value;
        this.next = next;
    }
```

We'll see that we will need to be able to set and retrieve the value and the next pointer. We'll call the accessors `value()` and `next()`, and the mutators `setValue()` and `setNext()`.

We would like to allow additions to any place in our list, so we will develop a general `add` method that deals with all three of the cases described above.

We'll need to provide our `add` method with an index and an object:

```
public void add(int pos, E obj) {...}
```

Adding to an empty list is easy. We know it's empty if `head` is `null`. If so, then `pos` must be 0, and we just add the node:

```
        // are we adding to an empty list?
        if (head == null) {
            Assert.pre(pos==0, "Attempt to add at position " + pos + " in empty lis
            head = new SimpleListNode<E>(obj, null);
            return;
        }
```

So that was an easy case. How about adding at position 0? This is likely a common operation. It's also very simple. We just want to drop in a new list node, whose `next` is set to the old `head`, and assign `head` to the new list node.

```
        // are we adding at the front of a non-empty list?
        if (pos==0) {
            head = new SimpleListNode<E>(obj, head);
            return;
        }
```

It gets more complicated if we want to insert in the middle or at the end (`pos` != 0). We need to search for the item after which we want to insert, then do the insertion.

```
        // we are adding somewhere else, find entry after which we will
        // insert our item
        int i = 0;
        SimpleListNode<E> finger = head;
        while (i < pos-1) {
            i++;
            finger = finger.next();
            Assert.pre(finger != null, "Attempt to add at position " + pos + " in l
        }
        // finger points at the node after which we want to add
        // so create the new object with finger's next as its next
        // and set finger's next to the new node.
        // note that this also works for the case when we are adding
        // to the end
        finger.setNext(new SimpleListNode<E>(obj, finger.next()));
```

OK, now that we can build up our lists, let's consider a few accessors. First, `get`. Again, we'll allow users to `get` an element at any position.

```
    public E get(int pos) {

        SimpleListNode<E> finger = head;
        int i = 0;

        Assert.pre(head != null, "Attempt to get from an empty list");

        while (i < pos) {
            i++;
            finger = finger.next();
            Assert.pre(finger != null, "Attempt to get element " + pos + " from a '
        }
        return finger.value();
    }
```

We can write a `set` method almost identical to this, except that instead of returning the value at the desired position, we just set it and return the old value.

So now about `contains`? We need to search through looking for the element until we find it or find the end of the list.

The basic structure is the same as `get`. We have a “finger” tracking our progress through the list.

```
    public boolean contains(E obj) {

        // easy when the list is empty
        if (head == null) return false;

        // otherwise look for it
        SimpleListNode<E> finger = head;
        while (finger != null) {
            if (finger.value().equals(obj)) return true;
            finger = finger.next();
        }
        return false;
    }
```

Let’s do an easy one: `size()`.

```
    public int size() {
        SimpleListNode<E> finger = head;
        int count = 0;

        // count up the number of list nodes until we get a null next
        while (finger != null) {
            count++;
            finger = finger.next();
        }

        return count;
    }
```

That was easy, but quite inefficient. More on that later.

Now, let's consider a harder one: `remove()`. We can remove items by value or by index. We'll just implement by index.

There are a number of cases to consider:

1. remove the only item from a list
2. remove the first item in a list from a list with at least two elements
3. remove the last item in a list from a list with at least two elements
4. remove an item from the middle of a list from a list with at least two elements

```
    public E remove(int pos) {
```

First, we make sure we're not removing from an empty list with an `Assert`.

```
        Assert.pre(head != null, "Attempt to remove from an empty list");
```

Next, we can take care of the first item case.

```
        // check for removal of the first item in the list
        // this work for the one-element case, as head gets set to null
        if (pos == 0) {
            E retval = head.value();
            head = head.next();
            return retval;
        }
```

In other cases, we need to find the item we want to remove and adjust some pointers.

So we need to have our "finger" on the element *before* the one we want to remove, since that's the one whose `next` pointer will need to be adjusted.

```
        // remove an item at a non-first position
        SimpleListNode<E> finger = head;
        int count = 0;
        // find the item before the one we want to remove
        while (count < pos-1) {
            count++;
            finger = finger.next();
            Assert.pre(finger != null, "Attempt to remove element at index " + pos
        }
        // finger is pointing to item pos-1
        // make sure there is something at pos
        Assert.pre(finger.next() != null, "Attempt to remove element at index " + p
        E retval = finger.next().value();
        finger.setNext(finger.next().next());

        return retval;
```

Removing everything is very simple.

```
    public void clear() {
        head = null;
    }
```

What about all those list nodes? We still have references to them! Not to worry, Java's garbage collector will clean them up.

However, not all languages are garbage collected like Java. In C or C++, you need to be careful to `free` (in C) or `delete` (in C++) all of the objects you no longer need.

Let's consider the complexity of our operations.

- `add(0)` : $\Theta(1)$
- `add(i)` : $\Theta(i)$
- `add(n)` : $\Theta(n)$
- `get/set(0)` : $\Theta(1)$
- `get/set(i)` : $\Theta(i)$
- `get/set(n-1)` : $\Theta(n)$
- `remove(0)` : $\Theta(1)$
- `remove(i)` : $\Theta(i)$
- `remove(n-1)` : $\Theta(n)$
- `get` all values in sequence : $\Theta(n^2)$ (hey, we need an `Iterator`!)
- `size()` : $\Theta(n)$ (hey, we can do better if we remember this)

How do these compare to similar operations on `Vector`s?

- adding at the front is easier.
- adding at the end is harder.
- adding in the middle, well it depends where.
- the cost is consistent, though, since there is no reallocation and copying to grow the structure.
- removing at the front is easier.
- removing at the end is harder.
- removing in the middle is probably similar.
- getting/setting an arbitrary value is harder.

What about space usage?

- there are no empty slots like we have in `Vector`s
- but there's an extra reference for each object stored! That's $\Theta(n)$ space overhead.

We still have a couple of problems with this implementation that we'd like to address. First, the $\Theta(n^2)$ traversal is no good – we need an `Iterator`.

Remember, an iterator must remember some state about the collection it's visiting. With our `Vector` iterator example, we just needed to remember the index of the next item to be returned. Remembering the index doesn't help us here. We need to remember something about the internals of the list to make this work. The most useful thing to remember here is the list node – that "finger" we used in most of the methods we've looked at.

We'll implement our iterator as an extension of the structure package's `AbstractIterator` abstract class:

```
class SimpleListIterator<E> extends AbstractIterator<E> { ...
```

Again, it's not a public class, since no one except our `SimpleLinkedList` is allowed to construct one.

We need to have data to support the regular iterator operations, plus be able to reset the iterator, so we need to have our iterator remember the head of the list and the "finger":

```
    protected SimpleListNode<E> current;
    protected SimpleListNode<E> head;
```

and to construct one, we need to have the head of the list passed in:

```
    public SimpleListIterator(SimpleListNode<E> t) {
        head = t;
        reset();
    }
```

To reset, we just set the current to `head`.

```
    public void reset() {
        current = head;
    }
```

So the `current` pointer always points to the next node whose value has *not yet been returned*. From this, we can construct the remaining methods:

```
    public boolean hasNext() {
        return current != null;
    }

    public E next() {
        E temp = current.value();
        current = current.next();
        return temp;
    }

    public E get() {
        return current.value();
    }
```

And in the `SimpleLinkedList` class, we have a method to create one:

```
    public Iterator<E> iterator() {
        return new SimpleListIterator<E>(head);
    }
```

We also make our `SimpleLinkedList` implement `Iterable` so we can use it in “for each” loops.

This entire implementation:

See Example:
`/home/cs501/examples/SimpleLinkedList`

We can improve the efficiency of the `size()` method by maintaining an extra instance variable that tracks how many elements are in the list.

Which methods would need to change to do this?

- count needs to be initialized in the constructor
- increment count in add
- decrement count in remove
- reset count to 0 in clear
- simplify size to return count

This seems worthwhile. We've added some complexity to our methods but only added a single `int` to the size of the structure. The biggest disadvantage of adding a count is that we could forget to update it in some circumstance, leading to an inconsistent structure. For example, there are three different cases in the `add` method, and we need to make sure we increment the count in each.

This is exactly what we find in the structure package's `SinglyLinkedList` implementation.

See Structure Source:
`/home/cs501/src/structure5/SinglyLinkedList.java`

Maintaining a `tail` pointer

There's another enhancement we can consider. Adding things to the end of the list seems like something that will happen pretty often, and it's an $\Theta(n)$ operation in our implementation.

Well, the only reason it's $\Theta(n)$ is because we need to follow all the links from the head to find the last element so we can add it.

We can fix that by maintaining another reference to the tail of the list.

- An empty list has `head`=`tail`=`null`, a list w/one element has `head` = `tail` = a reference to that element, while in all other cases, `head` != `tail`.
- Then, an `add` operation specifying the end becomes straightforward and $\Theta(1)$.
- It also would let us `get` or `set` the last element in $\Theta(1)$ time.
- But a `remove` from the end is still $\Theta(n)$. We need `tail`'s predecessor to be able to remove the last item, and we have to search all the way from the beginning to find it.

Still, this seems like a worthwhile enhancement. We add just one extra reference and that's that.

It adds coding complexity to `add` and `remove` methods since we must worry about resetting the `tail` field. Even adding at the beginning may have to reset `tail` field (why?).

Circular Lists

Can we simplify things further without changing any big-O behavior by noticing that tail node of list has a `next` field that is always "wasted" – it is always equal to `null`?

If we use that field to point to the beginning of the list, then we don't need a separate `head` field?

This is a *circularly linked list*. The `head` is always found as `tail.next()`!

This is the `CircularList` in the structure package.

See Structure Source:
`/home/cs501/src/structure5/CircularList.java`

- With this implementation, `add` at the end is now $\Theta(1)$.
- But now it takes one extra dereference (i.e., following a reference) to get to `head`.
- Only `contains`, `remove`, and `removeLast` are still $\Theta(n)$.
- `contains` and `remove` involve searches and seem likely always to be $\Theta(n)$ (unless we attempt to keep list in order and do binary search - which has its own problems - we'll consider this later..).
- However, why can't we make `removeLast` $\Theta(1)$? The problem is that we need to know the predecessor in order to delete an element from the list.

Doubly-linked lists

In our implementations so far, references only go from the front to the back of the list. Why not put them in the other direction instead? Well we could, but then it would be harder to delete from the front.

So... we can put references in both directions.

This is a more significant change: our list nodes now need to change. We need an extra field to hold a reference to the previous node, but the space overhead remains $\Theta(n)$.

```
class DoublyLinkedListElement<E> {

  protected E value;
  protected DoublyLinkedListElement<E> next;
  protected DoublyLinkedListElement<E> prev;
}
```

With this defined, it is now easy to define a doubly-linked list. This time we'll keep track of both the first (`head`) and last (`tail`) elements of the list so we can get to `tail` quickly.

This is the `DoublyLinkedList` in the structure package.

See Structure Source:
`/home/cs501/src/structure5/DoublyLinkedList.java`

Note that the constructor for `DoublyLinkedListElement`s automatically sets back pointers to maintain consistency.

`removeLast` is now $\Theta(1)$, but the tradeoff is that now all addition and removal operations must set one extra pointer in the list node. We must also worry about maintaining both the `head` and `tail` of the list, with complicated cases arising when adding and removing from a 0-, 1-, or 2-element list.

Design of List Classes

We now have a number of ways to implement linked list structures. To some extent, these are all interchangeable functionally. We can add, retrieve, remove, search, though there are time and space tradeoffs involved.

We would like to be able to use them interchangeably. This is what Java interfaces and abstract classes allow us to do.

We start by defining an interface that we'll use for a variety of implementations of lists:

- Accessors: `isEmpty()`, `size()`, `get()`, `contains()`, `iterator()`
- Mutators: `add()`, `set()`, `remove()`, `clear()`

See Structure Source:
`/home/cs501/src/structure5/List.java`

First notice that it extends `Structure`. This means a `List` requires the basic operations we expect on any of our structures.

See Structure Source:
`/home/cs501/src/structure5/Structure.java`

The text has a simple example of reading in successive lines from a text and adding each line to the end of a list if it doesn't duplicate an element already in the list. This is easily handled with the operations provided.

`AbstractList`s

The `List` interface requires a lot of methods, many of which will be the same in all of the implementations. So the structure package defines an abstract class for this:

See Structure Source:
`/home/cs501/src/structure5/AbstractList.java`

Then each of our actual list implementations `extends AbstractList`, and needs only to fill in the methods not already provided.

`Vector` as a `List`

Given this, we can imagine another implementation of the `List` interface.

`Vector` already provides all of the methods required by the `List` interface. In the structure package, `Vector extends AbstractList`. In the Java API, this explains the name `ArrayList`.

So we can use a `Vector` as a “list” as well. Some of the operations are more expensive, but anywhere we want a `List`, we can use a `Vector`.

This leaves 4 classes that implement the `List` interface (all by extending `AbstractList`) in the structure package:

- `SinglyLinkedList`
- `CircularList`
- `DoublyLinkedList`
- `Vector`

You should understand how each of these structures work, know how to use them correctly, should be able to develop the internals of any of the methods in these classes, and should understand the time and space complexities of the implementations.