SIENAcollege
Computer Science

Computer Science 210
Data Structures
Siena College
Fall 2020

Topic Notes: Linked Structures

Recursive Data Structures

We have seen that constructs such as arrays and `ArrayLists` allow us to group together collections of elements using a single name. These work very well in many situations, but are not always going to be the most efficient option. Consider a situation where you often need to add and remove elements from near the start of a large `ArrayList`. Each one of those operations, for an `ArrayList` of n elements, is an $O(n)$ operation. If that is something we will need to do frequently, we will want to do better.

Recursion affords us another mechanism to store collections of elements.

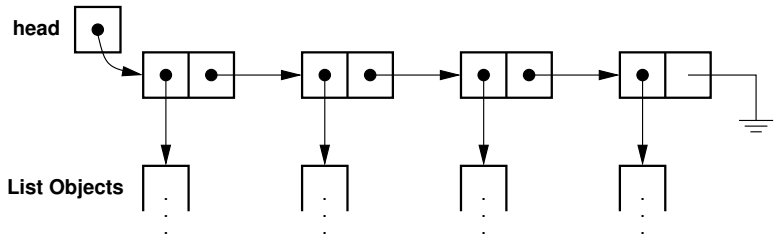
General Purpose Recursive Structures

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

The *singly linked list* structure is built using a collection of *list nodes*. Each list node is responsible for keeping track of one *list element*. The list elements are the values we actually want to store and retrieve in our data structure.



We will study this idea by looking at a simple implementation of generic singly linked lists.

`https://github.com/SienaCSISDataStructuresJDT/sll-example`

The list implementation is in `SimpleLinkedList.java`, and a `main` method we will use to learn how this list works is in `NotesExample.java`. As we trace through that `main` method, we will look at the details of the implementation and construct a series of memory diagrams (drawn on the board in class).

Before we start tracing through our example, we will look at the two main object types that represent a list node and the list itself.

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

1. `value`: the element (whose type here is specified by a type parameter) to be stored in the list.
2. `next`: a pointer to the next list node, which will be `null` for the node containing the last element.

```
class SimpleListNode<E> {

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

And the list structure itself is no more than a reference to the list node that contains the first list element.

```
public class SimpleLinkedList<E> {

  protected SimpleListNode<E> head;
}
```

We define `SimpleListNode` in the same Java file as `SimpleLinkedList`, but `public` is not specified in the class header for `SimpleListNode`, since we aren't allowing regular users to create instances of these. Only a `SimpleLinkedList<E>` can create a `SimpleListNode`. Note that the Java file's name needs to match the `public` class it defines.

The user who wishes to use a `SimpleLinkedList` would start by constructing an empty list, like we do at the start of our `main` method in `NotesExample`:

```
    SimpleLinkedList<Double> dlist = new SimpleLinkedList<Double>();
```

As with any class, this results in a call to a constructor. All the constructor needs to do in this case is to set the `head` instance variable to `null`.

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

Next, we'll work on adding elements. Adding an element 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, in our example, a 17.1.

```
dlist.add(17.1);
```

We want this list to go from just an empty `head` reference, to a node pointed at by `head` which has a reference to the 17.1 as its `value` and `null` as its `next`.

Our `SimpleListNode` has two `add` methods. The one-parameter version just calls that two-parameter version with a position parameter of 0. So that's the method we'll look at.

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

The method first does a little error checking. Turns out any add at position 0 is the same in our implementation, regardless of whether the list is empty. The code below takes care of this caee.

```
if (pos == 0) {
   head = new SimpleListNode<E>(obj, head);
   return;
}
```

We replace the list's `head` reference (which is `null` in this case) with a reference to a new list node. The constructor for `SimpleListNode` is straightforward enough, just storing the two parameters (the value and the `next` reference) in the node's instance variables:

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

In this case, the node's `next` instance variable is set to `null`, as the previous value of the list's `head` reference.

Now, we add another element at the start of the list.

```
dlist.add(23.0);
```

The same code is used for this case, but now the list's `head` reference starts as a reference to the list node responsible for the 17.1. That reference becomes the `next` of the new node we're creating for 23.0, which itself becomes the new `head` of the entire list.

The situation becomes a bit more interesting when we want to add a position other than 0.

```
dlist.add(1, -3.5);
```

To achieve this, we need to find the list node after which a new node will need to be inserted to store the new element, and have that new node's `next` reference point to the rest of the existing list beyond that point.

```
// 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();
    if (finger == null) {
        throw new IndexOutOfBoundsException("Attempt to add at position " + pos
    }
}
// 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()));
```

This makes use of trivial accessor and mutator methods that allow us to retrieve and modify a list node's `next` field.

The loop here follows `next` references, counting how many nodes we have "skipped over". Once we have skipped `pos-1` nodes, our `finger` reference points at the node after which we need to insert. Note that if the `finger` ever becomes `null`, it means the `pos` at which we were asked to insert a value is too large given the current contents of the list.

Once we know where we are inserting, the last line of code in the method accomplishes exactly what we need:

1. A new list node is created to store the element we are adding,
2. that new list node's `next` reference is set to the `next` reference of the "finger" node, and
3. the "finger" node's `next` reference is set to the new list node we just created.

Turns out this same code works when we add at the end of the list:

```
dlist.add(3, 1.1);
```

The only difference is that `finger`'s `next` reference will be `null`, resulting in the new node's `next` reference being `null`, which is exactly what we want for the new last node in our list.

Now that we can construct and populate `SimpleLinkedList`s, we consider the `get` method. Like with corresponding method of `Vector`/`ArrayList`, it can retrieve the element at any position in the list.

Our method call

```
double val = dlist.get(2);
```

results in a call to the method:

```
public E get(int pos) {

    // negative positions not allowed
    if (pos < 0) {
        throw new IndexOutOfBoundsException("Attempt to get from a negative pos
    }

    if (head == null) {
        throw new IndexOutOfBoundsException("Attempt to get from an empty list'
    }

    SimpleListNode<E> finger = head;
    int i = 0;
    while (i < pos) {
        i++;
        finger = finger.next();
        if (finger == null) {
            throw new IndexOutOfBoundsException("Attempt to get element " + pos
        }
    }
    return finger.value();
}
```

Other than some error checking, the code is strikingly similar to add. The only difference here is that once we've followed the correct number of `next` references to find the list node that contains the element we are trying to retrieve, we simply access that value (with the `value` accessor method of `SimpleListNode`) and return it.

The next operation used by our `NotesExample` `main` method is `set`:

```
dlist.set(1, -7.0);
```

The code for the `set` method (not reproduced here) is almost identical to `get`. Instead of returning the value at the desired position, we just set it to the new value and return the old value.

We next have a couple of calls to `contains`

```
boolean hasVal = dlist.contains(23.0);
System.out.println("Has a 23.0? " + hasVal);
hasVal = dlist.contains(0.5);
System.out.println("Has a 0.5? " + hasVal);
```

We again search through list nodes, but instead of counting how many we have skipped over to find particular one, we check the value in each node until we either find the one we're searching for (in which case the method returns `true`), or reach the end of the list, indicated by the "finger" becoming `null` (in which case the method returns `false`).

The basic structure is the same as `get` and `set`.

```
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;
}
```

Next up, we call `size`, which is also pretty straightforward. Here, we traverse the list nodes, counting how many we see. When the `finger` becomes `null`, we are out of list nodes to count, and return that count.

```
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 code is easy enough, but it is quite inefficient ($O(n)$ for an n-element list). More on that later.

Now, let's consider a harder one: `remove()`. In general, we will want to be able to remove items by value or by index. We'll just implement by index.

There are a number of cases that we need to make sure are considered, though some of the more specific might end up being taken care of by general cases:

1. remove the only item from a list (in which case it also was the first and the last)
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

Our running example in `NotesExample` tests all of these cases.

Let's see how the `remove` implementation handles these.

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

After some error checks, we can take care of the "first item" case, where it turns out not to matter if it is also the only item.

```
if (pos == 0) {
    E retval = head.value();
    head = head.next();
    return retval;
}
```

The list's `head` reference gets replaced with the `next` reference of the first node (which contains the element we wish to remove). Simple enough.

In other cases, we need to find the item we want to remove and adjust some references to "bypass" the node that contains the element we're removing.

The key idea here is that 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();
```

```
            if (finger == null) {
                throw new IndexOutOfBoundsException("Attempt to remove element at i
            }
        }
        // finger is pointing to item pos-1
        // make sure there is something at pos
        if (finger.next() == null) {
            throw new IndexOutOfBoundsException("Attempt to remove element at index
        }
        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)` : $O(1)$
- `add(i)` : $O(i)$
- `add(n)` : $O(n)$
- `get/set(0)` : $O(1)$
- `get/set(i)` : $O(i)$
- `get/set(n-1)` : $O(n)$
- `remove(0)` : $O(1)$
- `remove(i)` : $O(i)$
- `remove(n-1)` : $O(n)$
- `get` all values in sequence : $O(n^2)$ (we need to do better than this!)
- `size()` : $O(n)$ (hey, we can do better if we remember this)

How do these compare to similar operations on `Vectors`/`ArrayLists`?

- 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 `Vectors`/`ArrayLists`
- but there's an extra reference for each object stored! That's $O(n)$ space overhead.

Introducing an Iterator

We still have a couple of problems with this implementation that we'd like to address. First, the $O(n^2)$ traversal is no good – we need a way to remember where we left off on the traversal of the list from the previous `get` to more efficiently do the next `get`.

This is accomplished through a sort of companion class to a data structure called an *iterator*.

An iterator must remember some state about the collection it's visiting. We can think of it as putting a bookmark in the list reminding us where we left off so we can pick up there the next time we retrieve a value. As with many data structures, to create an iterator over a linked list, we will need to know 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.

Java provides an interface for this purpose in `java.util.Iterator`, and that is what we use in our linked list's iterator implementation:

```
class SimpleListIterator<E> implements java.util.Iterator<E> { ...
```

As with other interfaces we have seen, like `Comparable` and `Comparator`, `Iterator` requires specific methods to be provided by any class that `implements` it. The `Iterator` requires three methods: `hasNext`, `next`, and `remove`. The first two might sound familiar from your use of the `Scanner` class, and they play the same roles here: `hasNext` tells you if there is

more data in the structure that has not yet been returned by the iterator, and `next` returns the next such value. We will not be concerned about the `remove` method, but you will see it is included in our implementation.

A first thing to notice is that this is not a `public` class, meaning that it can only be constructed here inside the classes of our `SimpleLinkedList`.

The iterator needs an instance variable to be our "bookmark". We will see that at all times, it either points to the list node that contains the next item to be returned by the iterator, or is `null`, indicating that there are no more items to return.

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

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

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

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;
}
```

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 yet another interface called `Iterable`. This tells Java that the class provides an `iterator` method. This allows us to traverse the contents of our `SimpleLinkedLists` using enhanced `for` loops.

So if we have a list:

```
SimpleLinkedList<Integer> a = new SimpleLinkedList<Integer>();
```

which is then populated with n values, the code to traverse this list without an iterator would look like this (of course the `println` could be replaced with whatever code we wish to use to process each element of the list):

```
for (int i = 0; i < a.size(); i++) {
    System.out.println(a.get(i));
}
```

which we saw results in a total cost of $O(n^2)$, would we replaced with a loop based on the iterator:

```
for (Iterator<Integer> iter=a.iterator(); iter.hasNext(); ) {
    System.out.println(iter.next());
}
```

or formulated as a `while` loop:

```
Iterator<Integer> iter = a.iterator()
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

or with an enhanced `for` loop, which uses the iterator implicitly:

```
for (int x: a) {
  System.out.println(x);
}
```

In all of these cases, the fact that the iterator remembers the position within the list means the entire traversal is now done in $O(n)$. Much better!

Maintaining an element count

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 work to our methods, but it's all $O(1)$, and we 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.

Maintaining a `tail` pointer

There's another enhancement we can consider. Adding elements to the end of the list is an operation we might like to make as efficient as possible. It's an $O(n)$ operation in this implementation 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. Some things to note about such an implementation:

- An empty list has both `head` and `tail` as a `null` reference , a list with one element has both `head` and `tail` as a reference to that one element's list node, while in all other cases, `head` and `tail` refer to different list nodes.
- Our `add` operation in the final position is straightforward (the new node is stored at the `tail` node's `nextElement` and the `tail` is updated to refer to the new node. This is $O(1)$.
- The `tail` reference makes a `get` or `set` of the last element an $O(1)$ time operation.
- But a `remove` from the end is still $O(n)$. 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 have some efficiency benefits.

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