CS 310: Array-y vs Linky Lists

Chris Kauffman

Week 4-2
Logistics

Reading

- Weiss Ch 17 Linked Lists
- Pay attention to notion of Iterators

Goals

- Linked List implementation
- Iterators

Practice problems

- Write your own doubly linked list with List operations add(x) add(i,x) remove(i) get(i) set(i,x)
- 17.9: Implement LinkedList.removeAll(T x), get rid of all equals(x)
- 17.20: Augment ListIterator to have add(T x) for positional adding
Quick Review Questions

Method \texttt{xFer(double x[])} takes an array of size \( N \) and completes after \( 10 + 4N + \frac{1}{2}N^2 - \log_2(N) \) steps.

- What is the big-O runtime complexity of \texttt{xfer()}?

For an \texttt{ArrayList} with \( N \) elements...

- What is the runtime complexity of \texttt{get(i)}?
- What is the runtime complexity of \texttt{remove(i)}?
- What is the memory overhead of \texttt{remove(i)}?
- What is the runtime complexity of \texttt{add(x)}?
- What is the memory overhead of \texttt{add(x)}?

Stacks and Queues

- What are the basic operations of a Stack?
- Of a Queue?

- Describe two fundamental ways to implement a Stacks/Queues. Mention what fields are required in each
The Notion of a List

```java
package java.util;

public interface List<E> extends Collection<E>

An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Can implement this two ways

- Based on contiguous memory (ArrayList)
- Based on linked nodes of memory (LinkedList)

Choice leads to consequences for operations complexity
Links Aren’t Just for Queues

- Recall ListQueue - built on Node
- Slight tweaks enable general List data structure with get(i)/set(i,x)/add(x) capabilities
- Node fields head (front) and tail (back)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Big-O Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(x)</td>
<td>Add x to the end</td>
<td></td>
</tr>
<tr>
<td>addFront(x)</td>
<td>Add x at the 0th index</td>
<td></td>
</tr>
<tr>
<td>get(i)</td>
<td>Retrieve element i</td>
<td></td>
</tr>
<tr>
<td>set(i,x)</td>
<td>Set element i to x</td>
<td></td>
</tr>
<tr>
<td>insert(i,x)</td>
<td>Add x at position i</td>
<td></td>
</tr>
<tr>
<td>remove(i)</td>
<td>Remove element i</td>
<td></td>
</tr>
</tbody>
</table>

- How are these operations implemented? Pseudocode or Java
- What are the runtime complexities of each?
- How are the operation complexities of LinkedList different from ArrayList?
Exercise: Complexity of Access Forward/Backward

Print elements front to back

class ArrayList/LinkedList{
    public void printAll(){...}
}

- ArrayList implementation
- SinglyLinkedList implementation
- Make both $O(N)$

Print elements back to front

class ArrayList/LinkedList{
    public void printAllReverse(){...}
}

- ArrayList implementation
- SinglyLinkedList implementation (!)
- Can both be $O(N)$?
Double Your Fun

- Singly linked nodes: only next field
  - Node n = new Node(data, next);
- Doubly linked also has previous field as well
  - Node n = new Node(data, previous, next);

How about printAllReverse() now

1 Source: David H. Hovemeyer’s notes
To Header or Not to Header

- Some linked list implementations use auxiliary 'header' and 'tailer' nodes
- Contain no data, there to simplify code, no null cases
- Draw pictures to understand these
- Weiss uses header/tailer nodes
- Consider code below for add(x) to the back of a linked list

No Header

```java
public void add(T x) {
    if (empty()) {
        head =
            new Node(x, null, null);
        tail = head;
    } else {
        tail.next =
            new Node(x, tail, null);
        tail = tail.next;
    }
}
```

With Header/Tailer

```java
public void add(T x) {
    Node newLast =
        new Node(x, tail.previous, tail);
    tail.previous.next = newLast;
    tail.previous = newLast;
}
```
Exercise: Doubly Linked add(i,x) / insert(i,x)

- Insert x at position i in list
- List is doubly linked, headed
- Both head and tail point to ever-present dummy nodes

```java
class LinkedList<T>{
    static class Node<X>{
        X data; Node<X> next, prev;
        public Node(X d, Node<X> p, Node<X> n){
            this.data=d; this.next=n; this.prev=p;
        }
    }
    private Node<T> headNode, tailNode; // Always present, contains no data
    private int size; // Number of elements in the list
    // Constructor to create an empty list
    public LinkedList( ){
        this.headNode = new Node<T>(null,null,null);
        this.tailNode = new Node<T>(null,headNode,null);
        this.headNode.next = tailNode;
        this.size = 0;
    }
    // Add x at position i, 0 <= i <= size()
    public void add(int i, T x){
        // WRITE ME, don’t use iterators
    }
}
```
A Problem

Recall

- ArrayList.get(i): $O(1)$
- LinkedList.get(i): $O(n)$

Trouble

```java
List<Integer> l = ...;
int sum = 0;
for (int i = 0; i < l.size(); i++) {
    sum += l.get(i);
}
```

What is the complexity of the loop?
Peeking Inside with Iterators

Arrays are simple

- get/set anything in $O(1)$
- add/remove is obvious
- Very clear how data is laid out

Just about every other data structure is less straight-forward

- Getting/setting nontrivial
- Must preserve some internal structure - control access
- Element-by-element traversal needs to be done carefully

These qualities give rise to **iterators**

- A view of a data structure at a specific position
- Allow movement through structure, access to elements
Iterators

Give access to a position in a Collection

ListIterator Interface

```java
public interface ListIterator<T>{
    // Can the iterator be moved?
    public boolean hasNext();
    public boolean hasPrevious();

    // Move the iterator
    public T next();
    public T previous();

    // Modify the container
    public void add(T x);
    public void remove();
}
```

Semantics

- Collection creates/provides an iterator
- Use `next()`/`previous()` to move
- `next()`/`previous()` returns element "moved over"
- `remove()` removes element that was returned from last `next()`/`previous()`
- Illegal to remove w/o first calling `next()`/`previous()`
- `add(x)` before whatever `next()` would return
Iterators are In Between Elements

List Iterators have slightly complex semantics: *between* list elements

**Next/Previous**

```java
LL l = new LL([A, B, C, D])
itr = l.iterator()
```

```
[ A B C D ]
```

```
itr.next() [ A B C D ]
A ^
```

```
itr.next() [ A B C D ]
B ^
```

```
itr.previous() [ A B C D ]
B ^
```

```
itr.previous() [ A B C D ]
A ^
```

```
itr.previous() --> NoSuchElementException
```

**Add/Remove**

```java
LL l = new LL([A, B, C, D])
itr = l.iterator()
```

```
[ A B C D ]
```

```
itr.add(X) [ X A B C D ]
```

```
itr.next() [ X A B C D ]
A ^
```

```
itr.next() [ X A B C D ]
B ^
```

```
itr.remove() [ X A C D ]
```

```
itr.remove() --> ERROR
```

```
itr.previous() [ X A C D ]
A ^
```

```
itr.add(Y) [ X Y A C D ]
```

```
```
Common Patterns using Iterators / Iterable

public static void removeEqual(List<T> l, T x){
    Iterator<T> it=l.iterator(); // get an iterator
    while(it.hasNext()){// while it can move forward
       T y = it.next(); // grab an element
       if(x.equals(y)){ // if it should be removed
           it.remove(); // ask the iterator to remove it
    }
}
}

public static int countEqual(List<T> l, T x){
    int count=0;
    for(T y : l){ // iterator/hasNext() done
        if(x.equals(y)){ // automatically in a for-each
            count++;
        }
    }
    return count;
}
Exercise: Draw the Final List

```
LL l = new LL([A, B, C, D])
iter = l.iterator()
iter.next()
iter.next()
iter.next()
iter.add("X")
iter.previous()
iter.add("Y")
iter.next()
iter.next()
iter.next()
iter.remove()
iter.next()
iter.next()
iter.add("W")
iter.previous()
iter.remove()
```

Draw the final list as doubly linked nodes. Include the position of iter.
What would you do?

// l = [A, B, C, D];
it1 = l.iterator().next().next();
it2 = l.iterator().next();
// l = [ A B C D ]
// 1
// 2
it1.remove();
it2.next(); // ??

Where should it2 be now?

- Determine viable possibilities
- Explore what Java actually does
ConcurrentModificationException

Java’s premise: Danger!

```java
it1 = l.iterator();
it2 = l.iterator();
it1.remove();
it2.next(); // Error
```

Doesn’t try to coordinate multiple iterators changing a collection
- Multiple iterators easy for reading/viewing
- Very difficult to coordinate modifications
- A generally recurring pattern in CS: multiple simultaneous actors are a pain in the @$$
- Detect multiple concurrent modifications using modCount field, see Weiss
Nested Namespaces

- Java restriction: 1 public class per file
- Sometimes this is bothersome
- Group of related classes
  - One external public class
  - Uses some internal classes
  - Internal classes not for public consumption
  - List, Node, Iterator
- Endow internal classes with access to containing class
- Makes programming more convenient
Nested and Inner Classes

Straight from the official docs

```java
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

Both nested/inner classes

- Put multiple classes in a single file
- Give access to namespace of OuterClass
- Have access to private methods of OuterClass

\(^2\) Courtesy of Oracle
Some Textbook Classes

**Singly linked node, list, iterator (17.1-2)**

All classes separate

weiss.nonstandard.LinkedList
weiss.nonstandard.ListNode
weiss.nonstandard.LinkedListIterator

**Interfaces, ListIterator extends Iterator**

Implemented by ArrayList/LinkedList

weiss.util.List
weiss.util.Iterator
weiss.util.ListIterator

**Doubly Linked list (17.3-5)**

weiss.util.LinkedList
weiss.util.LinkedList.LinkedListIterator (inner, non-static)
weiss.util.LinkedList.Node (nested, static)
Wrap-up

Lists

- ArrayList and LinkedList
- Both allow sequential access of element $i$
- Both often provide iterators (see Weiss’s implementation of ArrayLists with Iterators)
- Have markedly different data layout, consequences for performance

Practice problems for home

- Write your own doubly linked list with List operations add($x$) add($i,x$) remove($i$) get($i$) set($i,x$)
- 17.9: Implement LinkedList.removeAll($T x$), get rid of all equals($x$)
- 17.20: Augment ListIterator to have add($T x$) for positional adding