CS 310: Iterator

Connor Baker

February 7, 2019

List Implementation O(n) Summary

Implementation	get/set	add/del at end	add/del start	add/del mid	search	can grow?
Static Array	1	1	N	N	N	no
Dynamic Array	1	**1***	N	N	N	no
Singly-Linked	N	1, N	1	N	N	yes
Doubly-Linked	N	1	1	N	N	yes

*Amortized analysis

Review: Pytania Questions

- Dynamic arrays
- Linked lists

Problem: List Iteration

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

- What is the complexity of this loop?
  - Array-based list
  - Linked list-based list
- Recall that
  - `ArrayList.get(i)` : $O(1)$
  - `LinkedList.get(i)` : $O(n)$

`Iterators`

- Data structures other than arrays have the following properties:
  - Nontrivial get/set
  - Must preserve some internal structure – that means controlling access
  - Element by element access takes time
- Those qualities give rise to `Iterators`
  - A view of a data structure
  - Supports efficient iterations on the collection

Java Iterators

- `Interface Iterator<E>`
  - Is in `java.util`
  - http://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
- Methods
  - **Required:** `hasNext()` and `next()`
  - Default: `remove()` and `forEachRemaining()`
- Any class that implements the `List<E>` interface needs to implement these methods
  - It follows then that `ArrayList`, `Vector`, and `Set` implement these

Textbook Sample

```
package weiss.util;

/**
 * Iterator interface
 */
 public interface Iterator<AnyType> extends java.util.Iterator<AnyType> {
    /**
     * Tests if there are items not yet iterated over.
     */
    boolean hasNext();

    /**
     * Obtains the next (as yet unseen) item in the collection.
     */
    AnyType next();

    /**
     * Remove the last item returned by next.
     * Can only be called once after next.
     */
    void remove();
    }
 }
```

Java List Iterators

- Sub-interface of `Iterator`: give access to a position in any `Collection`
- Different and more complex semantic: put a cursor **between** list items
- Operations
  - Most important methods: `next()` and `hasNext()`
  - Optional for general iterator: `remove()`, `previous()`, `hasPrevious()`, and `add()`

Common Iterator Operations

- Use `hasNext()` or `hasPrevious()` to determine whether the end has been reached
- Use `next()` or `previous()` to move the position of the cursor (you can modify both to return the element that the cursor moved over)

```
List l = new List([A, B, C, D])
ListIterator itr = l.iterator() // [^A B C D]
itr.next() // [ A^B C D]
// A
itr.hasNext() // [ A^B C D]
// True
itr.next() // [ A B^C D]
// B
itr.previous() // [ A^B C D]
// B
itr.previous() // [^A B C D]
// A
itr.previous() // [^A B C D]
// NoSuchElementException
```

`ListIterator` Operations

- `add(x)` add `x` before whatever `next()` would return
  - Insert before the implicit cursor
- `remove()` removes the element which was returned from the last `next()` or `previous()` call
  - It is **illegal** to remove an item without first calling `next()` or `previous()`

```
List l = new List([A, B, C, D])
ListIterator itr = l.iterator() // [^A B C D]
itr.add(X) // [X^A B C D]
itr.next() // [X A^B C D]
// A
itr.remove() // [ X^B C D]
itr.remove() // [ X^B C D]
// IllegalStateException
itr.previous() // [^X B C D]
// X
itr.add(Y) // [Y^X B C D]
```

Exercise

- Given a list and a sequence of operations, track the location of the iterator and the contents of the list

`Iterator` Implementation

- How do we code an iterator for a list?
  - The implementation depends on the underlying data structure
  - `ArrayList`, singly-linked list, and doubly-linked list should all provide their own unique iterator
- `interface Iterator<E>`
  - `java.util`

  - `boolean hasNext()`
  - `E next()`
  - How does this compare with the `Iterators` for `ArrayList`, `LinkedList` and so on?

Java Iterators

- `interface Iterable<T>`
  - `java.lang`
  - `Iterator<T> iterator()`
  - https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html
  - The `List` is only one of its subinterfaces, and is implemented by `ArrayList`, `LinkdList`, and so on
- What have we been missing so far?
  - Well, when implementing our own `ArrayList` or `LinkedList`, we should have implemented `.iterator()`
  - The return of `.iterator()` must be ab **object** that can perform `.next()` and `.hasNext()` on our list
    * But an object of which class?
    * How and where should we define that class?

`Iterator` as a Separate Class

```
package weiss.ds;

public class MyContainer {
    Object [] items;
    int size;

    public Iterator iterator() {
        return new MyContainerIterator(this);
    }

    // Other methods not shown
}

// An interator class that steps through a MyContainer.

package weiss.ds;

class MyContainerIterator implements Iterator {
    private int current = 0;
    private MyContaner container;

    MyContainerIterator(MyContainer c) {
        container = c;
    }

    public boolean hasNext() {
        return current < container.size;
    }
```

```
    public Object next() {
        return container.items[current++];
    }
}
```

- Container class
  - Issues?

`Iterator` as a Nested Class

```
package weiss.ds;

public class MyContainer {
    private Object [] items;
    private int size = 0;
    // Other methods for MyContainer not shown

    public Iterator iterator() {
        return new LocalIterator(this);
    }

    // The iterator class as a nested class
    private static class LocalIterator implements Iterator {
        private int current = 0;
        private MyContaner container;

        MyContainerIterator(MyContainer c) {
            container = c;
        }

        public boolean hasNext() {
            return current < container.size;
        }

        public Object next() {
            return container.items[current++];
        }
    }
}
```

Why use a Nested Class?

- Main issue: we don't want to expose details we don't have to to the outside world
  - The container class has to reveal details to `Iterator` (non-private data)
  - The `Iterator` class has to be accessible to the container class (package visible)
- Nested (private) iterator class: better encapsulation
  - Considered as a member of the container
  - Can access private data of the container

`Iterator` as an Inner Class

```
package weiss.ds;

public class MyContainer {
    private Object [] items;
    private int size = 0;
    // Other methods for MyContainer not shown

    public Iterator iterator() {
        return new LocalIterator(this);
    }

    // The iterator class as an inner class
    private class LocalIterator implements Iterator {
        private int current = 0;

        public boolean hasNext() {
            return current < MyContainer.this.size;
        }

        public Object next() {
            return MyContainer.this.items[current++];
        }
    }
}
```

Why use an Inner Class?

- Both nested and inner classes allow us to do the following:
  - Put multiple classes in a single file
  - Give access to the namespace of `OuterClass`
  - Have access to the private methods of `OuterClass`
- What's the difference between an inner class and a nested class?
  - A nested class can be `static`
  - Inner classes, upon construction, implicitly reference the instance of the outer class which caused its construction (`MyContainer.this`)
    - The constructor can also be dropped (the `MyContainer.this` reference is optional)

```
// The iterator class as an inner class

private class LocalIterator implements Iterator {
    private int current = 0;

    public boolean hasNext() {
        return current < size;
    }

    public Object next() {
        return items[current++];
    }
}
```

More Textbook Examples

- Singly linked node, list, and iterator (17.1-2)
  - All classes are separate entities!
    - `weiss.nonstandard.LinkedList`
    - `weiss.nonstandard.ListNode`
    - `weiss.nonstandard.LinkedListIterator`
- Doubly linkd list (17.3-5)
  - `weiss.util.LinkedList`
  - `weiss.util.LinkedList.LinkedListIterator` (inner, non-**`static`**)
  - `weiss.util.LinkedList.Node` (nested, **`static`**)

Typical Anonymous Class Style

```
class MyList<T> implements Iterable<T> {
    public Iterator<T> iterator() {
        return new Iterator<T>() {
            public boolean hasNext() {
                // Implementation
            }
            public T next() {
                // Implementation
            }
        };
    }
    ...
}
```

Two Ways to use an `Iterator`

```
public static void main main( String[] args ) {
    MyList<String> list = new MyList<>();
    list.add("Alpha");
    list.add("Bravo");
    list.add("Charlie");
    list.add("Delta");
    Iterator<String> iter = list.iterator();
    while( iter.hasNext() ) {
        String item = iter.next();
        System.out.println(item);
    }
    for (String item : list) {
        System.out.println(item);
    }
}
```

`ConcurrentModificationException`

- Our implementation doesn't try to coordinate multiple iterators changing the structure at the same time. **Note**: this is a different issue compared to access from different threads (we will not be covering that)

- Easy for reading and viewing
- Difficult for modification

```
Iterator itr1 = list.iterator();
Iterator itr2 = list.iterator();
itr1.remove();
itr2.next(); // Error!
```

Practice Problems

- Add `.iterator()` into our class examples of `ArrayList` and `LinkedList`
  - Pay attention to the details – it's always the edge cases that'll get ya
    - Where does it point when it is created?
    - What about `add()` and `remove()`?
  - What methods might be difficult in terms of time or algorithmic complexity for a singly linked list?

Summary

- `Iterators`
  - Support efficient traversing of all elements in the data structure
  - Implementation details are different, but `Iterator` and `Iterable` provide a common `interface`
- Next lecture: `Stack` and `Queue`
  - Reading: Chapter 6, Chapter 16