

CSE 143

Lecture 27

Advanced List Implementation

(ADTs; interfaces; abstract classes; inner classes;
generics; iterators)

read 11.1, 9.6, 15.3-15.4, 16.4-16.5

slides adapted from Marty Stepp

<http://www.cs.washington.edu/143/>

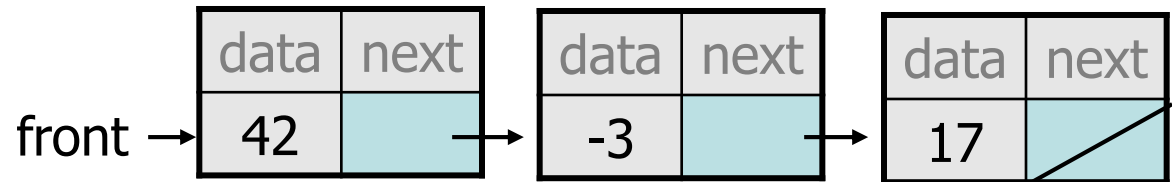
Our list classes

- We have implemented the following two list collection classes:

– `ArrayIntList`

index	0	1	2
value	42	-3	17

– `LinkedIntList`



– Problems:

- We should be able to treat them the same way in client code.
- **Some methods are implemented the same way (redundancy).**
- Linked list carries around a clunky extra node class.
- They can store only `int` elements, not any type of value.
- It is inefficient to get or remove each element of a linked list.

Common code

- Notice that some of the methods are implemented the same way in both the array and linked list classes.
 - `add(value)`
 - `contains`
 - `isEmpty`
- Should we change our interface to a class? Why / why not?
 - How can we capture this common behavior?

Abstract classes (9.6)

- **abstract class:** A hybrid between an interface and a class.
 - defines a superclass type that can contain method declarations (like an interface) and/or method bodies (like a class)
 - like interfaces, abstract classes that cannot be instantiated (cannot use `new` to create any objects of their type)
- What goes in an abstract class?
 - implementation of common state and behavior that will be inherited by subclasses (parent class role)
 - declare generic behaviors that subclasses must implement (interface role)

Abstract class syntax

```
// declaring an abstract class
public abstract class name {
    ...
    // declaring an abstract method
    // (any subclass must implement it)
    public abstract type name(parameters) ;
}
```

- A class can be `abstract` even if it has no abstract methods
- You can create variables (but not objects) of the abstract type
- Exercise: Introduce an abstract class into the list hierarchy.

Abstract and interfaces

- Normal classes that claim to implement an interface must implement all methods of that interface:

```
public class Empty implements IntList {} // error
```

- Abstract classes can claim to implement an interface without writing its methods; subclasses must implement the methods.

```
public abstract class Empty implements IntList {} // ok
```

```
public class Child extends Empty {} // error
```

An abstract list class

```
// Superclass with common code for a list of integers.
public abstract class AbstractIntList implements IntList {
    public void add(int value) {
        add(size(), value);
    }

    public boolean contains(int value) {
        return indexOf(value) >= 0;
    }

    public boolean isEmpty() {
        return size() == 0;
    }
}

public class ArrayIntList extends AbstractIntList { ...
public class LinkedIntList extends AbstractIntList { ...
```

Abstract class vs. interface

- Why do both interfaces and abstract classes exist in Java?
 - An abstract class can do everything an interface can do and more.
 - So why would someone ever use an interface?
- Answer: Java has single inheritance.
 - can extend only one superclass
 - can implement many interfaces
 - Having interfaces allows a class to be part of a hierarchy (polymorphism) without using up its inheritance relationship.

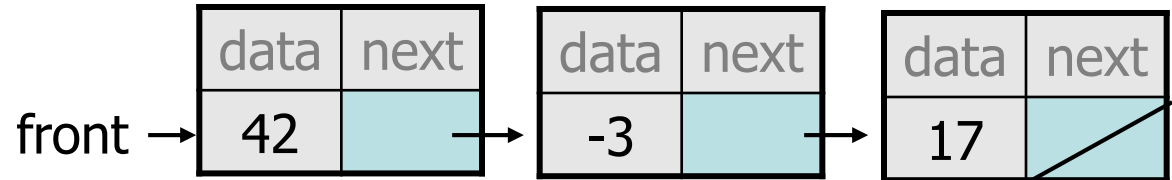
Our list classes

- We have implemented the following two list collection classes:

– `ArrayIntList`

index	0	1	2
value	42	-3	17

– `LinkedIntList`



– Problems:

- We should be able to treat them the same way in client code.
- Some of their methods are implemented the same way (redundancy).
- **Linked list carries around a clunky extra node class.**
- They can store only `int` elements, not any type of value.
- It is inefficient to get or remove each element of a linked list.

Recall: Inner classes

```
// outer (enclosing) class
public class name {
    ...

    // inner (nested) class
    private class name {
        ...
    }
}
```

- Only this file can see the inner class or make objects of it.
- Each inner object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
- Exercise: Convert the linked node into an inner class.

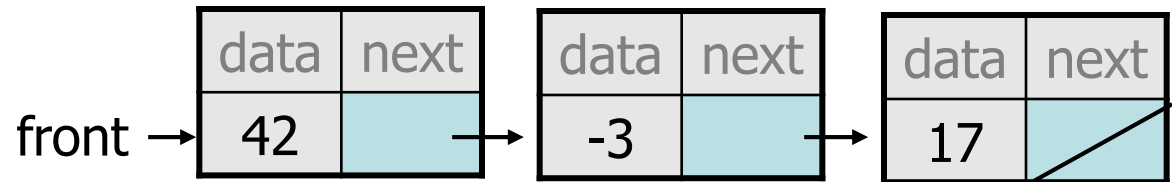
Our list classes

- We have implemented the following two list collection classes:

– `ArrayIntList`

index	0	1	2
value	42	-3	17

– `LinkedIntList`



– Problems:

- We should be able to treat them the same way in client code.
- Some of their methods are implemented the same way (redundancy).
- Linked list carries around a clunky extra node class.
- **They can store only `int` elements, not any type of value.**
- It is inefficient to get or remove each element of a linked list.

Implementing generics

```
// a parameterized (generic) class
public class name<Type> {
    ...
}
```

- Forces any client that constructs your object to supply a type.
 - Don't write an actual type such as `String`; the client does that.
 - Instead, write a type variable name such as `E` or `T`.
 - You can require multiple type parameters separated by commas.
- The rest of your class's code can refer to that type by name.
- Exercise: Convert our list classes to use generics.

Generics and arrays (15.4)

```
public class Foo<T> {  
    private T myField; // ok  
  
    public void method1(T param) {  
        myField = new T(); // error  
        T[] a = new T[10]; // error  
  
        myField = param; // ok  
        T[] a2 = (T[]) (new Object[10]); // ok  
    }  
}
```

- You cannot create objects or arrays of a parameterized type.
- You can create variables of that type, accept them as parameters, return them, or create arrays by casting from `Object[]`.

Comparing generic objects

```
public class ArrayList<E> {  
    ...  
    public int indexOf(E value) {  
        for (int i = 0; i < size; i++) {  
            // if (elementData[i] == value) {  
                if (elementData[i].equals(value)) {  
                    return i;  
                }  
            }  
        }  
        return -1;  
    }  
}
```

- When testing objects of type E for equality, must use `equals`

Generic interface (15.3, 16.5)

```
// Represents a list of values.
```

```
public interface List<E> {  
    public void add(E value);  
    public void add(int index, E value);  
    public E get(int index);  
    public int indexOf(E value);  
    public boolean isEmpty();  
    public void remove(int index);  
    public void set(int index, E value);  
    public int size();  
}  
  
public class ArrayList<E> implements List<E> { ...  
public class LinkedList<E> implements List<E> { ...
```

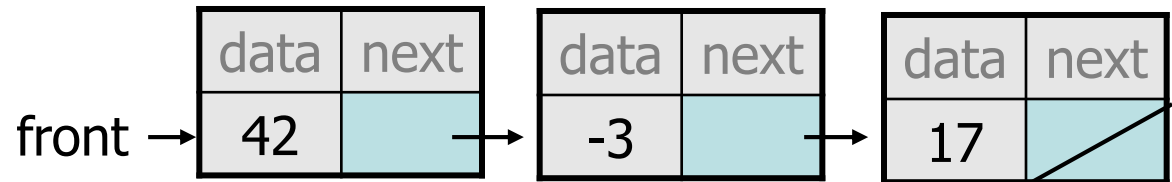
Our list classes

- We have implemented the following two list collection classes:

– `ArrayIntList`

index	0	1	2
value	42	-3	17

– `LinkedIntList`



– Problems:

- We should be able to treat them the same way in client code.
- Some of their methods are implemented the same way (redundancy).
- Linked list carries around a clunky extra node class.
- They can store only `int` elements, not any type of value.
- **It is inefficient to get or remove each element of a linked list.**

Linked list iterator

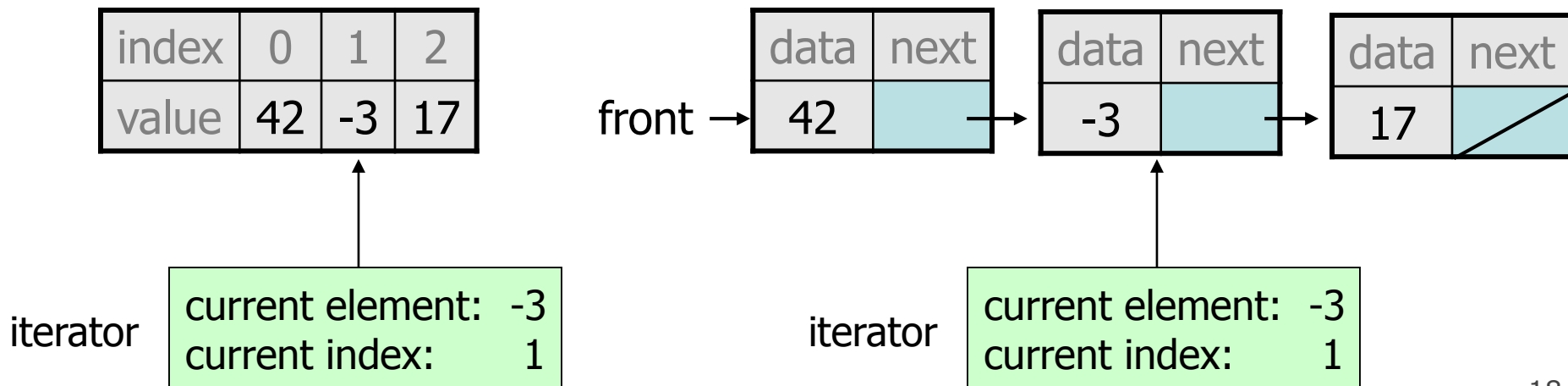
- The following code is particularly slow on linked lists:

```
List<Integer> list = new LinkedList<Integer>();  
...  
for (int i = 0; i < list.size(); i++) {  
    int value = list.get(i);  
    if (value % 2 == 1) {  
        list.remove(i);  
    }  
}
```

- Why?
- What can we do to improve the runtime?

Recall: Iterators (11.1)

- **iterator**: An object that allows a client to traverse the elements of a collection, regardless of its implementation.
 - Remembers a position within a collection, and allows you to:
 - get the element at that position
 - advance to the next position
 - (possibly) remove or change the element at that position
 - A common way to examine *any* collection's elements.



Iterator methods

<code>hasNext()</code>	returns <code>true</code> if there are more elements to examine
<code>next()</code>	returns the next element from the collection (throws a <code>NoSuchElementException</code> if there are none left to examine)
<code>remove()</code>	removes from the collection the last value returned by <code>next()</code> (throws <code>IllegalStateException</code> if you have not called <code>next()</code> yet)

- every provided collection has an `iterator` method

```
Set<String> set = new HashSet<String>();
```

```
...
```

```
Iterator<String> itr = set.iterator();
```

```
...
```

- Exercise: Write iterators for our linked list and array list.
 - You don't need to support the `remove` operation.

Array list iterator

```
public class ArrayList<E> extends AbstractIntList<E> {
    ...
    // not perfect; doesn't forbid multiple removes in a row
    private class ArrayIterator implements Iterator<E> {
        private int index;    // current position in list
        public ArrayIterator() {
            index = 0;
        }
        public boolean hasNext() {
            return index < size();
        }
        public E next() {
            index++;
            return get(index - 1);
        }
        public void remove() {
            ArrayList.this.remove(index - 1);
            index--;
        }
    }
}
```

Linked list iterator

```
public class LinkedList<E> extends AbstractIntList<E> {
    ...
    // not perfect; doesn't support remove
    private class LinkedIterator implements Iterator<E> {
        private ListNode current;    // current position in list
        public LinkedIterator() {
            current = front;
        }
        public boolean hasNext() {
            return current != null;
        }
        public E next() {
            E result = current.data;
            current = current.next;
            return result;
        }
        public void remove() {    // not implemented for now
            throw new UnsupportedOperationException();
        }
    }
}
```

for-each loop and Iterable

- Java's collections can be iterated using a "for-each" loop:

```
List<String> list = new LinkedList<String>();  
...  
for (String s : list) {  
    System.out.println(s);  
}
```

– Our collections do not work in this way.

- To fix this, your list must implement the `Iterable` interface.

```
public interface Iterable<E> {  
    public Iterator<E> iterator();  
}
```

Final List interface (15.3, 16.5)

```
// Represents a list of values.  
public interface List<E> extends Iterable<E> {  
    public void add(E value);  
    public void add(int index, E value);  
    public E get(int index);  
    public int indexOf(E value);  
    public boolean isEmpty();  
    public Iterator<E> iterator();  
    public void remove(int index);  
    public void set(int index, E value);  
    public int size();  
}
```