CSE 143
Lecture 25

Advanced collection classes

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

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

slides created by Marty Stepp

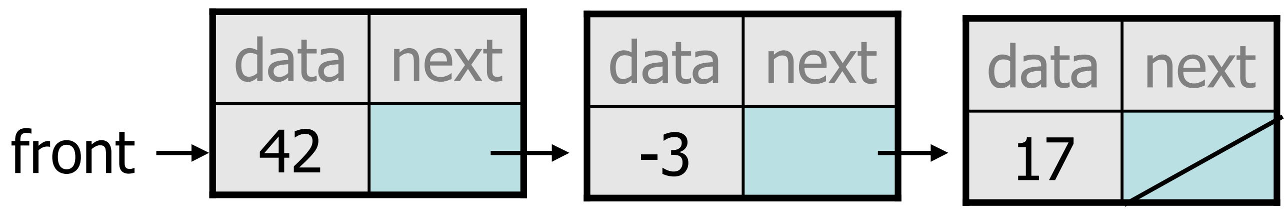
http://www.cs.washington.edu/143/

Our list classes

- We implemented the following two list 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: ADT interfaces (11.1)

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it.

- Java's collection framework describes ADTs with interfaces:
  - `Collection`, `Deque`, `List`, `Map`, `Queue`, `Set`, `SortedMap`

- An ADT can be implemented in multiple ways by classes:
  - `ArrayList` and `LinkedList` implement `List`
  - `HashSet` and `TreeSet` implement `Set`
  - `LinkedList` , `ArrayDeque`, etc. implement `Queue`

- Exercise: Create an ADT interface for the two list classes.

An `IntList` interface (16.4)

```
// Represents a list of integers.
public interface IntList {
    public void add(int value);
    public void add(int index, int value);
    public int get(int index);
    public int indexOf(int value);
    public boolean isEmpty();
    public void remove(int index);
    public void set(int index, int value);
    public int size();
}
```

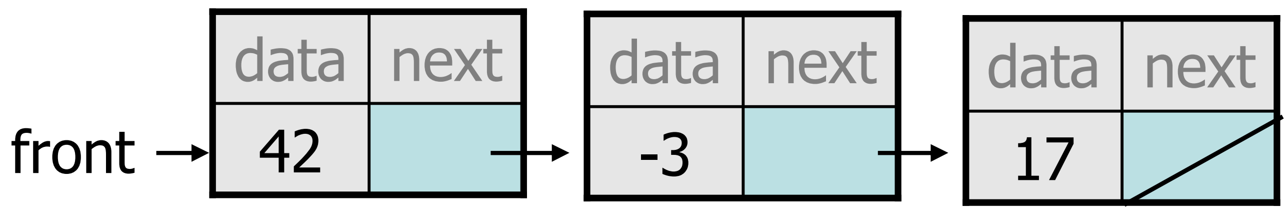
```
public class ArrayIntList implements IntList { ...

public class LinkedIntList implements IntList { ...
```

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?

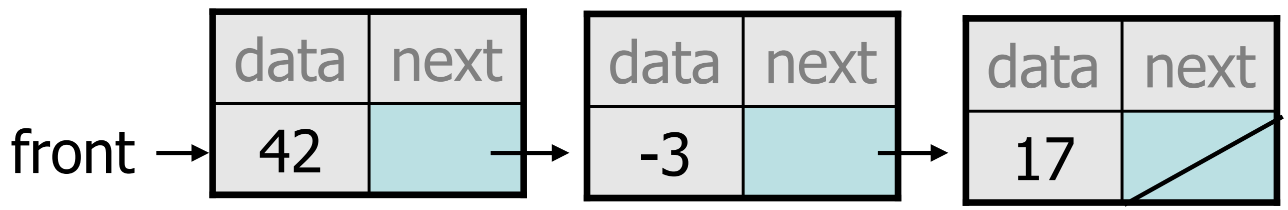
- Answer: Java has single inheritance.
  - can extend only one superclass
  - can implement many interfaces
  - Having interfaces allows a class to be part of a hierarchy (polymorphism) without using up its inheritance relationship.

Our list classes

- We have implemented the following two list collection classes:
  - `ArrayIntList`

index	0	1	2
value	42	-3	17

  - `LinkedIntList`

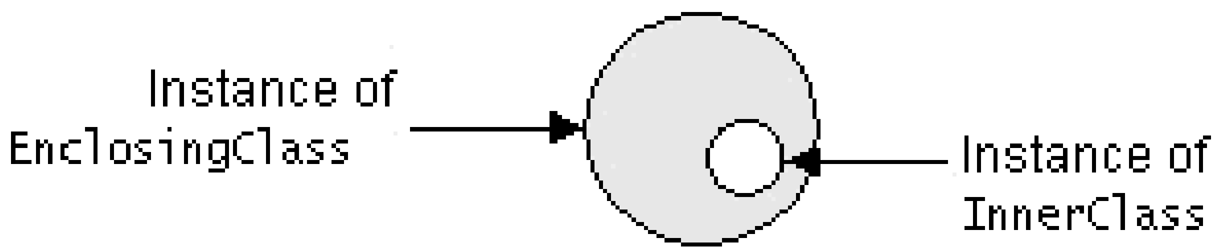


  - Problems:
    - We should be able to treat them the same way in client code.
    - Some of their methods are implemented the same way (redundancy).
    - **Linked list carries around a clunky extra node class.**
    - They can store only `int` elements, not any type of value.
    - It is inefficient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

Inner classes

- **inner class**: A class defined inside of another class.
  - can be created as `static` or non-static
  - we will focus on standard non-static ("nested") inner classes

- usefulness:
  - inner classes are hidden from other classes (encapsulated)
  - inner objects can access/modify the fields of the outer object



Inner class syntax

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

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

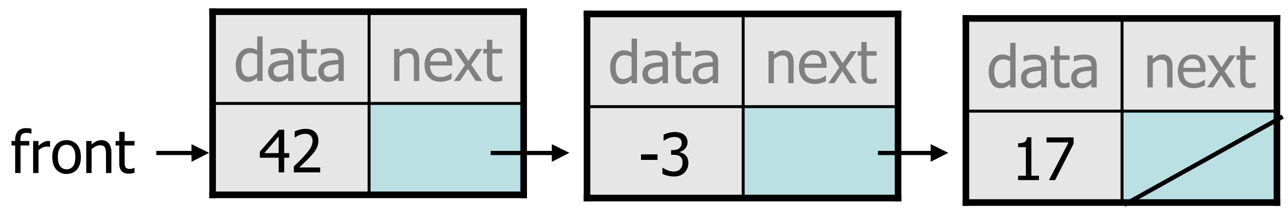
- Only this file can see the inner class or make objects of it.
- Each inner object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
  - If necessary, can refer to outer object as **OuterClassName**`.this`
- Exercise: Convert the linked 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.

Type Parameters (Generics)

```
ArrayList<Type> name = new ArrayList<Type>();
```

- Recall: When constructing a `java.util.ArrayList`, you specify the type of elements it will contain between < and >.
  - We say that the `ArrayList` class accepts a **type parameter**, or that it is a **generic** class.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Stuart Reges");
```

Implementing generics

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

- By putting the **Type** in < >, you are demanding that any client that constructs your object must supply a type parameter.
  - 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[]`.

Generics and inner classes

```
public class Foo<T> {
    private class Inner<T> {}    // incorrect

    private class Inner {}       // correct
}
```

- If an outer class declares a type parameter, inner classes can also use that type parameter.
- Inner class should NOT redeclare the type parameter. (If you do, it will create a second type parameter with the same name.)

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)

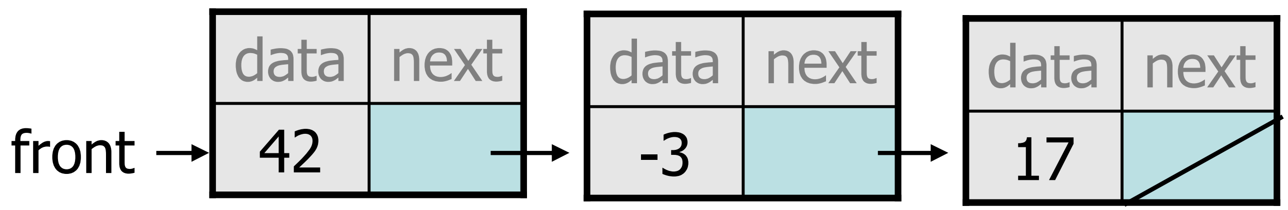
```
// Represents a list of values.
public interface List<E> {
    public void add(E value);
    public void add(int index, E value);
    public E get(int index);
    public int indexOf(E value);
    public boolean isEmpty();
    public void remove(int index);
    public void set(int index, E value);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

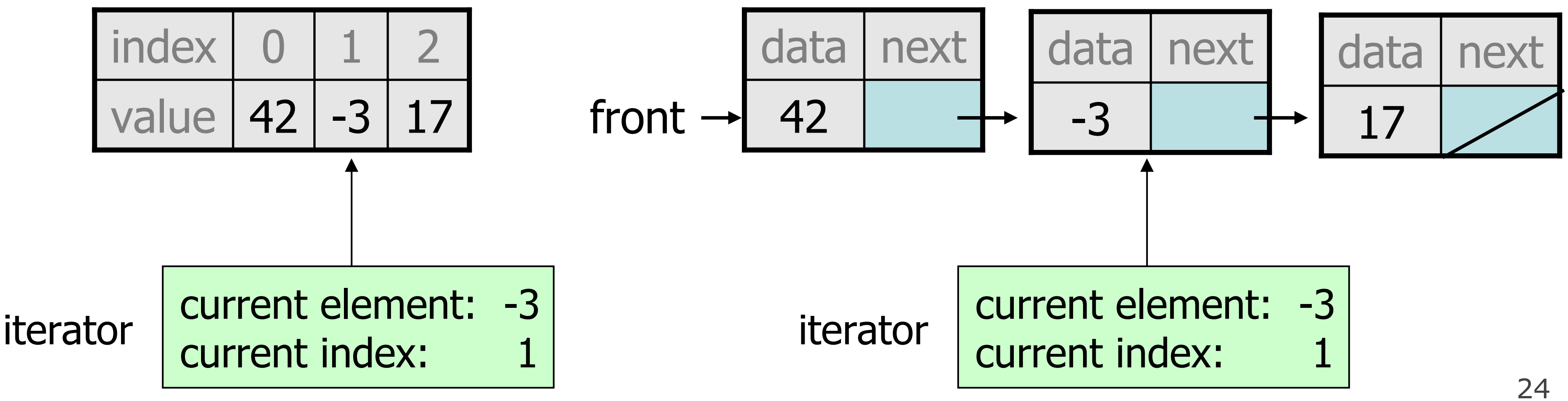
- The following code is particularly slow on linked lists:

```
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
  - Benefit: A common way to examine *any* collection's elements.



`Iterator` methods

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

- every provided collection has an `iterator` method

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

- Exercise: Write iterators for our array list and linked 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();
}
```

Hashing

read 11.2

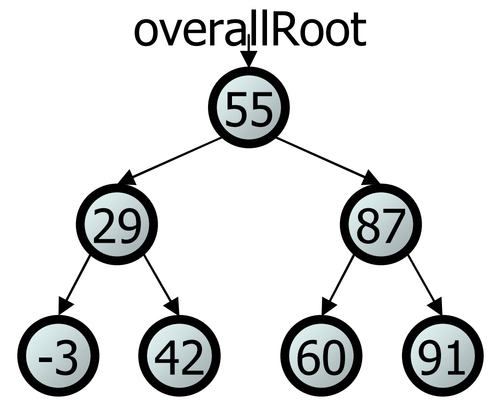
SearchTree as a set

- We implemented a class `SearchTree` to store a BST of `int`s:

- Our BST is essentially a set of integers.

  Operations we support:
  - `add`
  - `contains`
  - `remove`

    ...



- But there are other ways to implement a set...

How to implement a set?

- Elements of a `TreeSet` (`IntTree`) are in BST sorted order.
  - We need this in order to add or search in O(log *N*) time.

- But it doesn't really matter what order the elements appear in a set, so long as they can be added and searched quickly.

- Consider the task of storing a set in an array.
  - What would make a good ordering for the elements?

index	0	1	2	3	4	5	6	7	8	9
value	7	11	24	49	0	0	0	0	0	0

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	0	0	7	0	49

Hashing

- **hash**: To map a value to an integer index.
  - **hash table**: An array that stores elements via hashing.

- **hash function**: An algorithm that maps values to indexes.
  - one possible hash function for integers:

$$HF(I) \rightarrow I \ \% \ length$$

```
set.add(11);        // 11 % 10 == 1
set.add(49);        // 49 % 10 == 9
set.add(24);        // 24 % 10 == 4
set.add(7);         //  7 % 10 == 7
```

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	0	0	7	0	49

Efficiency of hashing

```
public static int HF(int i) {      // hash function
    return Math.abs(i) % elementData.length;
}
```

- Add: simply set `elementData[HF(i)] = i;`
- Search: check if `elementData[HF(i)] == i`
- Remove: set `elementData[HF(i)] = 0;`

- What is the runtime of `add`, `contains`, and `remove`?
  - **O(1)!** OMGWTFBBQFAST

- Are there any problems with this approach?

Collisions

- **collision**: When a hash function maps two or more elements to the same index.

```
set.add(11);
set.add(49);
set.add(24);
set.add(7);
set.add(54);   // collides with 24!
```

- **collision resolution**: An algorithm for fixing collisions.

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	54	0	0	7	0	49

Probing

- **probing**: Resolving a collision by moving to another index.
  - **linear probing**: Moves to the next index.

```
set.add(11);
set.add(49);
set.add(24);
set.add(7);
set.add(54);   // collides with 24
```

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	**54**	0	7	0	49

  - Is this a good approach?

Clustering

- **clustering**: Clumps of elements at neighboring indexes.
  - slows down the hash table lookup; you must loop through them.

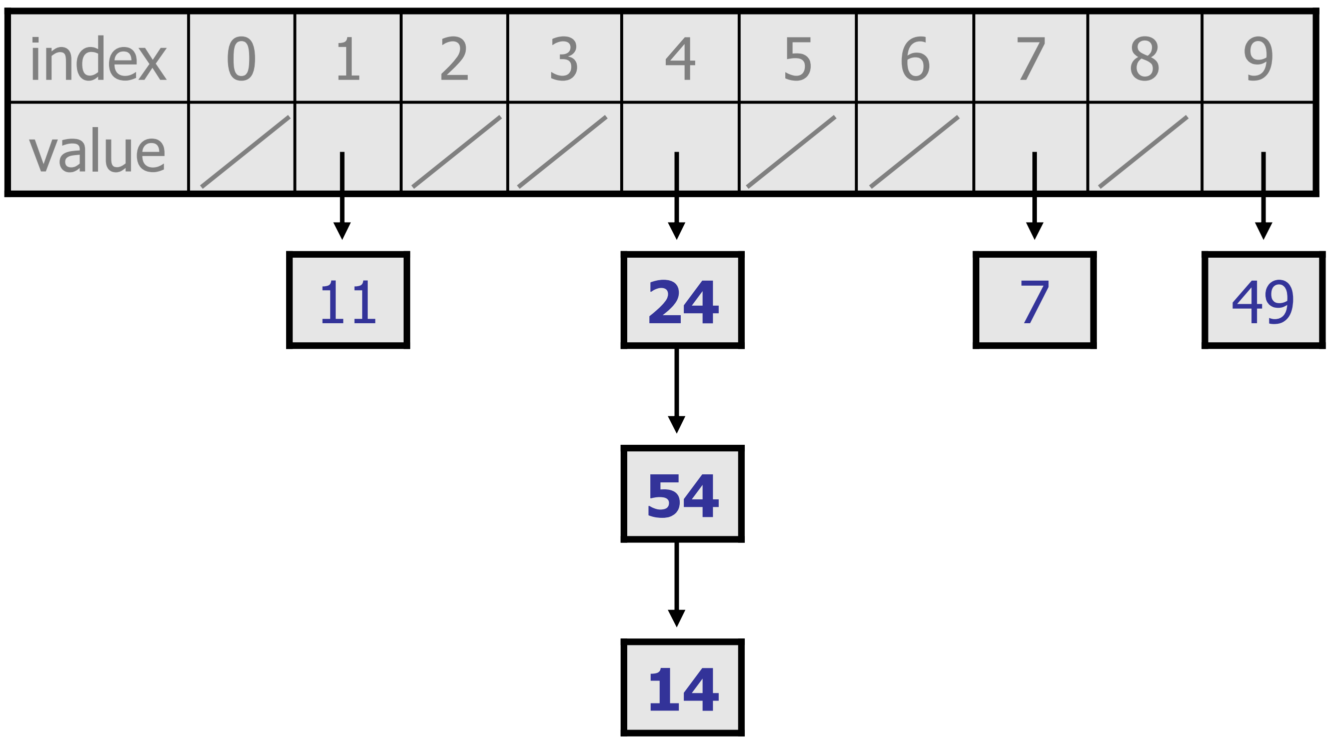
```
set.add(11);
set.add(49);
set.add(24);
set.add(7);
set.add(54);   // collides with 24
set.add(14);   // collides with 24, then 54
set.add(86);   // collides with 14, then 7
```

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	**24**	**54**	**14**	**7**	**86**	49

  - Now a lookup for 94 must look at 7 out of 10 total indexes.

Chaining

- **chaining**: Resolving collisions by storing a list at each index.
  - add/search/remove must traverse lists, but the lists are short
  - impossible to "run out" of indexes, unlike with probing



Hash set code

```
import java.util.*;   // for List, LinkedList
// All methods assume value != null;  does not rehash
public class HashIntSet {
    private static final int CAPACITY = 137;
    private List<Integer>[] elements;
    // constructs new empty set
    public HashSet() {
        elements = (List<Integer>[]) (new List[CAPACITY]);
    }
    // adds the given value to this hash set
    public void add(int value) {
        int index = HF(value);
        if (elements[index] == null) {
            elements[index] = new LinkedList<Integer>();
        }
        elements[index].add(value);
    }
    // hashing function to convert objects to indexes
    private int HF(int value) {
        return Math.abs(value) % elements.length;
    }
    ...
```

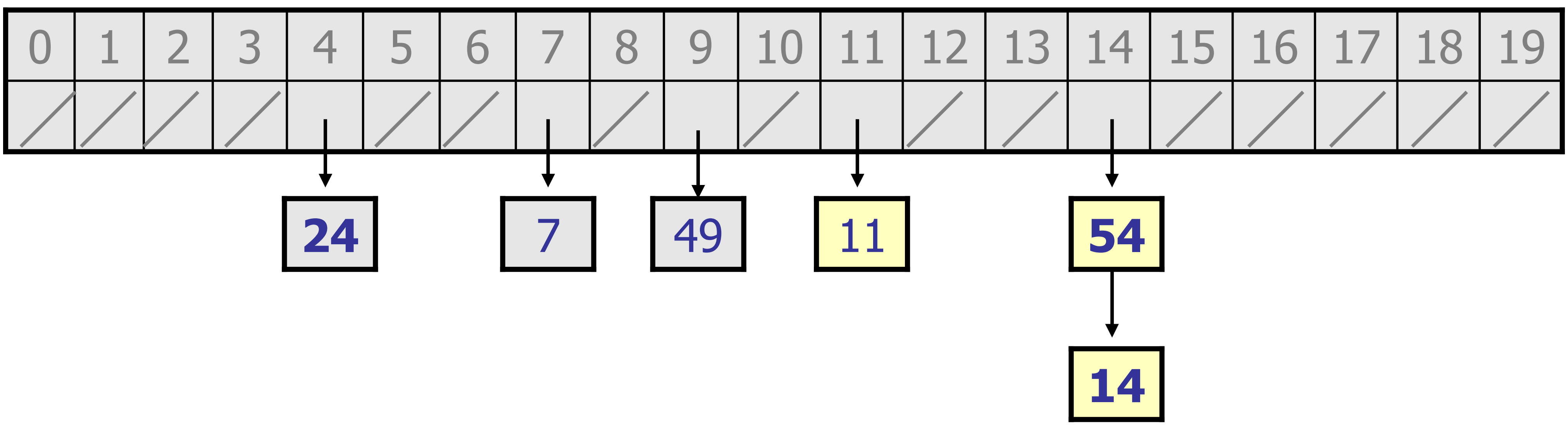
Final hash set code 2

```
    ...
    // Returns true if this set contains the given value.
    public boolean contains(int value) {
        int index = HF(value);
        return elements[index] != null &&
               elements[index].contains(value);
    }

    // Removes the given value from the set, if it exists.
    public void remove(int value) {
        int index = HF(value);
        if (elements[index] != null) {
            elements[index].remove(value);
        }
    }
}
```

Rehashing

- **rehash**: Growing to a larger array when the table is too full.
  - Cannot simply copy the old array to a new one. (Why not?)

- **load factor**: ratio of (*# of elements*) / (*hash table length*)
  - many collections rehash when load factor ≅ .75
  - can use big prime numbers as hash table sizes to reduce collisions



Hashing objects

- It is easy to hash an integer I (use index *I % length*).
  - How can we hash other types of values (such as objects)?

- All Java objects contain the following method:

  ```
  public int hashCode()
  ```

  Returns an integer hash code for this object.

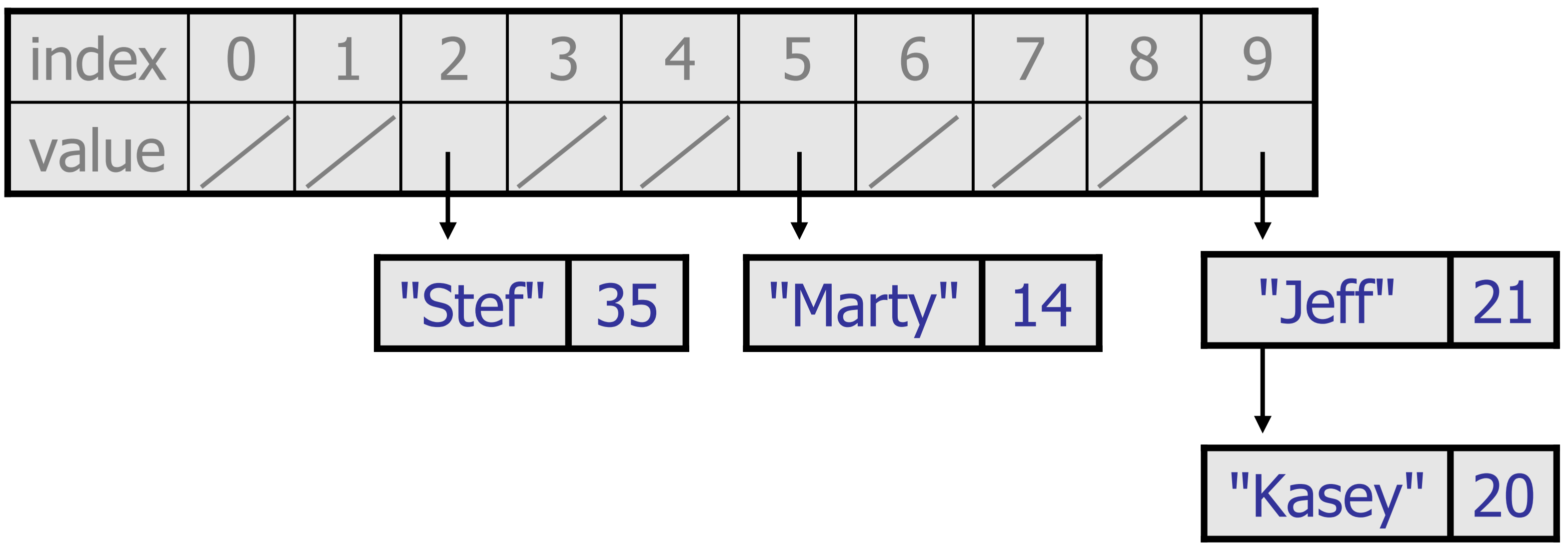
  - We can call `hashCode` on any object to find its preferred index.

- How is `hashCode` implemented?
  - Depends on the type of object and its state.
    - Example: a String's `hashCode` adds the ASCII values of its letters.
  - You can write your own `hashCode` methods in classes you write.

Implementing hash maps

- A hash map is just a set where the lists store key/value pairs:

```
//          key      value
map.put("Marty", 14);
map.put("Jeff",  21);
map.put("Kasey", 20);
map.put("Stef",  35);
```



  - Instead of a `List<Integer>`, write an inner `Entry` node class with `key` and `value` fields; the map stores a `List<Entry>`