CSE 143
Lecture 16

Iterators; Searching

reading: 13.1 - 13.4; 15.3; 16.5

slides adapted from Marty Stepp

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

Exercises

- Modify the word count program to print every word that appeared in the book at least 1000 times, in sorted order from least to most occurrences.

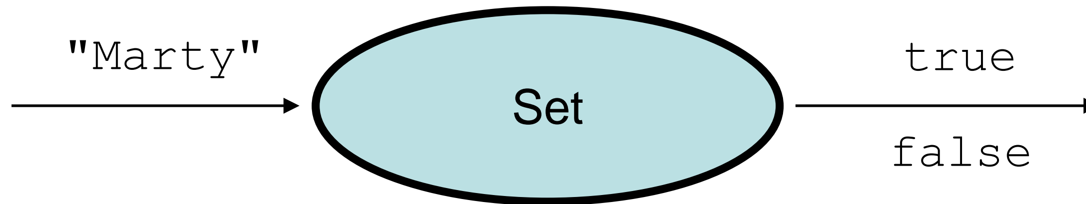
- Write a program that reads a list of TA names and quarters' experience, then prints the quarters in increasing order of how many TAs have that much experience, along with their names.

```
Allison 5
Alyssa 8
Brian 1
Kasey 5
...
```

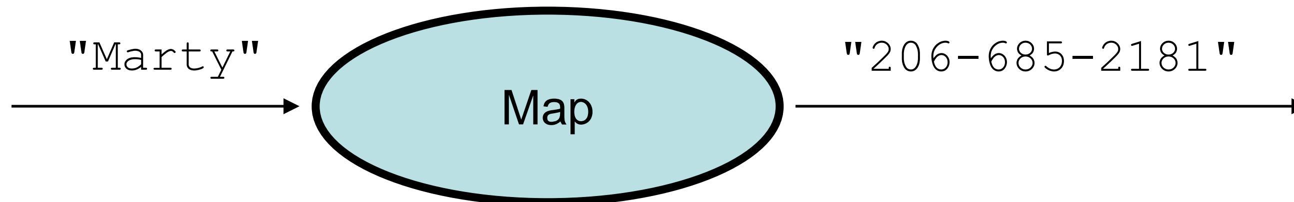
```
1 qtr: [Brian]
2 qtr: ...
5 qtr: [Allison, Kasey]
```

Maps vs. sets

- A set is like a map from elements to `boolean` values.
  - *Set: Is Marty found in the set? (true/false)*



  - *Map: What is Marty's phone number?*



Iterators

reading: 11.1; 15.3; 16.5

Examining sets and maps

- elements of Java `Set`s and `Map`s can't be accessed by index
  - must use a "foreach" loop:

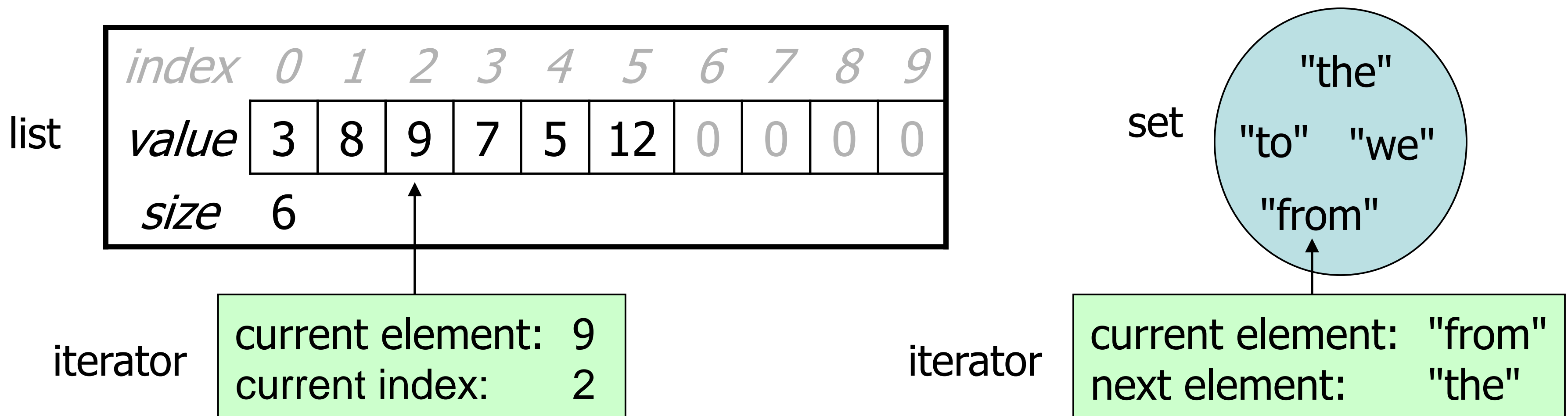
```
Set<Integer> scores = new HashSet<Integer>();
for (int score : scores) {
    System.out.println("The score is " + score);
}
```

  - Problem: foreach is read-only; cannot modify set while looping

```
for (int score : scores) {
    if (score < 60) {
    // throws a ConcurrentModificationException
        scores.remove(score);
    }
}
```

Iterators (11.1)

- **iterator**: An object that allows a client to traverse the elements of any collection.
  - Remembers a position, and lets you:
    - get the element at that position
    - advance to the next position
    - remove the element at that position



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 the last value returned by next() (throws an IllegalStateException if you haven't called next() yet)

- Iterator interface in java.util
  - every collection has an iterator() method that returns an iterator over its elements

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

Iterator example

```
Set<Integer> scores = new TreeSet<Integer>();
scores.add(94);
scores.add(38);    // Jenny
scores.add(87);
scores.add(43);    // Marty
scores.add(72);
...

Iterator<Integer> itr = scores.iterator();
while (itr.hasNext()) {
    int score = itr.next();

    System.out.println("The score is " + score);

    // eliminate any failing grades
    if (score < 60) {
        itr.remove();
    }
}
System.out.println(scores);  // [72, 87, 94]
```

Iterator example 2

```
Map<String, Integer> scores = new TreeMap<String, Integer>();
scores.put("Jenny", 38);
scores.put("Stef", 94);
scores.put("Greg", 87);
scores.put("Marty", 43);
scores.put("Angela", 72);
...

Iterator<String> itr = scores.keySet().iterator();
while (itr.hasNext()) {
    String name = itr.next();
    int score = scores.get(name);
    System.out.println(name + " got " + score);

    // eliminate any failing students
    if (score < 60) {
        itr.remove();      // removes name and score
    }
}
System.out.println(scores);  // {Greg=87, Stef=94, Angela=72}
```

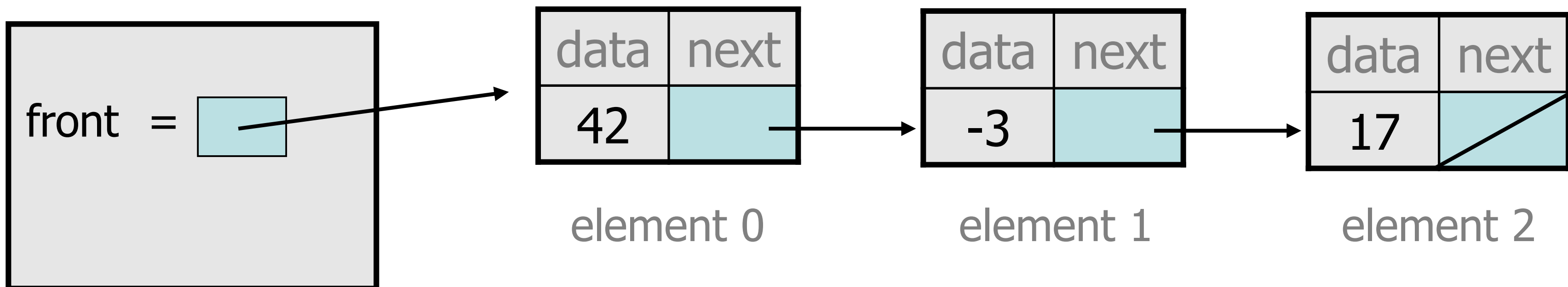
A surprising example

- What's bad about this code?

```
List<Integer> list = new LinkedList<Integer>();
```

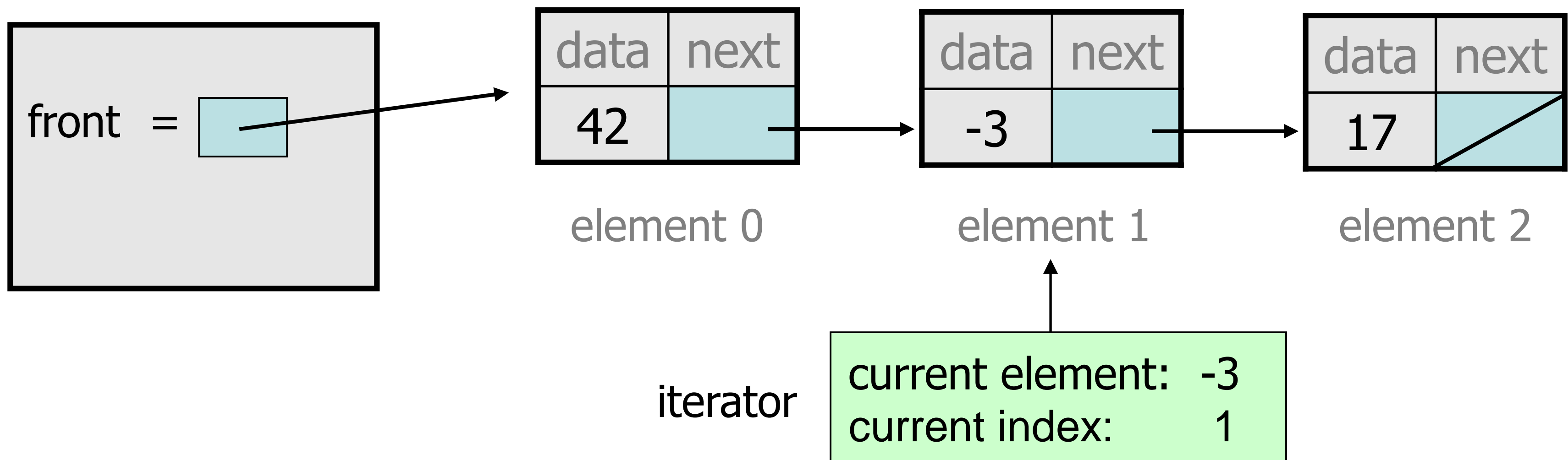
... (add lots of elements) ...

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



Iterators and linked lists

- Iterators are particularly useful with linked lists.
  - The previous code is $O(N^2)$ because each call on `get` must start from the beginning of the list and walk to index `i`.
  - Using an iterator, the same code is O(N). The iterator remembers its position and doesn't start over each time.



Exercise

- Modify the Book Search program from last lecture to eliminate any words that are plural or all-uppercase from the collection.

- Modify the TA quarters experience program so that it eliminates any TAs with 3 quarters or fewer of experience.

ListIterator

`add(`**value**`)`	inserts an element just after the iterator's position
`hasPrevious()`	`true` if there are more elements *before* the iterator
`nextIndex()`	the index of the element that would be returned the next time `next` is called on the iterator
`previousIndex()`	the index of the element that would be returned the next time `previous` is called on the iterator
`previous()`	returns the element before the iterator (throws a `NoSuchElementException` if there are none)
`set(`**value**`)`	replaces the element last returned by `next` or `previous` with the given value

```
ListIterator<String> li = myList.listIterator();
```

- lists have a more powerful `ListIterator` with more methods
  - can iterate forwards or backwards
  - can add/set element values (efficient for linked lists)

Searching

reading: 13.1 – 13.4

Binary search (13.1)

- **binary search**: Locates a target value in a *sorted* array/list by successively eliminating half of the array from consideration.
  - How many elements will it need to examine? **O(log N)**
  - Can be implemented with a loop or recursively
  - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

min (index 0) mid (index 8) max (index 16)

Binary search code

```
// Returns the index of an occurrence of target in a,
// or a negative number if the target is not found.
// Precondition: elements of a are in sorted order
public static int binarySearch(int[] a, int target) {
    int min = 0;
    int max = a.length - 1;

    while (min <= max) {
        int mid = (min + max) / 2;
        if (a[mid] < target) {
            min = mid + 1;
        } else if (a[mid] > target) {
            max = mid - 1;
        } else {
            return mid;    // target found
        }
    }

    return -(min + 1);    // target not found
}
```

Recursive binary search (13.3)

- Write a recursive `binarySearch` method.
  - If the target value is not found, return its negative insertion point.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

```
int index  = binarySearch(data, 42);  // 10
int index2 = binarySearch(data, 66);  // -14
```

Exercise solution

```
// Returns the index of an occurrence of the given value in
// the given array, or a negative number if not found.
// Precondition: elements of a are in sorted order
public static int binarySearch(int[] a, int target) {
    return binarySearch(a, target, 0, a.length - 1);
}

// Recursive helper to implement search behavior.
private static int binarySearch(int[] a, int target,
                                int min, int max) {
    if (min > max) {
        return -1;          // target not found
    } else {
        int mid = (min + max) / 2;
        if (a[mid] < target) {            // too small; go right
            return binarySearch(a, target, mid + 1, max);
        } else if (a[mid] > target) {  // too large; go left
            return binarySearch(a, target, min, mid - 1);
        } else {
            return mid;    // target found; a[mid] == target
        }
    }
}
```