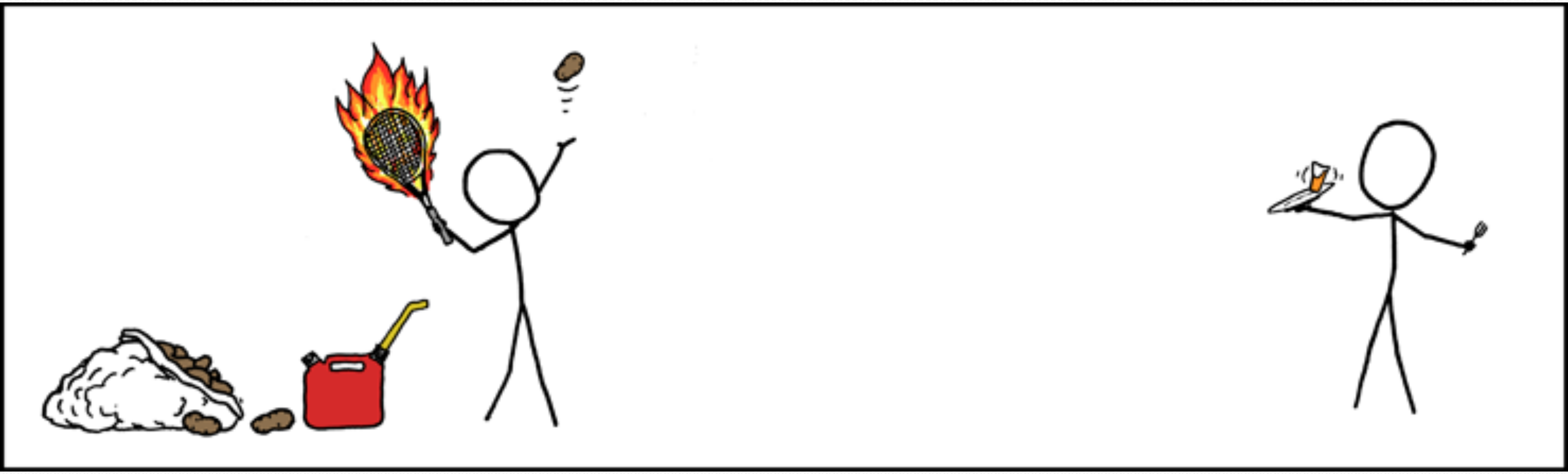
Building Java Programs

Iterators, hashing

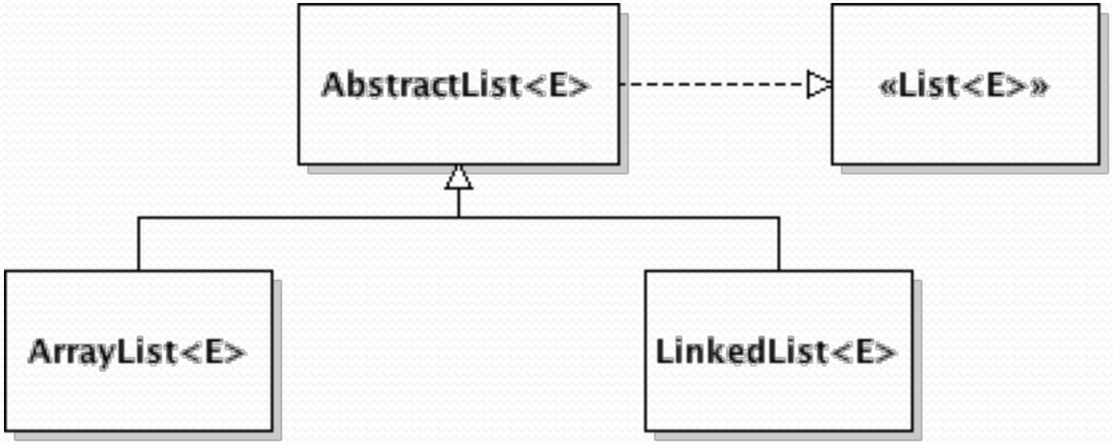
reading: 11.1, 15.3



Making hash browns...

(We're talking about hashing. It makes sense.)

List Case Study



Linked list performance

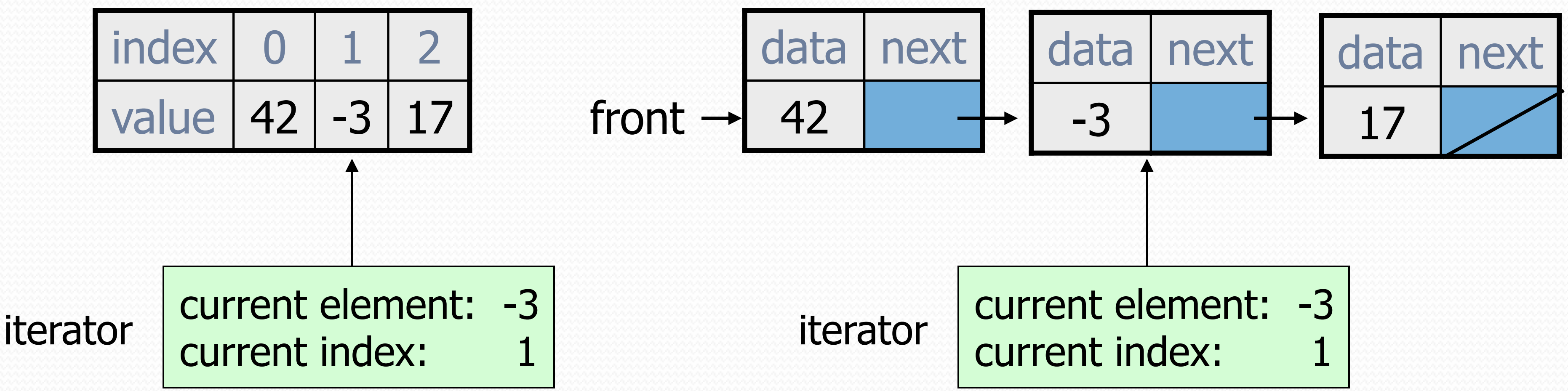
- The following code is particularly slow on linked lists:

```
public void addAll(List<E> other) {
  for (int i = 0; i < other.size(); i++) {
        add(other.get(i));
  }
}
```

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

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

`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 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

```java
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.

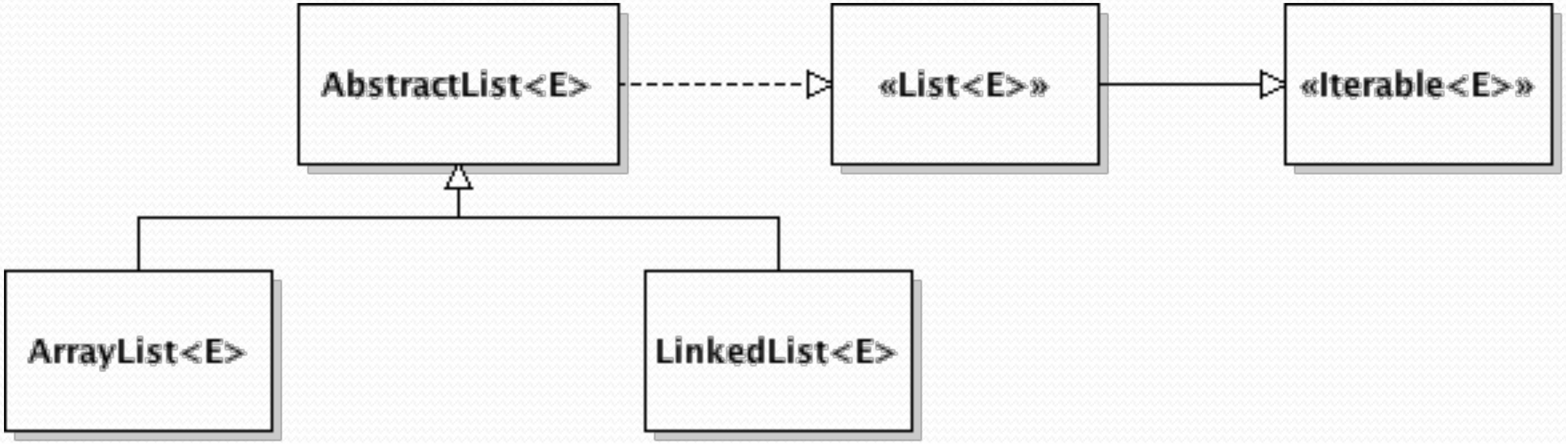
- To fix this, your list must implement the `Iterable` interface.

```
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();
}
```

List Case Study



Runtimes

Structure	add	find	remove
unsorted array	O(1)	O(N)	O(N)
sorted array	O(N)	O(logN)	O(N)
unsorted linked list	O(1)	O(N)	O(N)
sorted linked list	O(N)	O(N)	O(N)
binary search tree	O(logN)	O(logN)	O(logN)

- Why Java includes tree implementations of sets and maps
- Can we do better?

Ordering in sets/maps

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

- But it **doesn't 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) → 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 hashFunction(int i) {
    return Math.abs(i) % elementData.length;
}
```

- Add: 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)**

- Are there any problems with this approach?

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.
    - All classes come with a default version based on memory address.

Hash function for objects

```
public static int hashFunction(E e) {
    return Math.abs(e.hashCode()) % elements.length;
}
```

- Add: set `elements[HF(o)] = o;`
- Search: check if `elements[HF(o)].equals(o)`
- Remove: set `elements[HF(o)] = null;`

String's hashCode

- The `hashCode` function inside `String` objects looks like this:

```
public int hashCode() {
    int hash = 0;
    for (int i = 0; i < this.length(); i++) {
        hash = 31 * hash + this.charAt(i);
    }
    return hash;
}
```

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

- As with any general hashing function, collisions are possible.
  - Example: "Ea" and "FB" have the same hash value.
- Early versions of the Java examined only the first 16 characters.
  For some common data this led to poor hash table performance.

Collisions

- **collision**: When hash function maps 2 values to 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; must probe
```

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?
    - variation: **quadratic probing** moves increasingly far away

Clustering

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

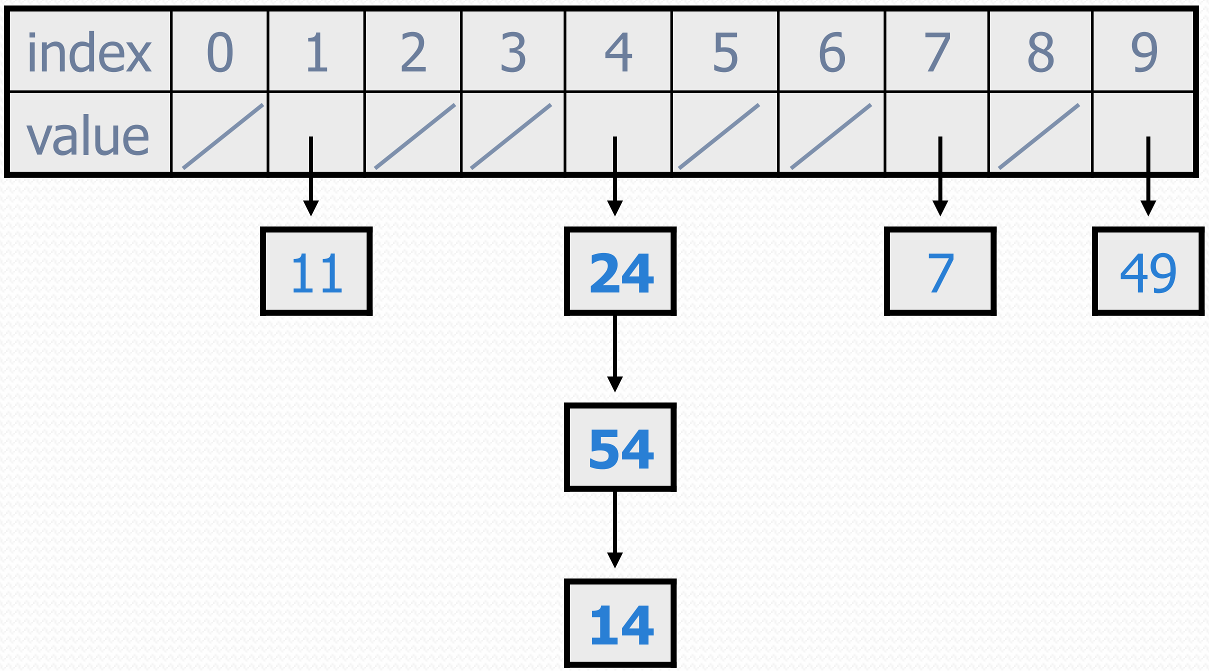
```
set.add(11);
set.add(49);
set.add(24);
set.add(7);
set.add(54);   // collides with 24
set.add(14);   // collides with 24, then 54
set.add(86);   // collides with 14, then 7
```

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

  - How many indexes must a lookup for 94 visit?

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

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 = hashFunction(value);
        if (elements[index] == null) {
            elements[index] = new LinkedList<Integer>();
        }
        elements[index].add(value);
    }

    // hashing function to convert objects to indexes
    private int hashFunction(int value) {
        return Math.abs(value) % elements.length;
    }
    ...
```

Hash set code 2

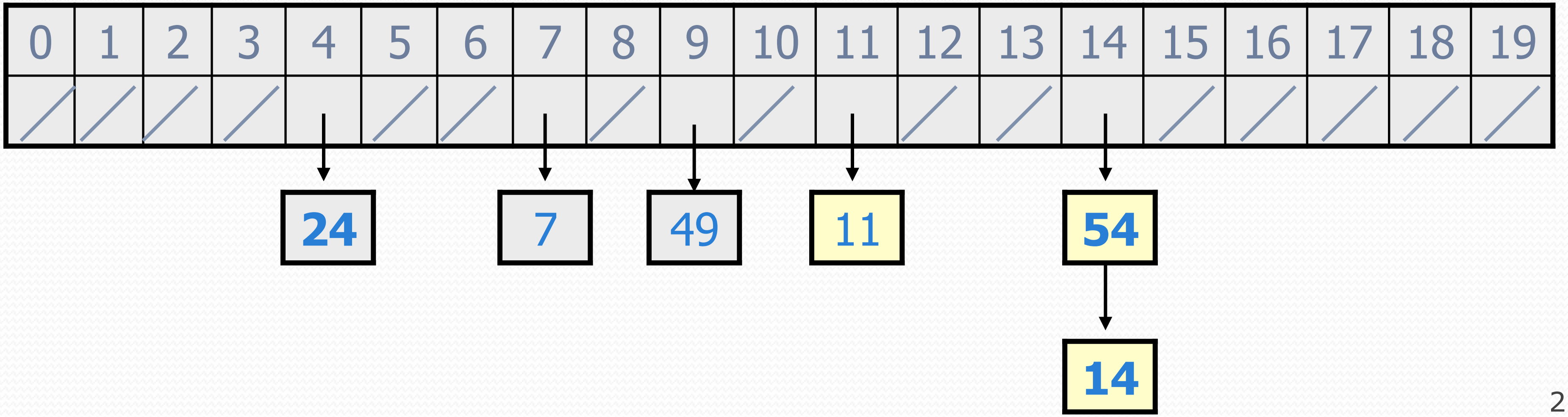
```
    ...
    // Returns true if this set contains the given value.
    public boolean contains(int value) {
        int index = hashFunction(value);
        return elements[index] != null &&
               elements[index].contains(value);
    }

    // Removes the given value from the set, if it exists.
    public void remove(int value) {
        int index = hashFunction(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

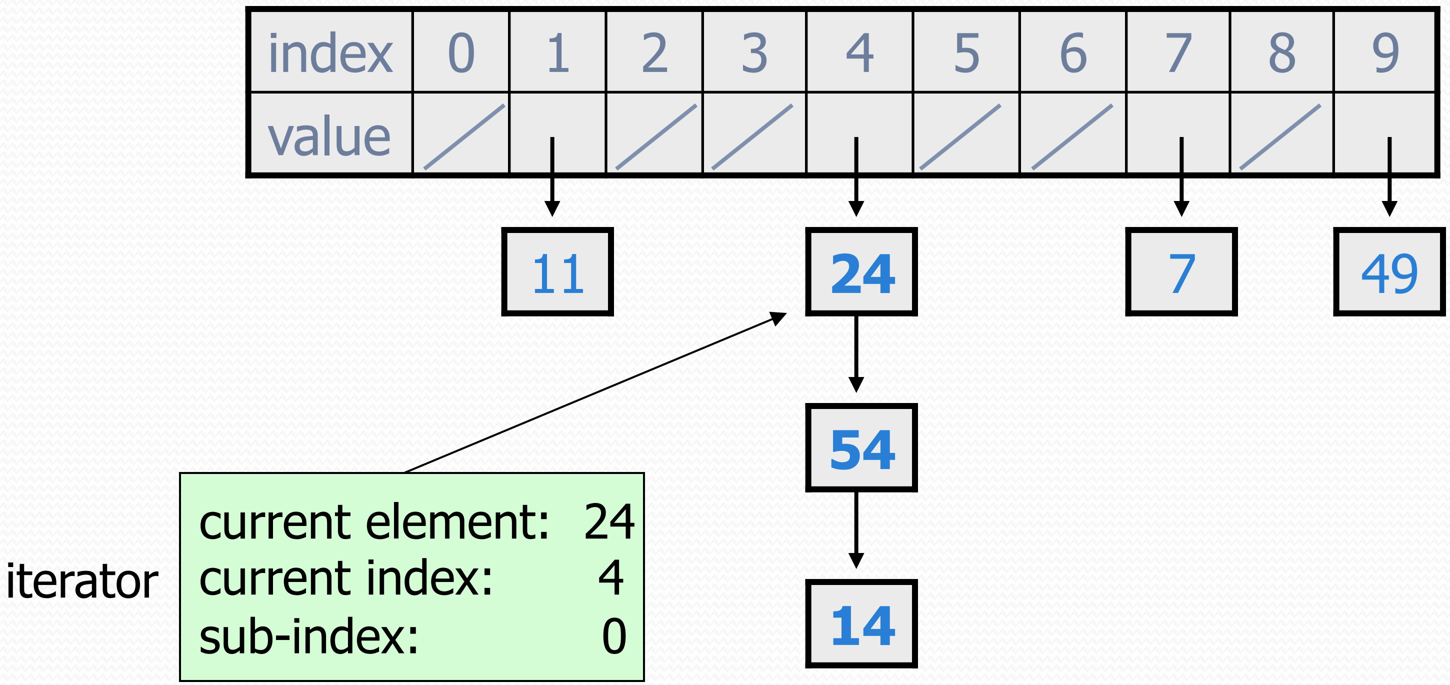


Rehashing code

```
...
// Grows hash array to twice its original size.
private void rehash() {
    List<Integer>[] oldElements = elements;
    elements = (List<Integer>[])
                   new List[2 * elements.length];
    for (List<Integer> list : oldElements) {
        if (list != null) {
            for (int element : list) {
                add(element);
            }
        }
    }
}
```

Other questions

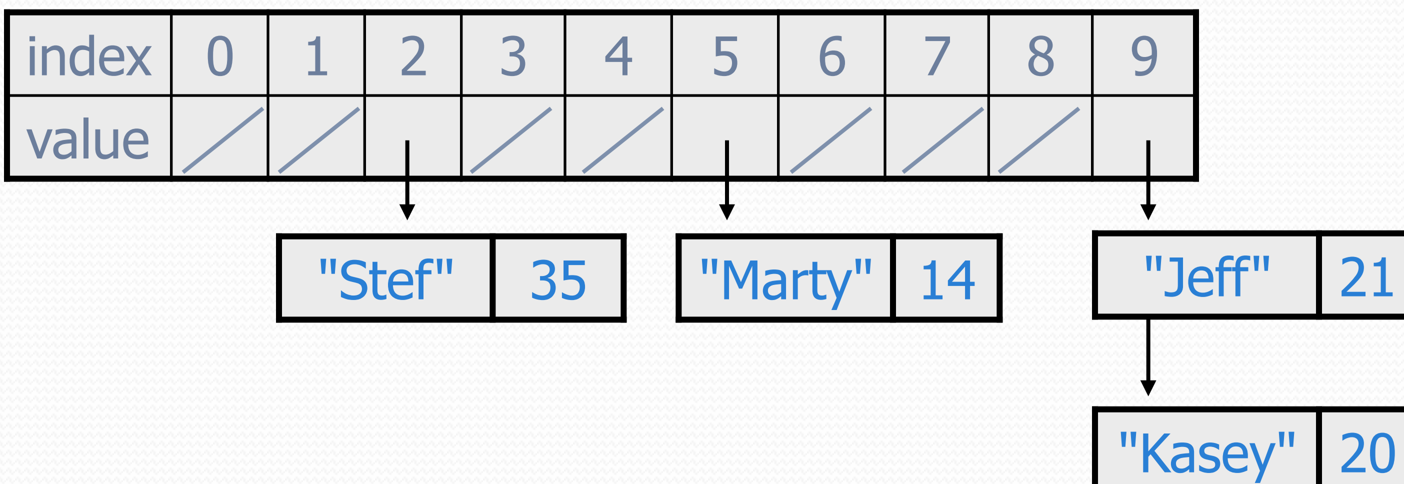
- How would we implement `toString` on a `HashSet`?
- How would we implement an `Iterator` over a `HashSet`?



Implementing a hash map

- 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>`

Implementing a tree map

- Similar to difference between `HashMap` and `HashSet`:
  - Each node now will store both a key and a value
  - tree is BST ordered by keys
  - keys must be `Comparable`