





Building Java Programs

Hashing

reading: 18.1

Arrays

- **Random access:** we have fast access if we know the index we are looking for in an array
- How would we add a value to an unsorted array of integers? How fast is this?

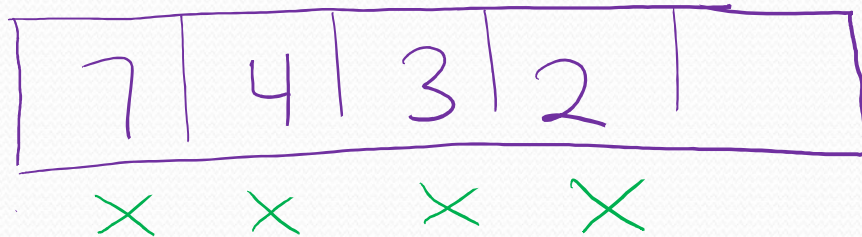
add(2)

$O(1)$ Constant

- How would we see if our unsorted array contains a particular value? How fast is this?

contains(8)

$O(n)$ linear



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
set.add(28);          // 28 % 10 == 8
```

length = 10

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

set.add(21);

↑ // 21 % 10 == 1

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?

Hash Functions

- Maps an object to a number
 - result should be constrained to some range
 - passing in the same object should always give the same result
- Results from a hash function should be distributed over a range
 - very bad if everything hashes to 1!
 - should "look random"
- How would we write a hash function for String objects?

$$\text{"abc"} \rightarrow 0 * 97 + 1 * 98 + 2 * 99$$

$$\text{"cba"} \rightarrow 0 * 99 + 1 * 98 + 2 * 97$$

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;  
}
```

"doggo"

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

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



Implementing generics

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

- 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 \mathbb{E} (for "element") or \mathbb{T} (for "type").
 - You can require multiple type parameters separated by commas.
- The rest of your class's code can refer to that type by name.

Collisions

- **collision:** When hash function maps 2 values to same index.

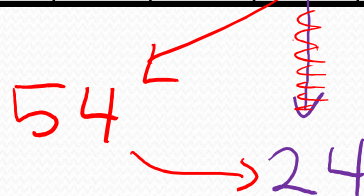
```
set.add(11);  
set.add(49);  
set.add(24);  
set.add(7);
```

num % length

```
set.add(54); // collides with 24! Where should it go?
```

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

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

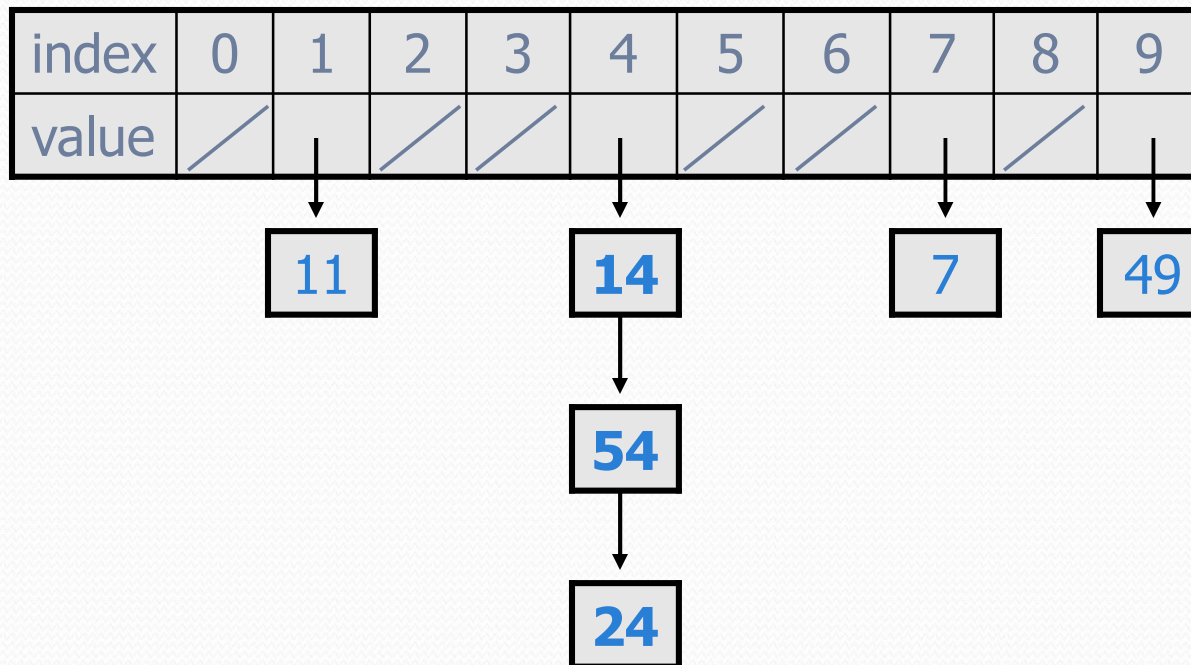
```
set.add(11);  
set.add(49);  
set.add(24);  
set.add(7);  
set.add(54); // collides with 24  
set.add(14); // collides with 24, then 54  
set.add(86); // collides with 14, then 7
```

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



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);
}

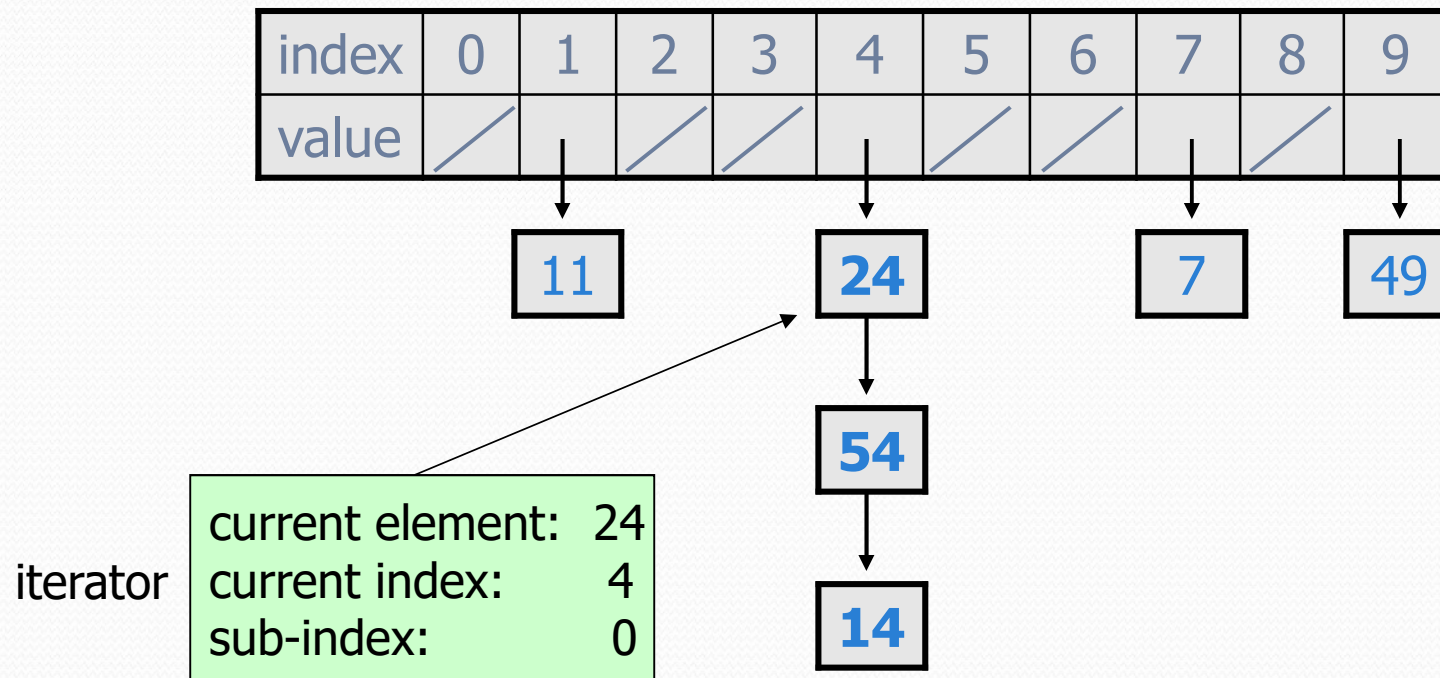
// Removes the given value from the set, if it exists.
public void remove(int value) {
    int index = hashFunction(value);
    if (elements[index] != null) {
        elements[index].remove(value);
    }
}
}
```


Rehashing 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

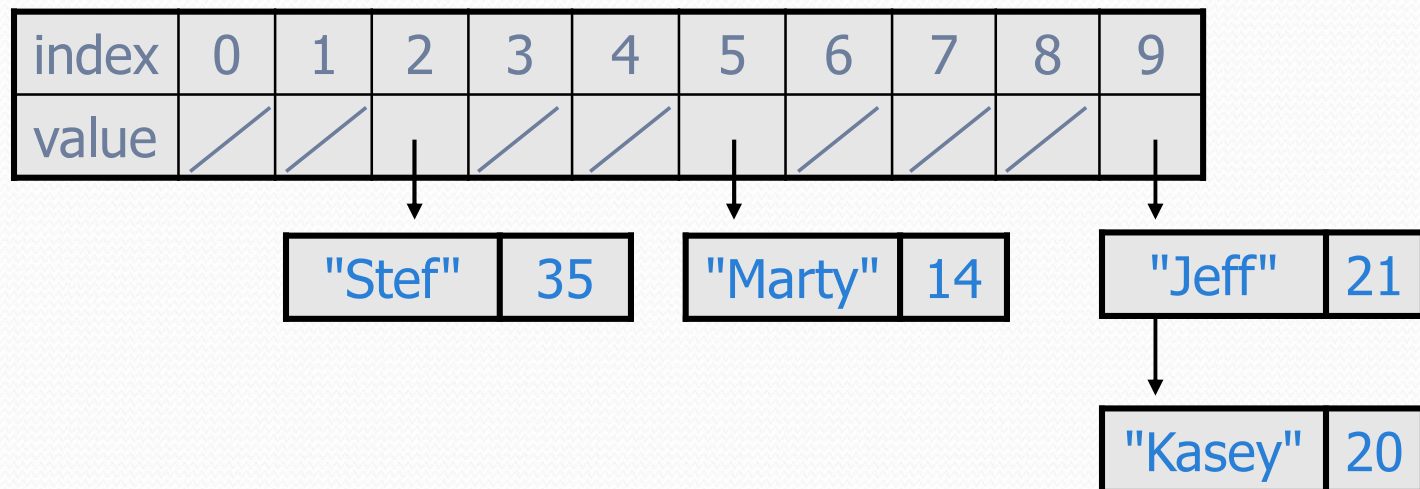
- How would we implement `toString` on a `HashSet`?



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>`