CSE 373

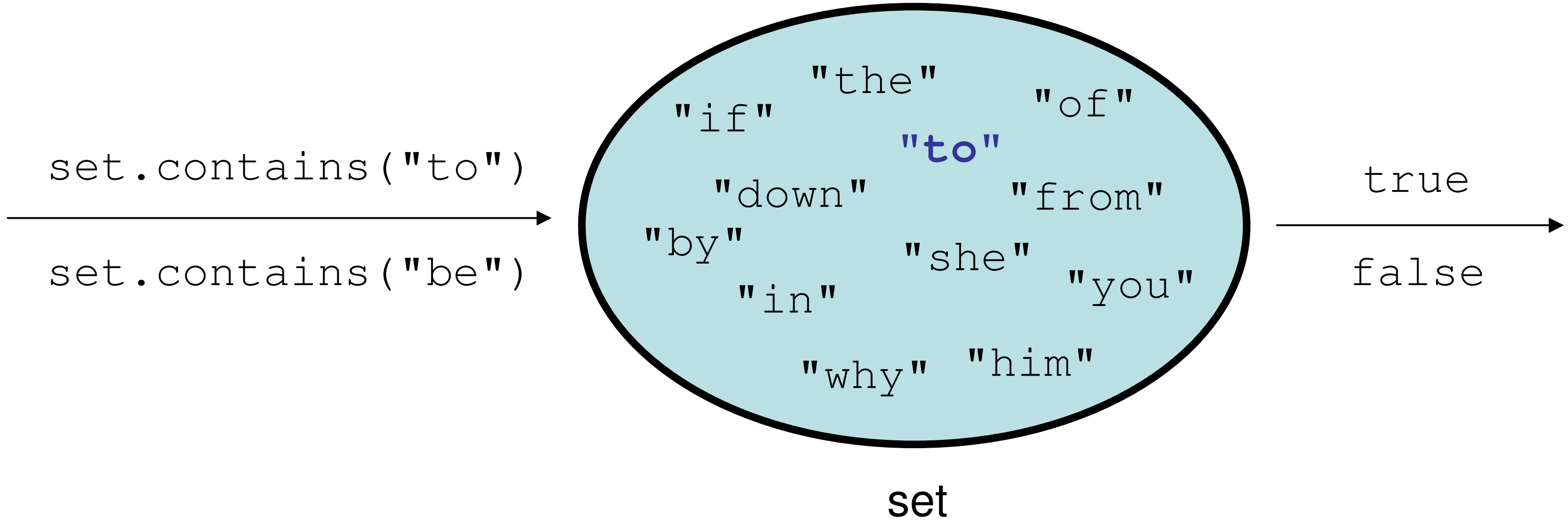
Set implementation; intro to hashing

read: Weiss 5.1 - 5.2, 5.4, 5.5

slides created by Marty Stepp
http://www.cs.washington.edu/373/

Sets

- **set**: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - add, remove, search (contains)
  - The client doesn't think of a set as having indexes; we just add things to the set in general and don't worry about order



Int Set ADT interface

- Let's think about how to write our own implementation of a set.
  - To simplify the problem, we only store `int`s in our set for now.
  - As is (usually) done in the Java Collection Framework, we will define sets as an ADT by creating a Set interface.
  - Core operations are: add, contains, remove.

```
public interface IntSet {
    void add(int value);
    boolean contains(int value);
    void clear();
    boolean isEmpty();
    void remove(int value);
    int size();
}
```

Unfilled array set

- Consider storing a set in an unfilled array.
  - It doesn't really matter what order the elements appear in a set, so long as they can be added and searched quickly.
  - What would make a good ordering for the elements?

- If we store them in the next available index, as in a list, ...

  -
    ```
    set.add(9);
    set.add(23);
    set.add(8);
    set.add(-3);
    set.add(49);
    set.add(12);
    ```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	9	23	8	-3	49	12	0	0	0	0
size	6									

  - How efficient is `add`? `contains`? `remove`?
    - O(1), O(*N*), O(*N*)
    - (`contains` must loop over the array; `remove` must shift elements.)

Sorted array set

- Suppose we store the elements in an unfilled array, but in *sorted* order rather than order of insertion.

  -
    ```
    set.add(9);
    set.add(23);
    set.add(8);
    set.add(-3);
    set.add(49);
    set.add(12);
    ```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	-3	8	9	12	23	49	0	0	0	0
size	6									

  - How efficient is `add`? `contains`? `remove`?
    - O(*N*), O(log *N*), O(*N*)
    - (You can do an O(log *N*) binary search to find elements in `contains`, and to find the proper index in `add`/`remove`; but `add`/`remove` still need to shift elements right/left to make room, which is O(*N*) on average.)

A strange idea

- *Silly idea:* When client adds value `i`, store it at index `i` in the array.
  - Would this work?
  - Problems / drawbacks of this approach? How to work around them?

```
set.add(7);
set.add(1);
set.add(9);
...
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	1	0	0	0	0	0	7	0	9
size	3									

```
set.add(18);
set.add(12);
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	1	0	0	0	0	0	7	0	9	0	0	12	0	0	0	0	0	18	0
size	5																			

Hashing

- **hash**: To map a large domain of values to a smaller fixed domain.
  - Typically, mapping a set of elements to integer indexes in an array.
  - *Idea:* Store any given element value in a particular predictable index.
    - That way, adding / removing / looking for it are constant-time (O(1)).
  - **hash table**: An array that stores elements via hashing.

- **hash function**: An algorithm that maps values to indexes.
  - **hash code**: The output of a hash function for a given value.

  - In previous slide, our "hash function" was: **hash(i)** → **i**
    - Potentially requires a large array (a.length > i).
    - Doesn't work for negative numbers.
    - Array could be very sparse, mostly empty (memory waste).

Improved hash function

- *To deal with negative numbers:* **hash(i) → abs(i)**
- *To deal with large numbers:* **hash(i) → abs(i) `%` length**

```
set.add(37);       // abs(37) % 10 == 7
set.add(-2);       // abs(-2) % 10 == 2
set.add(49);       // abs(49) % 10 == 9
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	0	-2	0	0	0	0	37	0	49
size	3									

```
// inside HashIntSet class
private int hash(int i) {
    return Math.abs(i) % elements.length;
}
```

Sketch of implementation

```
public class HashIntSet implements IntSet {
    private int[] elements;
    ...
    public void add(int value) {
        elements[hash(value)] = value;
    }

    public boolean contains(int value) {
        return elements[hash(value)] == value;
    }

    public void remove(int value) {
        elements[hash(value)] = 0;
    }
}
```

- Runtime of `add`, `contains`, and `remove`: **O(1) !!**
  - Are there any problems with this approach?

Collisions

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

```
set.add(11);
set.add(49);
set.add(24);
set.add(37);
set.add(54);   // collides with 24!
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	11	0	0	54	0	0	37	0	49
size	5									

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

Probing

- **probing**: Resolving a collision by moving to another index.
  - **linear probing**: Moves to the next available index (wraps if needed).

```
set.add(11);
set.add(49);
set.add(24);
set.add(37);
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	37	0	49
size	5									

  - *variation:* **quadratic probing** moves increasingly far away: +1, +4, +9, ...

Implementing HashIntSet

- Let's implement an `int` set using a hash table with linear probing.
  - For simplicity, assume that the set cannot store 0s for now.

```
public class HashIntSet implements IntSet {
    private int[] elements;
    private int size;

    // constructs new empty set
    public HashIntSet() {
        elements = new int[10];
        size = 0;
    }

    // hash function maps values to indexes
    private int hash(int value) {
        return Math.abs(value) % elements.length;
    }
    ...
```

The add operation

- How do we add an element to the hash table?
  - Use the hash function to find the proper bucket index.
  - If we see a 0, put it there.
  - If not, move forward until we find an empty (0) index to store it.
  - If we see that the value is already in the table, don't re-add it.

  - `set.add(54);    // client code`
  - `set.add(14);`

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	11	0	0	24	54	14	37	0	49
size	6									

Implementing add

- How do we add an element to the hash table?

```
public void add(int value) {
    int h = hash(value);
    while (elements[h] != 0 &&
           elements[h] != value) {      // linear probing
        h = (h + 1) % elements.length;  // for empty slot
    }
    if (elements[h] != value) {         // avoid duplicates
        elements[h] = value;
        size++;
    }
}
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	11	0	0	24	54	0	37	0	49
size	5									

The contains operation

- How do we search for an element in the hash table?
  - Use the hash function to find the proper bucket index.
  - Loop forward until we either find the value, or an empty index (0).
  - If find the value, it is contained (`true`). If we find 0, it is not (`false`).

  - `set.contains(24)    // true`
  - `set.contains(14)    // true`
  - `set.contains(35)    // false`

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	11	0	0	24	54	14	37	0	49
size	6									

Implementing contains

```
public boolean contains(int value) {
    int h = hash(value);
    while (elements[h] != 0) {
        if (elements[h] == value) {     // linear probing
            return true;                // to search
        }
        h = (h + 1) % elements.length;
    }
    return false;                       // not found
}
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	11	0	0	24	54	0	37	0	49
size	5									