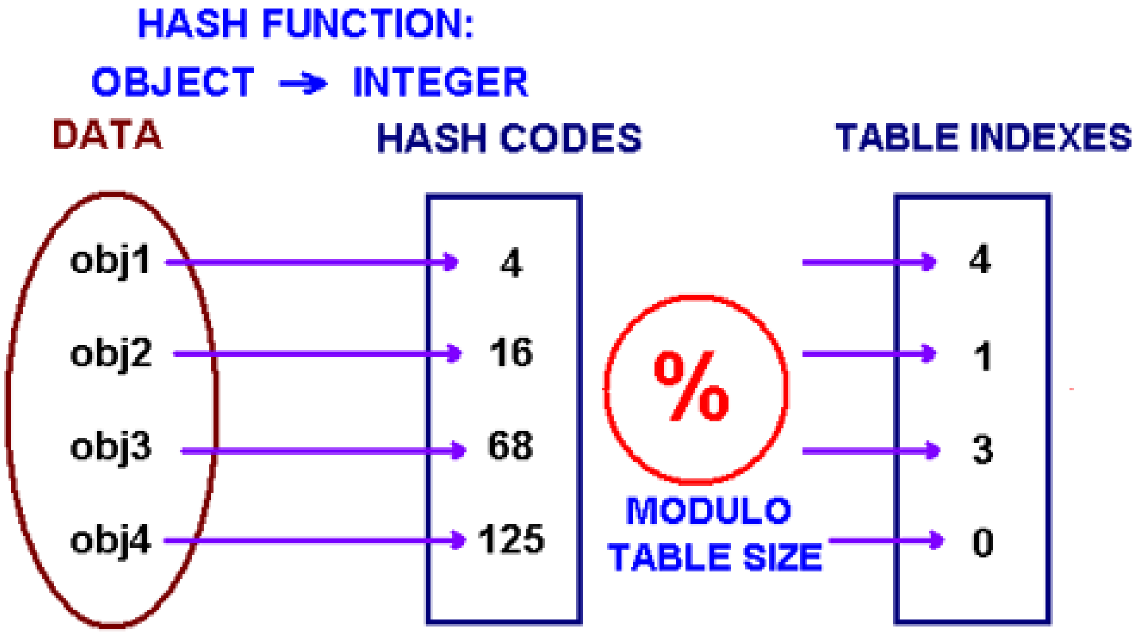
Hashing



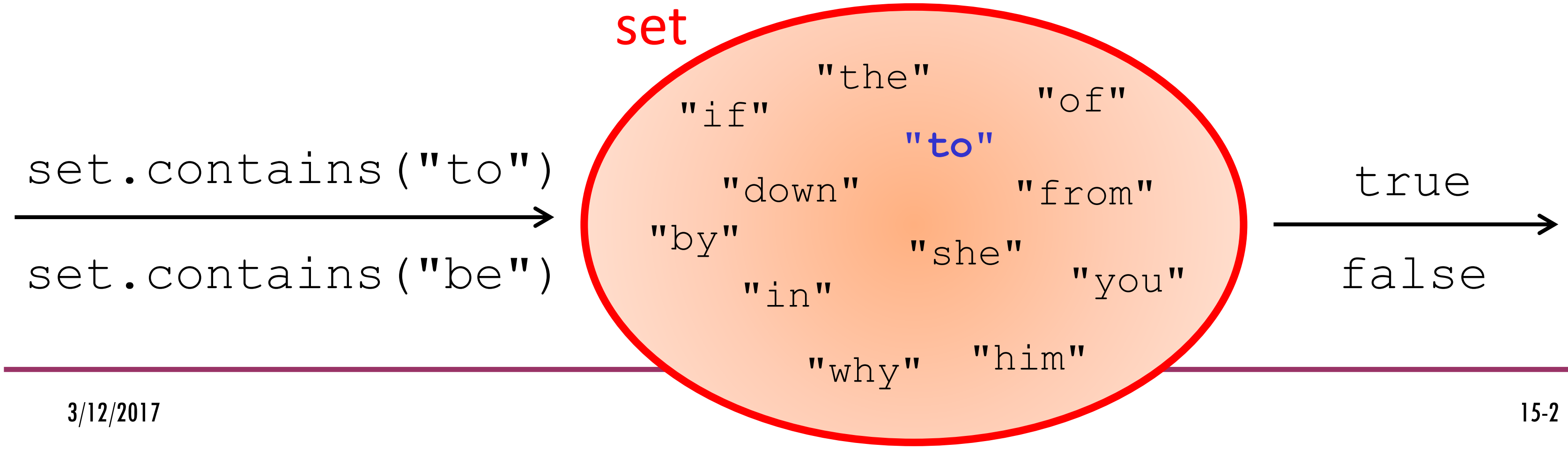
HASH TABLE

OBJ4	OBJ2		OBJ3	OBJ1
0	1	2	3	4

What do you know about Set?

- A collection of unique values (no duplicates allowed).
- We do not think of a set as having indexes; we just add things to the set in general and do NOT worry about order
- **Why do we need Sets?**
  - It models the mathematical set abstraction.
  - Can perform the following operations **efficiently**:

add, remove, search (contains)



set

`set.contains("to")`

`set.contains("be")`

"the" "of" "if" "to" "down" "from" "by" "she" "you" "in" "why" "him"

true

false

`IntegerSet` ADT interface

- Let's implement

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

What is our GOAL for today?

`add`, **`contains`**, **`remove`** should be O(1)

→ Add and search quickly

Storing a set in unfilled array set

- Order of elements appearance in a set does NOT matter
- Any suggestions on how to store the elements?
- Where to store the next element?
- 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);
```

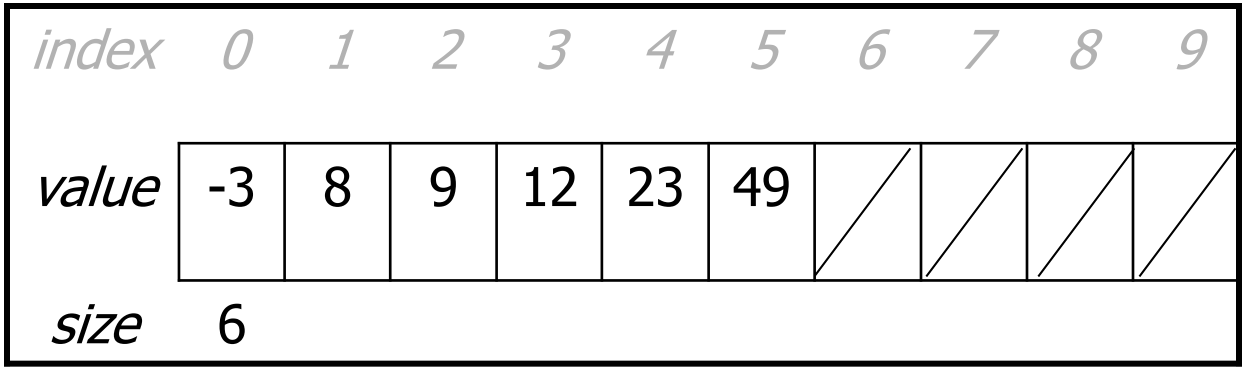
index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	9	23	8	-3	49	12	/	/	/	/
size	6									

- How efficient is `add`? `contains`? `remove`?
    - `add` – **O(1)** (if you assume there are no duplicates)
    - `contains` - O(*N*) loops over the array
    - `remove` - O(*N*) `contains` + shifts elements

Sorted array set

- What about *sorted* order (as opposed to order of insertion).

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



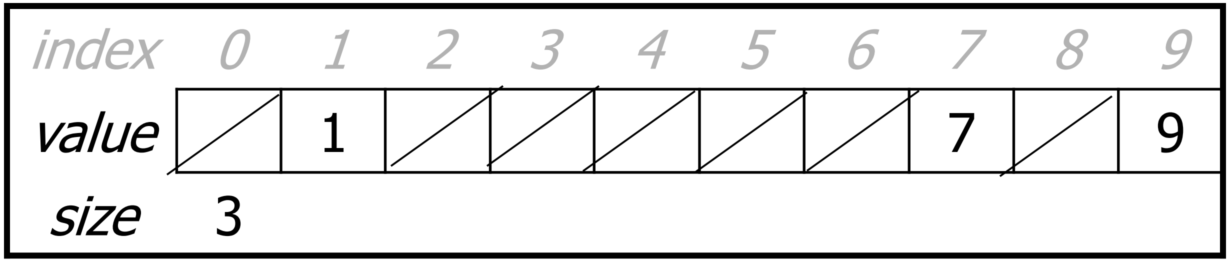
index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	-3	8	9	12	23	49				
size	6									

- How efficient is `add`? `contains`? `remove`?
- O(*N*), O(log *N*), O(*N*)
  - O(log *N*) - binary search to find elements (in `contains`, and to find the proper index in `add`/`remove`)
  - O(*N*) on average - in `add`/`remove` need to shift elements right/left to make room

A strange idea

- If value `i` is added → store it at index `i` in the array.
  - Would this work?

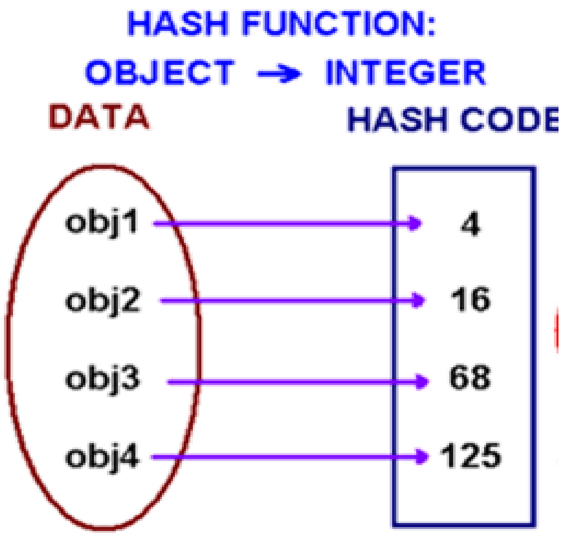
```
set.add(7);
set.add(1);
set.add(9);
```



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

Why is it useful?

- Elements are stored in a **predictable** index.
  - ➤`add`, `contains`, `remove` should be O(1)
- **hash table**: An **array** (why?) 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.



Hashing

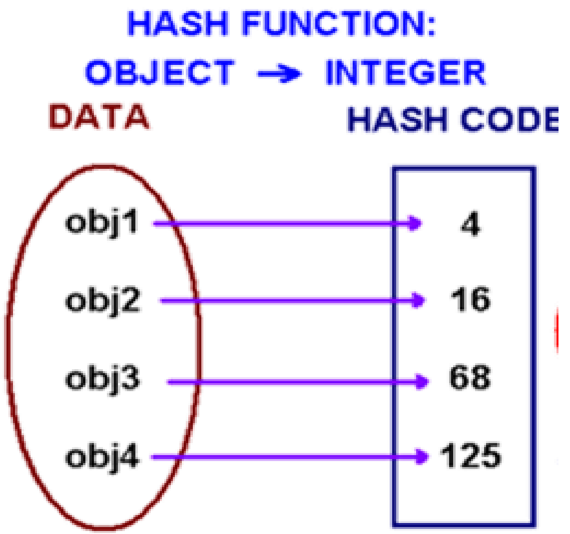
- In previous slide, hash function was:

hash(i) → i

Drawbacks:

- Potentially requires a large array (a.length > i).
- Does not work for negative numbers.
- Array could be very sparse (mostly empty)

→ memory waste.



Improved Hashing

- *For negative numbers:*

hash(i) → abs(i)

- *For large numbers:*

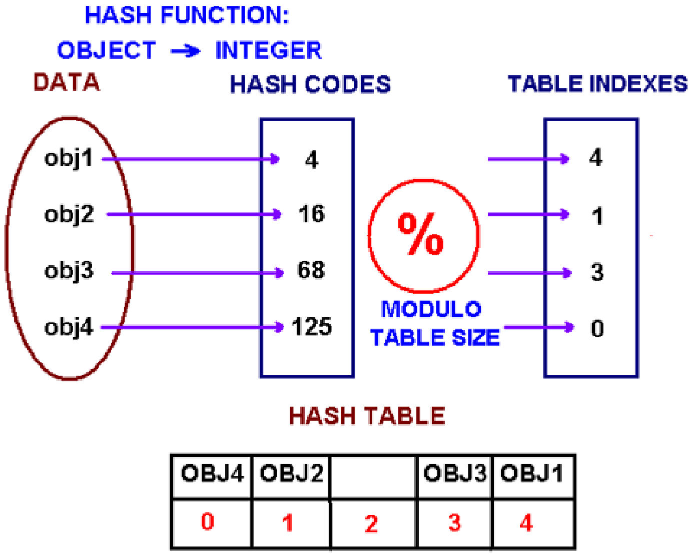
hash(i) → abs(i) % |table|

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

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

Is Table Size 10 Optimal?

Where will 20, 30, 40, … will be hashed ?



Primes

Usually better to use |table| =a prime number (like 13,27,31,…)

Why?

- Real-life data has patterns, which are **unlikely** to follow a prime sequence

For example: |table| =12 = 3*2*2

{0,12,24,36,…} → Common Factor is 12 → mapped to 0 (12 % 12)

{3,15,27,39,…} → Common Factor is 3 → mapped to 3

{6,18,30,42,…} → Common Factor is 6 → mapped to 6

Every hash code that has a common factor with |table| will be mapped to index that is a multiple of this factor (Greatest Common Factor)

- HOWEVER: If data IS uniformly distributed than primes are not crucial

Improved Hashing

- *For negative numbers:*

hash(i) → abs(i)

- *For large numbers:*

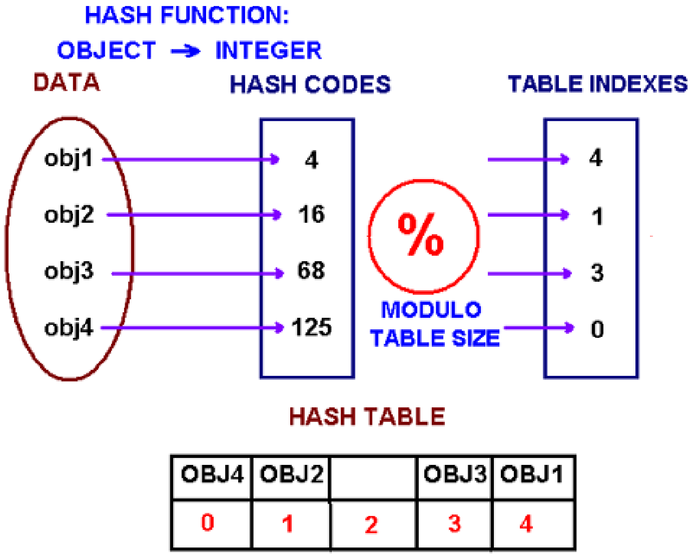
hash(i) → abs(i) % |table|

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

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

What is **hash(i)** for more complex Objects?

hash(i) → i.hashCode() % |table|



hashCode()

- Ideally, hashing function would have the following properties:

Uniform Distribution of Outputs:

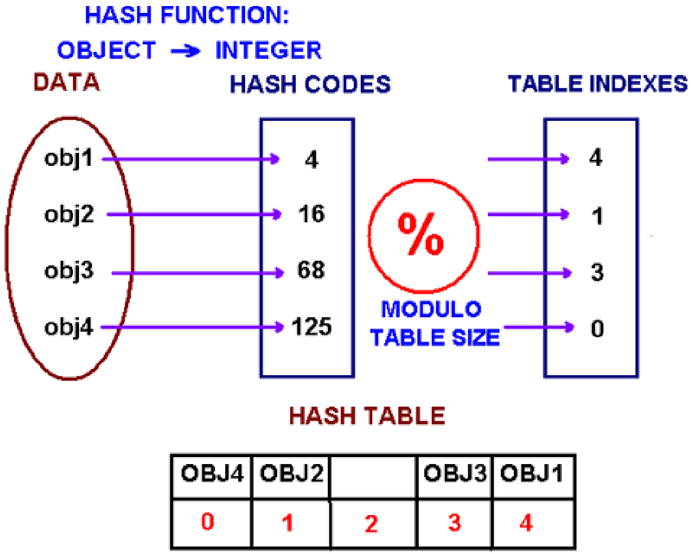
- We want unique hash code for different objects
- We do NOT want to collide all objects into the same hash bucket.
- There are 2^{32} 32-bit integers

→the probability that the hash function maps to any individual output should be $1/2^{32}$

Low Computational Cost:

hash function will be computed a lot;

→should be very easy to compute.



hashCode() Contract Reminder

- *Consistently with itself* (must produce same results on each call):

  `o.hashCode() == o.hashCode()`,

  if **`o`**'s state doesn't change

- *Consistently with equality*:

`a.equals(b)` must imply that `a.hashCode() == b.hashCode()`,

→ `equals` or `hashCode` should be overridden together

`!a.equals(b)` does NOT necessarily imply that

`a.hashCode() != b.hashCode()` *(why not?)*

If you do NOT override than

usually `hashCode()` method defined by class `Object` returns distinct integers for distinct objects (implemented by converting the internal address of the object into an integer).

Some Ideas for Hash functions for Strings

- $h(s_0 s_1 \dots s_{m-1}) = 1$

→**fast**, but **everything is mapped to the same index**

- $h(s_0 s_1 \dots s_{m-1}) = \sum_{i=0}^{m-1} s_i$

→ **ignores crucial information about the string**: the positions of the characters.

- $h(s_0 s_1 \dots s_{m-1}) = \sum_{i=0}^{m-1} 31^i s_i$

→**a nice compromise: all information about the String is used**, but **might sometimes map to the same index**

Mid-way Summary
Sketch of implementation

```
public class HashIntegerSet implements IntegerSet {
    private Integer[] elements;
    ...
    public void add(Integer value) {
        elements[hash(value)] = value;
    }

    public boolean contains(Integer value) {
        return elements[hash(value)]!=null &&
               elements[hash(value)].equals(value);
    }
    public void remove(Integer value) {
        elements[hash(value)] = null;
    }
}
```

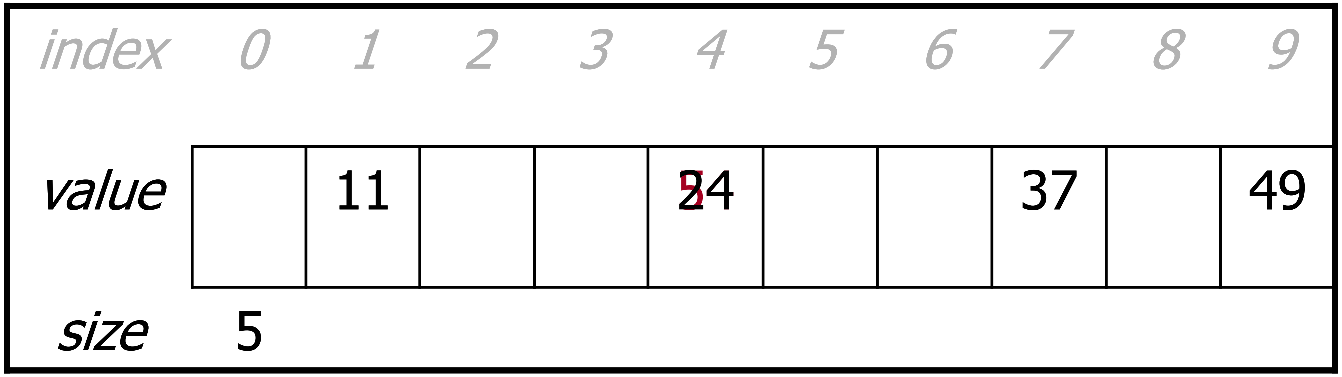
- Runtime of `add`, `contains`, and `remove`: **O(1) !!**

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		11			24			37		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		11			24	54		37		49
size	5									

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

Implementing HashIntegerSet using hash table and linear probing

```
public class HashIntegerSet implements IntegerSet {
    private Integer[] elements;
    private int size;

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

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

The add operation

- Use the hash function to find the proper bucket index.
- If we see a `null` (empty bucket) → put it there.
- If not, move forward until we find an empty (`null`) index to store it.
- If the value is already in the table

→ do NOT re-add it (WHY?)

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

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

Implementing add

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

}
```

The contains operation

- Use the hash function to find the proper bucket index.
- Loop forward until the value is found, or an empty index (`null`).
- If the value is found → return `true`
- If null is found → return `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		11			24	54	14	37		49
size	6									

Implementing contains

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

The remove operation

- We cannot remove by simply zeroing out an element (WHY?):

```
set.remove(54);     // set index 5 to 0
set.contains(14)    // false???  oops
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		11			24	**null**	14	34		49
size	5									

- Instead, we replace it by a special "removed" placeholder value
  - (can be re-used on `add`, but keep searching on `contains`)

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		11			24	XX	14	34		49
size	5									

Implementing remove

```
public void remove(Integer value) {
    int h = hash(value);
    while (elements[h] != null && !elements[h].equals(value)) {
         h = (h + 1) % elements.length;
    }
    if (elements[h] != null) {
        elements[h] = -999;    // "removed" flag value
        size--;
    }
}
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value		11			24	-999	14	34		49

size 5

```
set.remove(54);    // client code
set.remove(11);
set.remove(34);
```

Patching add, contains

```
private static final Integer REMOVED = -999;

public void add(Integer value) {
    int h = hash(value);
    while (elements[h] != null && !elements[h].equals(value) &&
           !elements[h].equals(REMOVED)) {
        h = (h + 1) % elements.length;
    }
    if (elements[h] == null || elements[h].equals(REMOVED)) {
        elements[h] = value;
        size++;
    }
}

// contains does not need patching;
// it should keep going on a -999, which it already does
public boolean contains(Integer value) {
    int h = hash(value);
    while (elements[h] != null && !elements[h].equals(value)) {
        h = (h + 1) % elements.length;
    }
    return elements[h] != null;
}
```

Mid - way Summary

- Indexing by the key needs too much memory
- Index into smaller size array may yield collisions

  → **probing** → try different array locations

  Complexity:

- `add` → scans till the **next** open spot. The **worst case** is **O(n)**

- What happens if the array is full?

Problem: full array

- **clustering**: Clumps of elements at neighboring indexes:

→must loop through them.

→slows down the hash table lookup

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	95	11			24	54	14	37	66	48
size	8									

```
set.add(11);
set.add(49);
set.add(24);
set.add(37);
set.add(54);  // collides with 24
set.add(14);  // collides with 24, then 54
set.add(86);  // collides with 14, then 37
```

- Where will you put 86?
- How many indexes must be examined to answer `contains(94)`?
- What will happen if the array is full?

Rehashing

- **rehash**: Growing to a larger array when the table is TOO full.
- **load factor**: ratio of (*# of elements*) / (*hash table length*)
  - many collections **rehash when load factor ≅ .75**

index	0	1	2	3	4	5	6	7	8	9
value	95	11			24	54	14	37	66	48
size	8									

  - Cannot simply copy the old array to a new one. (Why not?)

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
value					24		66		48			11			54	95	14	37		
size	8																			

Rehash Cost

- No profound algorithm: re-insert each item
- Linear time O(N)
- Insert still costs O(1)

```
// Grows hash table to twice its original size.
   private void rehash() {
       Integer[] old = elements;
       elements = new Integer[2 * old.length];
       size = 0;
       for (Integer value : old) {
           if (value != null && !value.equals(REMOVED)) {
               add(value);
           }
       }
   }
```

Implementing rehash

```
// Grows hash table to twice its original size.
private void rehash() {
     Integer[] old = elements;
    elements = new Integer[2 * old.length];
    size = 0;
    for (Integer value : old) {
        if (value != null && !value.equals(REMOVED)) {
            add(value);
        }
    }
}

public void add(Integer value) {
    if ((double) size / elements.length >= 0.75) {
        rehash();
    }
    ...
}
```

Recall Hash table size discussion

Use **prime numbers** as hash table sizes to

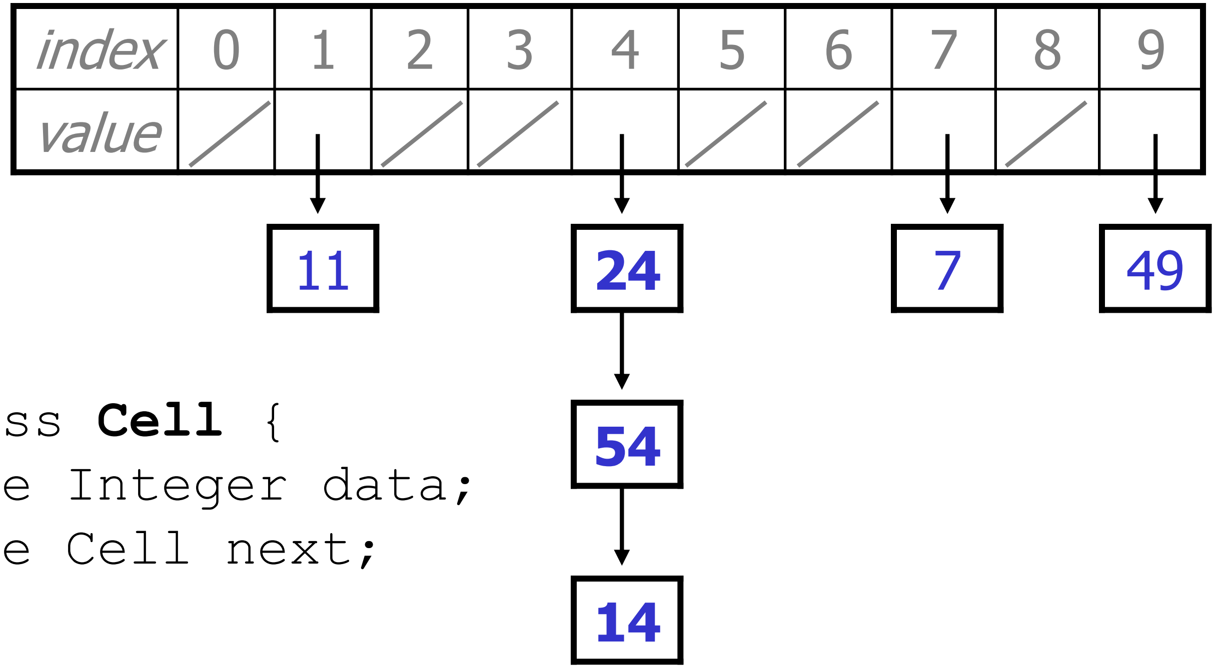
- reduce collisions
- improve spread / reduce clustering on rehash.

```
set.add(11);  //  11 % 13 == 11
set.add(39);  //  39 % 13 ==  0
set.add(21);  //  21 % 13 ==  8
set.add(29);  //  29 % 13 ==  3
set.add(71);  //  81 % 13 ==  6
set.add(41);  //  41 % 13 ==  2
set.add(99);  // 101 % 13 == 10
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
value	39		41	29			71		21		101	11	
size	7												

Separate chaining

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



```
public class Cell {
    private Integer data;
    private Cell next;
    ...
}
```

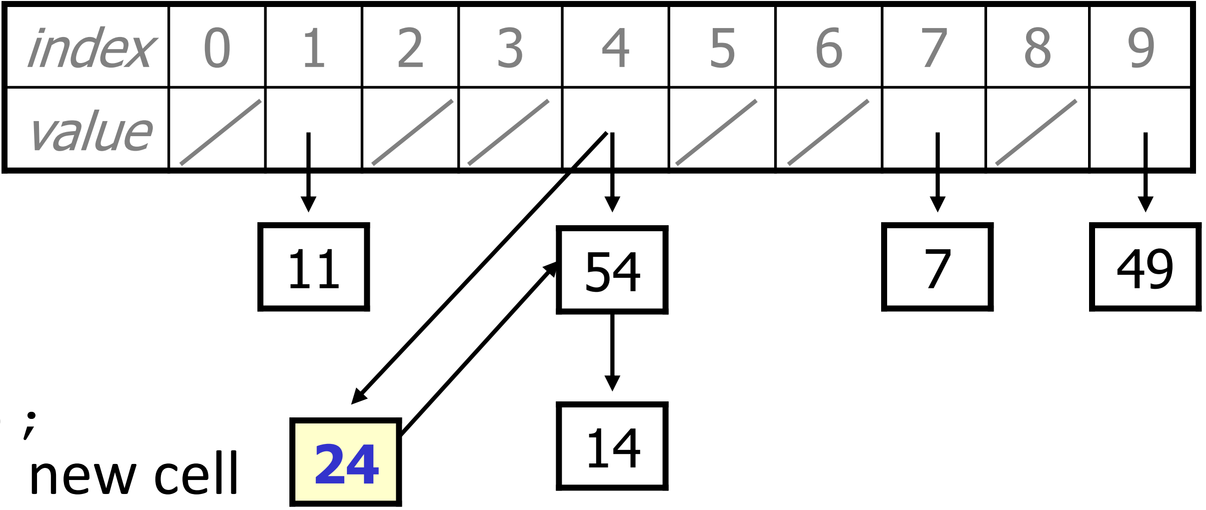
Implementing HashIntegerSet using separate chaining

```
public class HashIntegerSet implements IntegerSet
  {
    // array of linked lists;
    // elements[i] = front of list #i (null if
  empty)
    private Cell[] elements;
    private int size;

    // constructs new empty set
    public HashIntegerSet() {
        elements = new Cell[10];
        size = 0;
    }
    // hash function maps values to indexes
    // We do NOT use here the Integer hashCode()
    private int hash(Integer value) {
        return Math.abs(value) % elements.length;
    }
    ...
}
```

The add operation

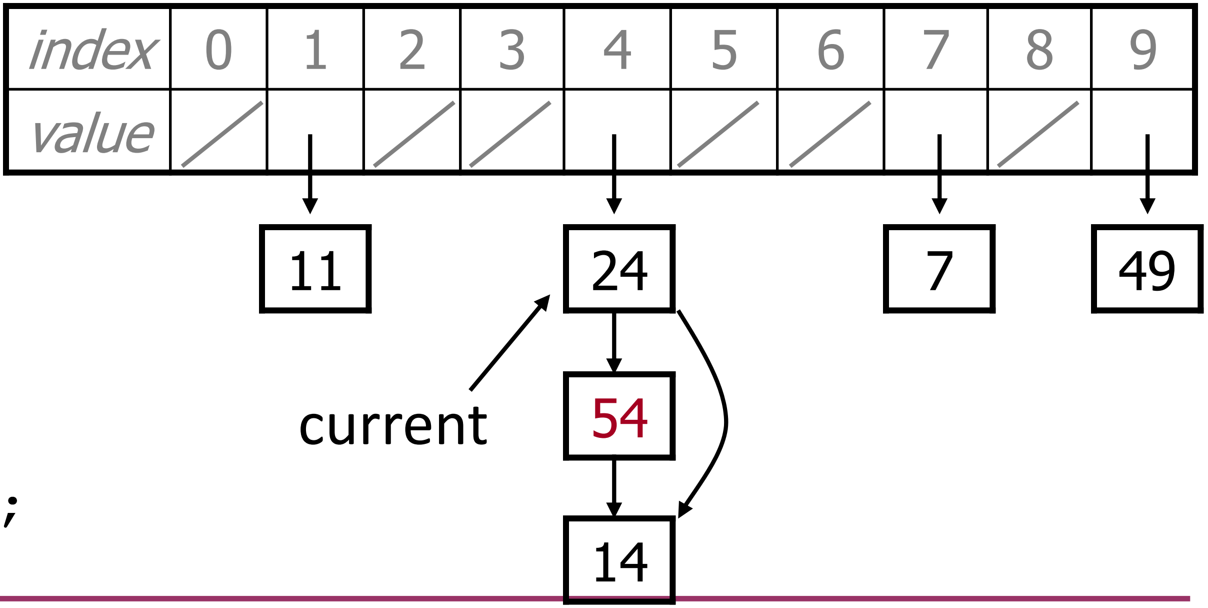
- How do we add an element to the hash table?
  - Modification of a linked list change can be done by
    - the list's `head` reference
    - or the `next` field of a node in the list.
  - Where/when should we add the new element?
  - Must make sure to avoid duplicates.



- `set.add(24);`

The remove operation

- How do we remove an element from the hash table?
  - Cases to consider:
    - `head` (24),
    - non-head (14),
    - not found (94),
    - `null` (32)



```
set.remove(54);
```

Summary

- Indexing by the key needs too much memory
- Index into smaller size array, may yield collisions
- If collisions occur
  - probing → try different array locations (linear, quadratic)
  - separate chaining, lists in array
- If the array is full (load factor too high) → slows performances
- For quadratic probing, insert may fail if load > 1/2
  - We can rehash as soon as load > 1/2
  - Or, we can rehash only when insert fails

→ Heuristically choose a load factor threshold, rehash when threshold breached