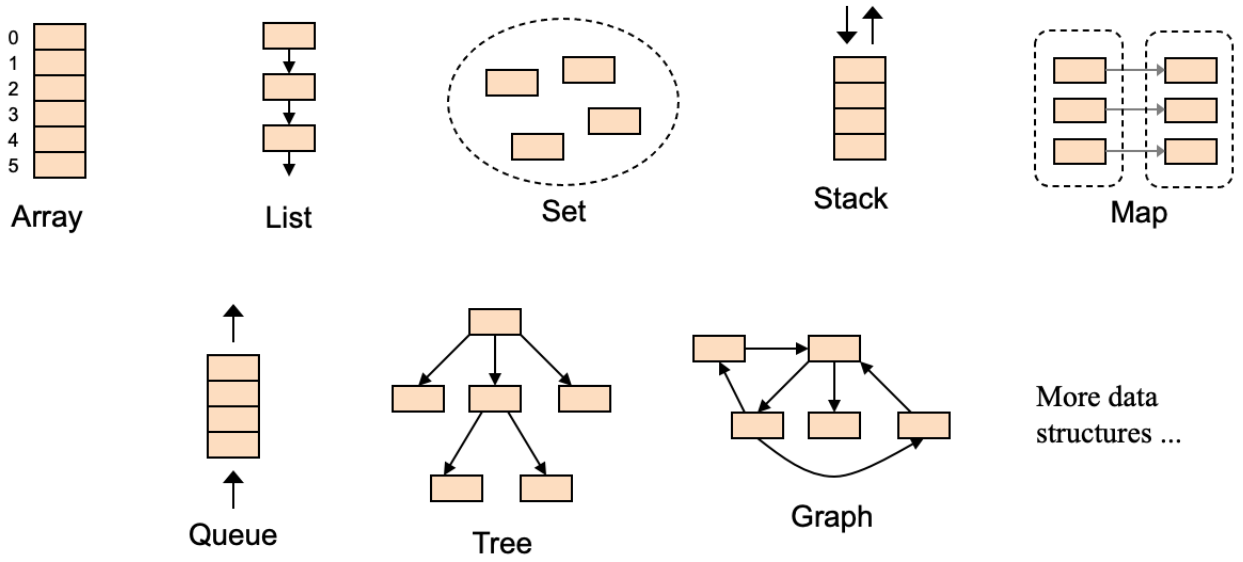
Efi Arazi School of Computer Science
Introduction to Computer Science

Lecture 10-2

Data Structures II

Data structures

Data structure: A way of organizing, processing, storing, and retrieving data. Different data structures are suited to different kinds of applications, and different kinds of efficiency.

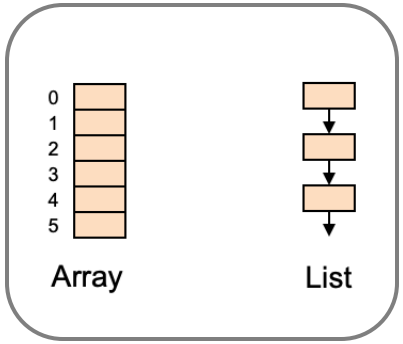


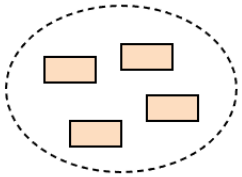
- All widely used in computer science
- All can be implemented using *arrays* or *lists*

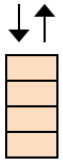
Data structures

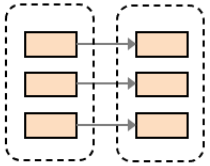
Data structure: A way of organizing, processing, storing, and retrieving data. Different data structures are suited to different kinds of applications, and different kinds of efficiency.

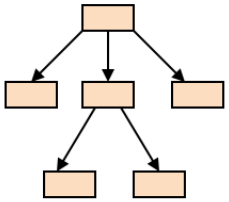
Previous lectures

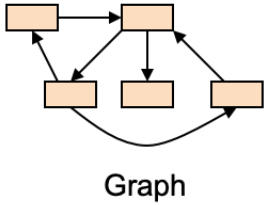










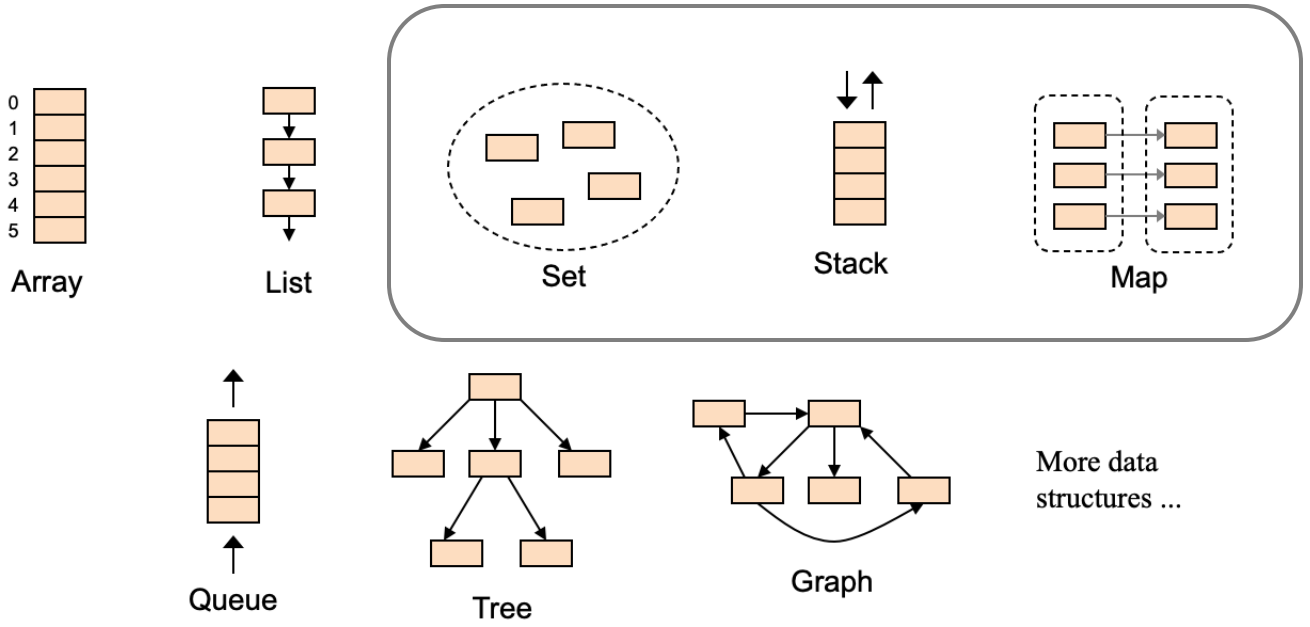


More data structures ...

- All widely used in computer science
- All can be implemented using *arrays* or *lists*.

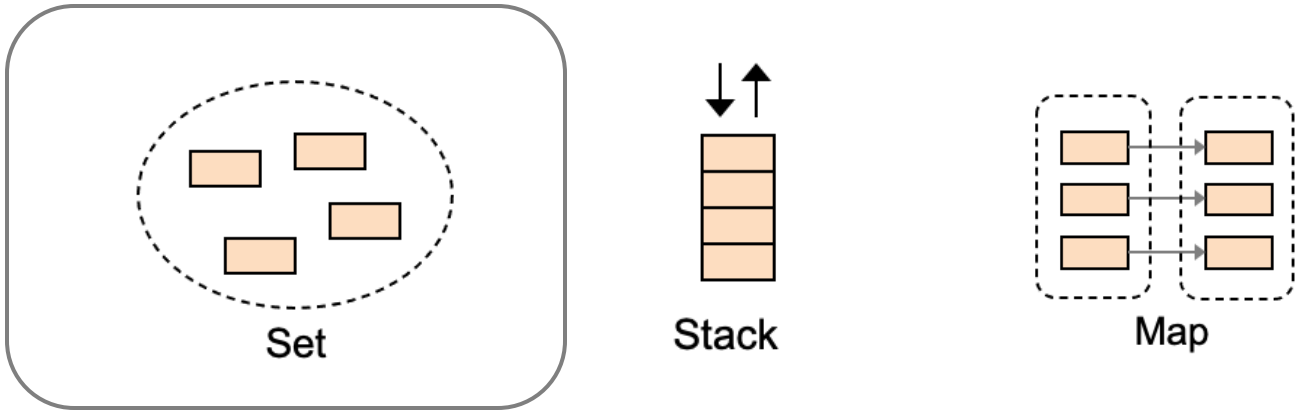
Data structures

Data structure: A way of organizing, processing, storing, and retrieving data. Different data structures are suited to different kinds of applications, and different kinds of efficiency.



- All widely used in computer science
- All can be implemented using *arrays* or *lists*

Lecture plan



Methodology

Abstraction: How to *use* the data structure

Implementation: How to *realize* the data structure (using arrays / lists)

Set abstraction

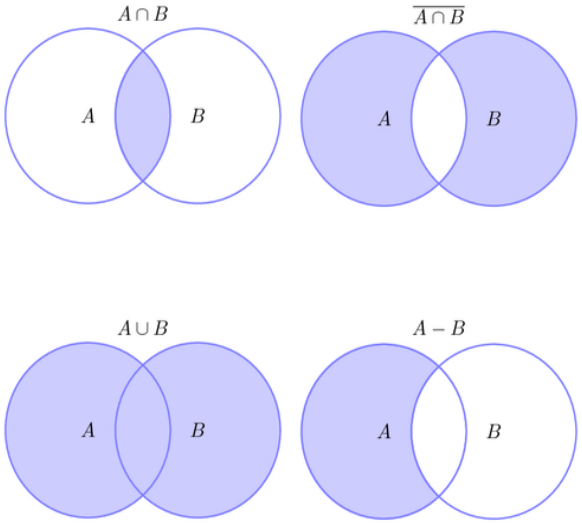
What makes a collection of elements a *set* ?

- No order

  For example, {4, 3, 5} and {3, 4, 5} are same set

- No duplication

  For example, {4, 3, 5, 4} is not a valid set;
  To make it a set, we must reduce it to {4, 3, 5}



Basic operations (for any collection)

- Create an empty set
- Add elements
- Check if the set contains an element

  …

Set specific operations

- Intersection
- Union
- Difference
- Subset / superset

  …

Set abstraction

API

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    /** Constructs an empty set. */
    public Set()

     /** Creates a set containing the elements in the given array. */
    public Set(int[] arr)

    /** Creates a set containing the elements in the given set. */
    public Set(Set set)

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Returns a set which is the union of this set and the other set. */
    public Set union(Set other)

    /** Returns a set which is the intersection of this set and the other set. */
    public Set intersection(Set other)

    /** Returns a string representing this set in the form of
     *  {e1, e2, e3, ...}, where the e's are the set elements. */
    public String toString()
    ...
}
```

Basic operations

- Create an empty set
- Add elements
- Check if the set contains an element

  …

Set specific operations

- Intersection
- Union
- Difference
- Subset / superset

  …

Set abstraction

API

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    /** Constructs an empty set. */
    public Set()

     /** Creates a set containing the elements in the given array. */
    public Set(int[] arr)

    /** Creates a set containing the elements in the given set. */
    public Set(Set set)

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Returns a set which is the union of this set and the other set. */
    public Set union(Set other)

    /** Returns a set which is the intersection of this set and the other set. */
    public Set intersection(Set other)

    /** Returns a string representing this set in the form of
     *  {e1, e2, e3, ...}, where the e's are the set elements. */
    public String toString()
    ...
}
```

```
// Some class (client code)
Set s1 = new Set();
s1.add(3); s1.add(1); s1.add(3); s1.add(5); s1.add(7);
System.out.println("s1 = " + s1);

System.out.println(s1.contains(3));
System.out.println(s1.contains(4));
```

```
// Output
s1 = {3, 1, 5, 7}
true
false
```

Let's open the black box...

Set representation

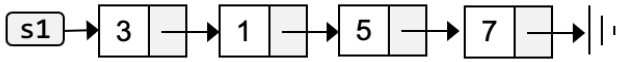
```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set() {
        elements = new List();
    }

    ...
```

The set elements are implemented as a `List` object

```
// Some class (client code)
Set s1 = new Set();
s1.add(3); s1.add(1); s1.add(3); s1.add(5); s1.add(7);
System.out.println("s1 = " + s1);

System.out.println(s1.contains(3));
System.out.println(s1.contains(4));
```

```
// Output
s1 = {3, 1, 5, 7}
true
false
```

Behind the scene:



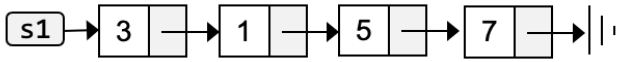
Contains: Abstraction

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set() {
        elements = new List();
    }

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)
```

```
// Some class (client code)
Set s1 = new Set();
s1.add(3); s1.add(1); s1.add(3); s1.add(5); s1.add(7);
System.out.println("s1 = " + s1);

System.out.println(s1.contains(3));
System.out.println(s1.contains(4));
```

```
// Output
s1 = {3, 1, 5, 7}
true
false
```

Behind the scene:



Contains: Implementation

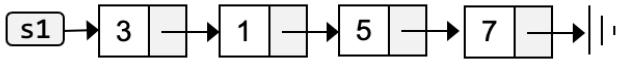
```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set() {
        elements = new List();
    }

    /** Checks if the given element exists in this set. */
    public boolean contains(int e) {
        return (elements.indexOf(e) > -1);
    }
```

since `elements` is a `List`, we can process it using `List` methods, like `indexOf`

```
// Some class (client code)
Set s1 = new Set();
s1.add(3); s1.add(1); s1.add(3); s1.add(5); s1.add(7);
System.out.println("s1 = " + s1);

System.out.println(s1.contains(3));
System.out.println(s1.contains(4));
```

```
// Output
s1 = {3, 1, 5, 7}
true
false
```

Behind the scene:



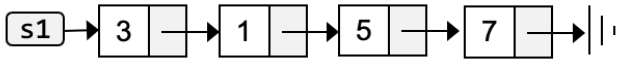
Adding elements: Abstraction

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set() {
        elements = new List();
    }

    /** Checks if the given element exists in this set. */
    public boolean contains(int e) {
        return (elements.indexOf(e) > -1);
    }

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)
```

```
// Some class (client code)
Set s1 = new Set();
s1.add(3); s1.add(1); s1.add(3); s1.add(5); s1.add(7);
System.out.println("s1 = " + s1);

System.out.println(s1.contains(3));
System.out.println(s1.contains(4));
```

```
// Output
s1 = {3, 1, 5, 7}
true
false
```

Behind the scene:



Adding elements: Implementation

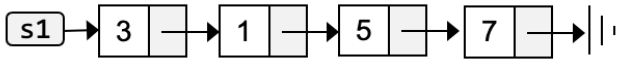
```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set() {
        elements = new List();
    }

    /** Checks if the given element exists in this set. */
    public boolean contains(int e) {
        return (elements.indexOf(e) > -1);
    }

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e) {
        if (!contains(e)) {
            elements.add(e);
        }
    }

    ...
```

```
// Some class (client code)
Set s1 = new Set();
s1.add(3); s1.add(1); s1.add(3); s1.add(5); s1.add(7);
System.out.println("s1 = " + s1);

System.out.println(s1.contains(3));
System.out.println(s1.contains(4));
```

Calls the `add` method of `Set`, which calls the `add` method of `List`...

```
// Output
s1 = {3, 1, 5, 7}
true
false
```

Behind the scene:



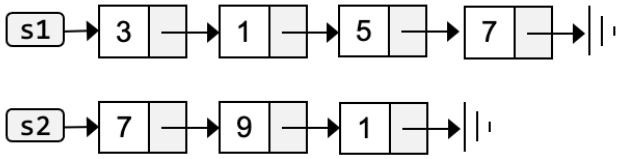
Constructing a set from an array: Abstraction

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set()

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Creates a set containing the elements in the given array. */
    public Set(int[] arr)
```

```
// Some class (client code)
Set s1 = new Set();
s1.add(3); s1.add(1); s1.add(3); s1.add(5); s1.add(7);
System.out.println("s1 = " + s1);

int[] data = {7, 9, 1, 9, 1, 7};
Set s2 = new Set(data);
System.out.println("s2 = " + s2);
```

```
// Output
s1 = {3, 1, 5, 7}
s2 = {7, 9, 1}
```

Behind the scene:



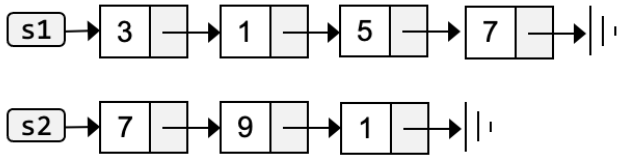
Constructing a set from an array: Implementation

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set()

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Creates a set containing the elements in the given array. */
    public Set(int[] arr) {
        this();
        // Iterates over the given array,
        //  and adds its elements to this set
        for (int e : arr) {
            add(e);
        }
    }

    ...
```

```
// Some class (client code)
Set s1 = new Set();
s1.add(3); s1.add(1); s1.add(3); s1.add(5); s1.add(7);
System.out.println("s1 = " + s1);

int[] data = {7, 9, 1, 9, 1, 7};
Set s2 = new Set(data);
System.out.println("s2 = " + s2);
```

```
// Output
s1 = {3, 1, 5, 7}
s2 = {7, 9, 1}
```

Behind the scene:



Aside: For Each (built-in Java iterator)

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set()

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Creates a set containing the elements in the given array. */
    public Set(int[] arr) {
        this();
        // Iterates over the given array,
        //  and adds its elements to this set
        for (int e : arr) {
            add(e);
        }
    }
```

```
// The above for-each code is the same as:
for (int i = 0; i < arr.length; i++) {
    add(arr[i]);
}
```

For-each rules

A built-in Java *iterator*;

Can be used under some restrictions:

The loop must iterate over *all* the elements;

The statements in the body of the loop cannot refer to specific elements;

The loop cannot modify any element;

Best practice

For-each is elegant and readable.

Whenever possible, use it!

Aside: For Each (built-in Java iterator)

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set()

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Creates a set containing the elements in the given array. */
    public Set(int[] arr) {
        this();
        // Iterates over the given array,
        //  and adds its elements to this set
        for (int e : arr) {
            add(e);
        }
    }
```

For-each rules

A built-in Java *iterator*;

Can be used under some restrictions:

The loop must iterate over *all* the elements;

The statements in the body of the loop cannot refer to specific elements;

The loop cannot modify any element;

Best practice

For-each is elegant and readable.

Whenever possible, use it!

Iterator's logic

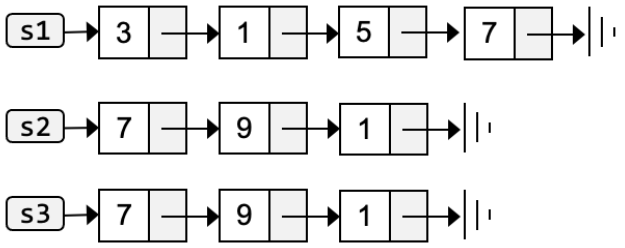
- As long as the iteration has a next element...
- Gets the current element (e), and advance the iteration.

Building a set from another set: Abstraction

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set()

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Creates a set containing the elements in the given array. */
    public Set(int[] arr) {

    /** Creates a set containing the elements in the other set. */
    public Set(Set other)
```

```
// Some class (client code)
Set s1 = new Set();
s1.add(3); s1.add(1); s1.add(3); s1.add(5); s1.add(7);
System.out.println("s1 = " + s1);

int[] data = {7, 9, 1, 9, 1, 7};
Set s2 = new Set(data);
System.out.println("s2 = " + s2);

Set s3 = new Set(s2);
System.out.println("s3 = " + s3);
```

```
// Output
s1 = {3, 1, 5, 7}
s2 = {7, 9, 1}
s3 = {7, 9, 1}
```



Building a set from another set: Implementation

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set()

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Creates a set containing the elements in the given array. */
    public Set(int[] arr) {

    /** Creates a set containing the elements in the other set. */
    public Set(Set other) {
        this();
        // Iterates over the given set and adds its elements to this set
        ListIterator itr = other.elements.listIterator();
        while (itr.hasNext()) {
            int e = itr.next();
            add(e);
        }
    }

    ...
```

```
// Some class (client code)
Set s1 = new Set();
s1.add(3); s1.add(1); s1.add(3); s1.add(5); s1.add(7);
System.out.println("s1 = " + s1);

int[] data = {7, 9, 1, 9, 1, 7};
Set s2 = new Set(data);
System.out.println("s2 = " + s2);

Set s3 = new Set(s2);
System.out.println("s3 = " + s3);
```

```
// Output
s1 = {3, 1, 5, 7}
s2 = {7, 9, 1}
s3 = {7, 9, 1}
```

Building a set from another set: Implementation

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set()

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Creates a set containing the elements in the given array. */
    public Set(int[] arr) {

    /** Creates a set containing the elements in the other set. */
    public Set(Set other) {
        this();
        // Iterates over the given set and adds its elements to this set
        ListIterator itr = other.elements.listIterator();
        while (itr.hasNext()) {
            int e = itr.next();
            add(e);
        }
    }

    ...
```

`listIterator()` is a `List` method;

It returns a `ListIterator` object, which acts as an iterator.

```
// Some class (client code)
Set s1 = new Set();
s1.add(3); s1.add(1); s1.add(3); s1.add(5); s1.add(7);
System.out.println("s1 = " + s1);

int[] data = {7, 9, 1, 9, 1, 7};
Set s2 = new Set(data);
System.out.println("s2 = " + s2);

Set s3 = new Set(s2);
System.out.println("s3 = " + s3);
```

```
// Output
s1 = {3, 1, 5, 7}
s2 = {7, 9, 1}
s3 = {7, 9, 1}
```

Building a set from another set: Implementation

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set()

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Creates a set containing the elements in the given array. */
    public Set(int[] arr) {

    /** Creates a set containing the elements in the other set. */
    public Set(Set other) {
        this();
        // Iterates over the given set and adds its elements to this set
        ListIterator itr = other.elements.listIterator();
        while (itr.hasNext()) {
            int e = itr.next();
            add(e);
        }
    }

    ...
```

This iterator is not built into Java;

We built it into the `List` class.

```
// Some class (client code)
Set s1 = new Set();
s1.add(3); s1.add(1); s1.add(3); s1.add(5); s1.add(7);
System.out.println("s1 = " + s1);

int[] data = {7, 9, 1, 9, 1, 7};
Set s2 = new Set(data);
System.out.println("s2 = " + s2);

Set s3 = new Set(s2);
System.out.println("s3 = " + s3);
```

```
// Output
s1 = {3, 1, 5, 7}
s2 = {7, 9, 1}
s3 = {7, 9, 1}
```

Same iteration logic:

- As long as `hasNext()` is true...
- `next()` returns the current element, and advance the iteration;

  (the term "`next`" is confusing, but commonly used in iterators)

Building a set from another set

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set()

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Creates a set containing the elements in the given array. */
    public Set(int[] arr) {

    /** Creates a set containing the elements in the other set. */
    public Set(Set other) {
        this();
        // Iterates over the given set and adds its elements to this set
        ListIterator itr = other.elements.listIterator();
        while (itr.hasNext()) {
            int e = itr.next();
            add(e);
        }
    }

    ...
```

Can we use *for-each* instead?

```
// Iterates over the given set...
for (int e : other.elements) {
    add(e);
}
```

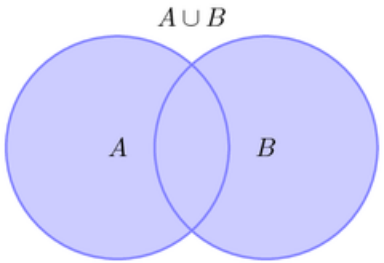
Yes! To do so, we must make the `List` class *iterable* (later in the course).

Set Union: Abstraction

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set()

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Creates a set containing the elements in the given array. */
    public Set(int[] arr) {

    /** Creates a set containing the elements in the given set. */
    public Set(Set set)

    /** Returns a set which is the union of this set and the other set. */
    public Set union(Set other) {
```

```
// Some class (client code)
Set s1 = new Set();
s1.add(3); s1.add(1); s1.add(3); s1.add(5); s1.add(7);
System.out.println("s2 = " + s2);
int[] data = {7, 9, 1, 9, 1, 7};
Set s2 = new Set(data);
System.out.println("s2 = " + s2);
System.out.println(s1.union(s2));
```

```
// Output
S1 = {3, 1, 5, 7}
S2 = {7, 9, 1}
{3, 1, 5, 7, 9}  // Union
```

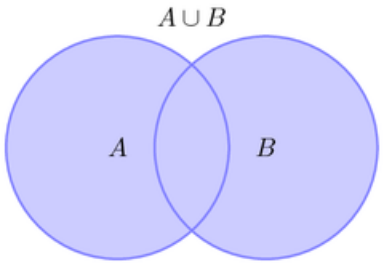


Set Union: Implementation

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set()

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Creates a set containing the elements in the given array. */
    public Set(int[] arr) {

    /** Creates a set containing the elements in the given set. */
    public Set(Set set)

    /** Returns a set which is the union of this set and the other set. */
    public Set union(Set other) {
        // Creates a result set with the elements of this set
        Set result = new Set(this);
        // Iterates over the other set and adds it elements to the result set
        ListIterator itr = other.elements.listIterator();
        while (itr.hasNext()) {
            int e = itr.next();
            result.add(e);
        }
        return result;
    }
```

```
// Some class (client code)
Set s1 = new Set();
s1.add(3); s1.add(1); s1.add(3); s1.add(5); s1.add(7);
System.out.println("s2 = " + s2);
int[] data = {7, 9, 1, 9, 1, 7};
Set s2 = new Set(data);
System.out.println("s2 = " + s2);
System.out.println(s1.union(s2));
```

```
// Output
S1 = {3, 1, 5, 7}
S2 = {7, 9, 1}
{3, 1, 5, 7, 9}  // Union
```

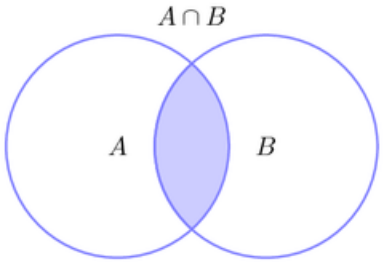


Set Intersection: Abstraction

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set()

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Creates a set containing the elements in the given array. */
    public Set(int[] arr) {

    /** Creates a set containing the elements in the given set. */
    public Set(Set set)

    /** Returns a set which is the intersection of this set and the other set. */
    public Set intersection(Set other) {
```

```
// Some class (client code)
Set s1 = new Set();
s1.add(3); s1.add(1); s1.add(3); s1.add(5); s1.add(7);
System.out.println("s2 = " + s2);
int[] data = {7, 9, 1, 9, 1, 7};
Set s2 = new Set(data);
System.out.println("s2 = " + s2);
System.out.println(s1.union(s2));
System.out.println(s1.intersection(s2));
```

```
// Output
S1 = {3, 1, 5, 7}
S2 = {7, 9, 1}
{3, 1, 5, 7, 9}  // Union
{1, 7}           // Intersection
```

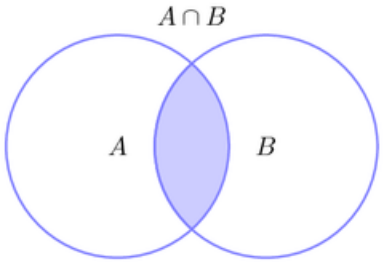


Set Intersection: Implementation

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    private List elements;    // The elements of this set

    /** Constructs an empty set. */
    public Set()

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Creates a set containing the elements in the given array. */
    public Set(int[] arr) {

    /** Creates a set containing the elements in the given set. */
    public Set(Set set)

    /** Returns a set which is the intersection of this set and the other set. */
    public Set intersection(Set other) {
        // Creates an empty result set
        Set result = new Set();
        // For each element in this set, if the other set contains it, adds it
        ListIterator itr = elements.listIterator();
        while (itr.hasNext()) {
            int e = itr.next();
            if (other.contains(e))
                result.add(e);
            }
        }
        return result;
    }
    ...
```

```
// Some class (client code)
Set s1 = new Set();
s1.add(3); s1.add(1); s1.add(3); s1.add(5); s1.add(7);
System.out.println("s2 = " + s2);
int[] data = {7, 9, 1, 9, 1, 7};
Set s2 = new Set(data);
System.out.println("s2 = " + s2);
System.out.println(s1.union(s2));
System.out.println(s1.intersection(s2));
```

```
// Output
S1 = {3, 1, 5, 7}
S2 = {7, 9, 1}
{3, 1, 5, 7, 9}  // Union
{1, 7}           // Intersection
```



Set: Recap

API

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    /** Constructs an empty set. */
    public Set()

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Creates a set containing the elements in the given array. */
    public Set(int[] arr)

    /** Creates a set containing the elements in the given set. */
    public Set(Set set) {

    /** Returns the union of this set and the other set. */
    public Set union(Set other)

    /** Returns the intersection of this set and the other set. */
    public Set intersection(Set other) {

    /** Returns a string representing this set in the form of
     *  {e1, e2, e3, ...}, where the e's are the set elements. */
    public String toString()
    ...
}
```

```
// Client code that creates and processes
// sets of integers
Set s1 = new Set();
s1.add(3); …

Set s2 = new Set();
s2.add(5); …

Set s3 = s1.union(s2));

Set s4 = s1.intersection(s2));
```

Using the `Set` class, any client program can create and manipulate sets;

The `Set` API provides a *domain language* for handling sets;

OOP can be viewed as a tool for designing domain languages for numerous domains.

Set: Challenge

API

```
/** Represents a set of integers.
 *  A set is a collection of values without repetition or order.
 *  The set has an unlimited size. */
public class Set {

    /** Constructs an empty set. */
    public Set()

    /** Checks if the given element exists in this set. */
    public boolean contains(int e)

    /** If the given element is not in this set, adds it to this set.
     *  Otherwise, does nothing. */
    public void add(int e)

    /** Creates a set containing the elements in the given array. */
    public Set(int[] arr)

    /** Creates a set containing the elements in the given set. */
    public Set(Set set) {

    /** Returns the union of this set and the other set. */
    public Set union(Set other)

    /** Returns the intersection of this set and the other set. */
    public Set intersection(Set other) {

    /** Returns a string representing this set in the form of
     *  {e1, e2, e3, ...}, where the e's are the set elements. */
    public String toString()
    ...
}
```

Big problem, Big solution

The `List` class, as defined, can handle only `int` values. Therefore, any class that depends on it, like `Set`, can also handle only `int` values.

Can we make `List`, and any class that depends on it, handle objects of any type?

Yes!

To do so, we must make these classes *generic* (later in the course).

Set: Efficiency

In a set of N elements:

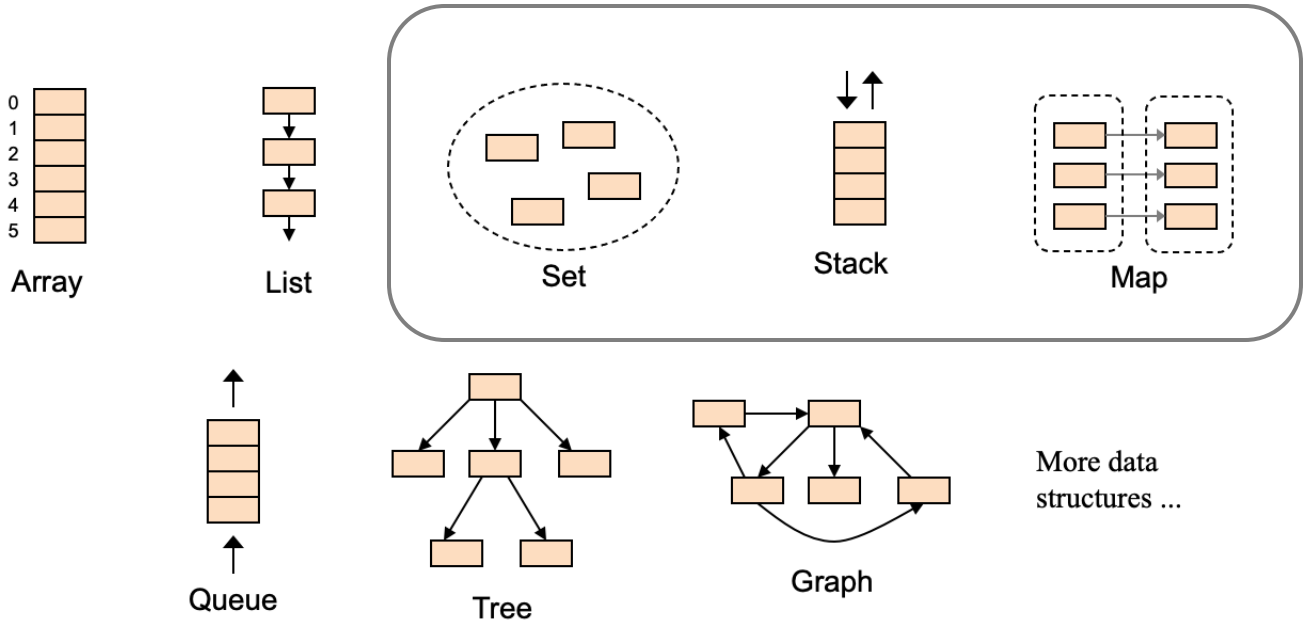
Add: $O(N)$

Union: $O(N^2)$

Intersection: $O(N^2)$

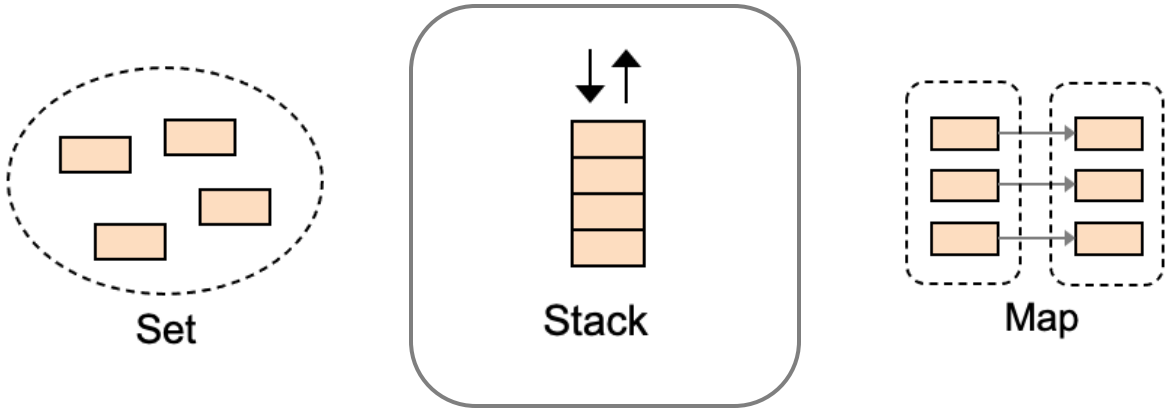
Data structures

Data structure: A way of organizing, processing, storing, and retrieving data. Different data structures are suited to different kinds of applications, and different kinds of efficiency.



- All widely used in computer science
- All can be implemented using *arrays* or *lists*

Lecture plan



Methodology

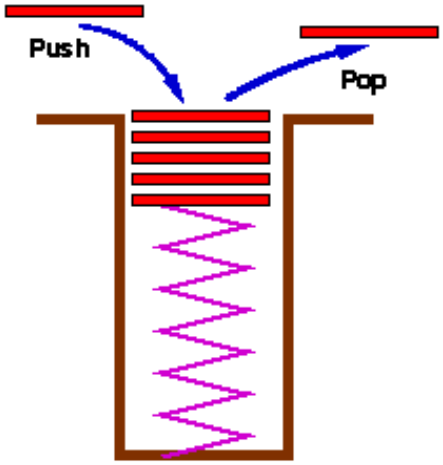
Abstraction: How to *use* the data structure

Implementation: How to *realize* the data structure (using arrays / lists)

Stack

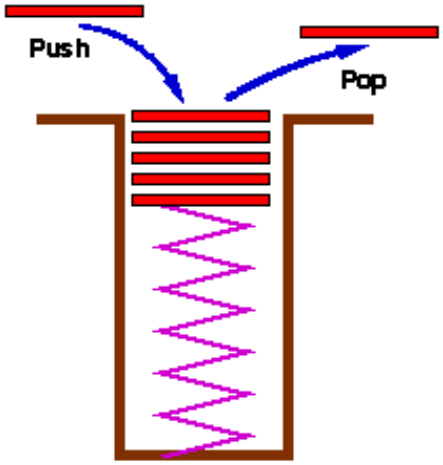
An ordered collection of elements, with a single entry / exit point

- Elements are pushed (added) onto the stack's top
- Elements are popped (removed) from the stack's top



Last In, First Out
(LIFO)

Stack



Last In, First Out
(LIFO)

An ordered collection of elements, with a single entry / exit point

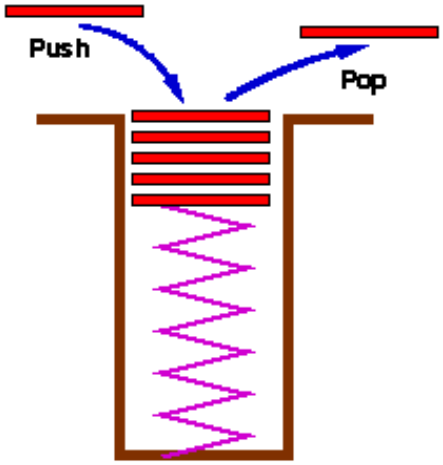
- Elements are pushed (added) onto the stack's top
- Elements are popped (removed) from the stack's top

```
// Some stack operation examples:
create an empty stack   [ ]
push 3                  [3]
push 7                  [3, 7]
push 6                  [3, 7, 6]
push 8                  [3, 7, 6, 8]
push 2                  [3, 7, 6, 8, 2]
x = pop // x = 2        [3, 7, 6, 8]
y = pop // y = 8        [3, 7, 6]
push 4                  [3, 7, 6, 4]
...
```

Stack

A fundamental data structure,
used in numerous applications
in the theory and practice of CS

Stack



Last In, First Out
(LIFO)

An ordered collection of elements, with a single entry / exit point

- Elements are pushed (added) onto the stack's top
- Elements are popped (removed) from the stack's top

```
// Some stack operation examples:
create an empty stack   [ ]
push 3                  [ 3 ]
push 7                  [ 3, 7 ]
push 6                  [ 3, 7, 6 ]
push 8                  [ 3, 7, 6, 8 ]
push 2                  [ 3, 7, 6, 8, 2 ]
x = pop // x = 2        [ 3, 7, 6, 8 ]
y = pop // y = 8        [ 3, 7, 6 ]
push 4                  [ 3, 7, 6, 4 ]
add     // addition     [ 3, 7, 10 ]
neg     // negation     [ 3, 7, -10 ]
...
```

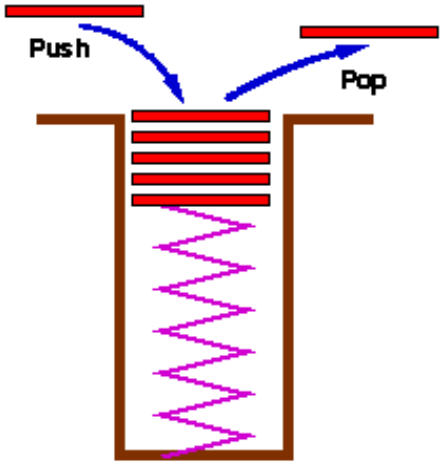
Arithmetic operations on a stack:

1. Pops as many operands as needed
2. Computes the operation
3. Pushes the resulting value

Stack

A fundamental data structure,
used in numerous applications
in the theory and practice of CS

Stack abstraction



API

```
/** Represents a stack of integers, with stack arithmetic. */
public class Stack {

    /** Constructs an empty  stack. */
    public Stack()

    /** Pushes the given element onto the stack's top. */
    public void push(int e)

    /** Removes and returns the stack's top element */
    public int pop()

    /** Checks if this stack is empty. */
    public boolean isEmpty()

    /** Addition: Pops the two top elements,
     *  adds them up, and pushes the result onto the stack */
    public void add()

    /** Negation: Pops the top element, negates it,
     *  and pushes the result onto the stack */
    public void neg()

    /** Similar arithmetic operations: subtract, multiply... */

    ...

    /** Returns a string representing this stack, in the form of [e1, e2, e3, ...],
     *  where the e's are the stack elements. */
    public String toString()
}
```

```
// Some class (client code)
Stack stack = new Stack();
stack.push(10); stack.push(20); stack.push(30);
stack.push(40); stack.push(50);
System.out.println("Stack:       " + stack);

int x = stack.pop();
System.out.println("After pop(): " + stack);

x = stack.pop();
System.out.println("After pop(): " + stack);

stack.add();
System.out.println("After add(): " + stack);
```

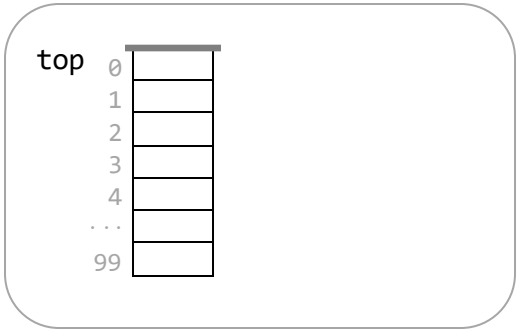
```
// Output
Stack:     [10, 20, 30, 40, 50]
After pop: [10, 20, 30, 40]
After pop: [10, 20, 30]
After add: [10, 50]
```

Stack implementation

```
/** Represents a stack of integers. */
public class Stack {

    // The default capacity of this stack
    private final static int MAX_SIZE = 100;

    // The elements of this stack
    private int[] elements = new int[MAX_SIZE];

    // The location in this stack just following the stack's top element
    private int top;

   /** Constructs a stack with the default size. */
    public Stack() {
        top = 0;
    }
```

```
// Some class (client code)
Stack stack = new Stack();
```

Behind the scene:



Push: Abstraction

```
/** Represents a stack of integers. */
public class Stack {

    // The default capacity of this stack
    private final static int MAX_SIZE = 100;

    // The elements of this stack
    private int[] elements = new int[MAX_SIZE];

    // The location in this stack just following the stack's top element.
    private int top;

   /** Constructs a stack with the default size. */
    public Stack() {
        top = 0;
    }

    /** Pushes the given element onto the stack. */
    public void push(int e) {
```

```
// Some class (client code)
Stack stack = new Stack();
stack.push(10); stack.push(20); stack.push(30);
System.out.println("Stack:       " + stack);
```

```
// Output
Stack:    [10, 20, 30]
```

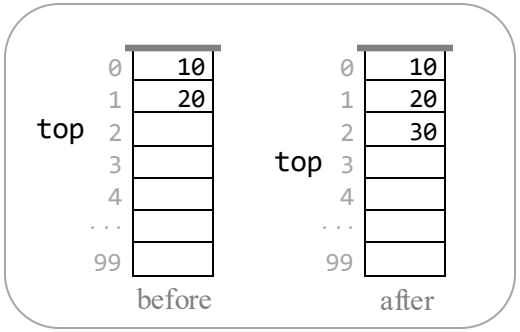
Push: Implementation

```
/** Represents a stack of integers. */
public class Stack {

    // The default capacity of this stack
    private final static int MAX_SIZE = 100;

    // The elements of this stack
    private int[] elements = new int[MAX_SIZE];

    // The location in this stack just following the stack's top element.
    private int top;

   /** Constructs a stack with the default size. */
    public Stack() {
        top = 0;
    }

    /** Pushes the given element onto the stack. */
    public void push(int e) {
        elements[top++] = e;
    }
```

```
// Some class (client code)
Stack stack = new Stack();
stack.push(10); stack.push(20); stack.push(30);
System.out.println("Stack:       " + stack);
```

```
// Output
Stack:    [10, 20, 30]
```

Implementation of `stack.push(30)`:



Pop: Abstraction

```
/** Represents a stack of integers. */
public class Stack {

    // The default capacity of this stack
    private final static int MAX_SIZE = 100;

    // The elements of this stack
    private int[] elements = new int[MAX_SIZE];

    // The location in this stack just following the stack's top element.
    private int top;

   /** Constructs a stack with the default size. */
    public Stack() {
        top = 0;
    }

    /** Pushes the given element onto the stack. */
    public void push(int e) {
        elements[top++] = e;
    }

    /** Removes and returns the stack's top element */
    public int pop() {
```

```
// Some class (client code)
Stack stack = new Stack();
stack.push(10); stack.push(20); stack.push(30);
System.out.println("Stack:       " + stack);

int x = stack.pop();
System.out.println("After pop(): " + stack);

x = stack.pop();
System.out.println("After pop(): " + stack);
```

```
// Output
Stack:     [10, 20, 30]
After pop: [10, 20]
After pop: [10]
```

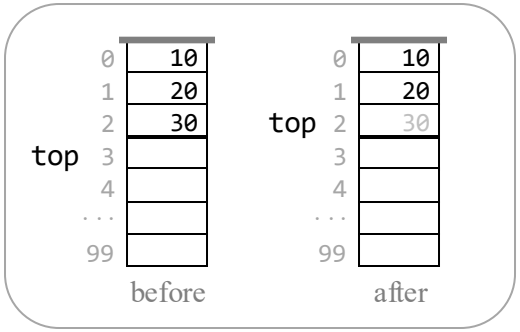
Pop: Implementation

```
/** Represents a stack of integers. */
public class Stack {

    // The default capacity of this stack
    private final static int MAX_SIZE = 100;

    // The elements of this stack
    private int[] elements = new int[MAX_SIZE];

    // The location in this stack just following the stack's top element.
    private int top;

   /** Constructs a stack with the default size. */
    public Stack() {
        top = 0;
    }

    /** Pushes the given element onto the stack. */
    public void push(int e) {
        elements[top++] = e;
    }

    /** Removes and returns the stack's top element */
    public int pop() {
        return elements[--top];
    }
```

```
// Some class (client code)
Stack stack = new Stack();
stack.push(10); stack.push(20); stack.push(30);
System.out.println("Stack:       " + stack);

int x = stack.pop();
System.out.println("After pop(): " + stack);

x = stack.pop();
System.out.println("After pop(): " + stack);
```

```
// Output
Stack:     [10, 20, 30]
After pop: [10, 20]
After pop: [10]
```

Implementation of `stack.pop()`:



Arithmetic operations: Abstraction

API

```
/** Represents a stack of integers, with stack arithmetic. */
public class Stack {

    /** Constructs an empty  stack. */
    public Stack()

    /** Pushes the given element onto the stack's top. */
    public void push(int e)

    /** Removes and returns the stack's top element */
    public int pop()

    ...

    /** Addition: Pops the two top elements,
     *  adds them up, and pushes the result onto the stack */
    public void add()

    /**Negation: Pops the top element, negates it,
     *  and pushes the result onto the stack */
    public void neg()

    ...
```

```
// Some class (client code)
Stack stack = new Stack();
stack.push(10); stack.push(20); stack.push(30);

stack.add();
System.out.println("After add: " + stack);

stack.neg();
System.out.println("After neg: " + stack);
```

```
// Output
Stack:     [10, 20, 30]
After add: [10, 50]
After neg: [10, -50]
```

Arithmetic operations: Implementation

```
/** Represents a stack of integers, with stack arithmetic. */
public class Stack {

    /** Constructs an empty  stack. */
    public Stack()

    /** Pushes the given element onto the stack's top. */
    public void push(int e)

    /** Removes and returns the stack's top element */
    public int pop()

    ...

    /** Addition: Pops the two top elements,
     *  adds them up, and pushes the result onto the stack */
    public void add() {
        push(pop() + pop());
    }

    /**Negation: Pops the top element, negates it,
     *  and pushes the result onto the stack */
    public void neg() {
        push(-1 * pop());
    }

    ...
```

```
// Some class (client code)
Stack stack = new Stack();
stack.push(10); stack.push(20); stack.push(30);

stack.add();
System.out.println("After add: " + stack);

stack.neg();
System.out.println("After neg: " + stack);
```

```
// Output
Stack:     [10, 20, 30]
After add: [10, 50]
After neg: [10, -50]
```

`toString()`: Abstraction

```
/** Represents a stack of integers. */
public class Stack {

    // The capacity of this stack
    private final static int MAX_SIZE = 100;

    // The elements of this stack
    private int[] elements = new int[MAX_SIZE];

    // The location in this stack just following the stack's top element.
    private int top;

    ...

    /** Returns a string representing this stack, in the form of [e1, e2, e3, ...],
     *  where the e's are the stack elements. */
    public String toString() {
```

```
// Some class (client code)
Stack stack = new Stack();
stack.push(10); stack.push(20); stack.push(30);
stack.push(40); stack.push(50);
System.out.println("Stack: " + stack);
```

```
// Output
Stack: [10, 20, 30, 40, 50]
```

`toString()`: Implementation

```
/** Represents a stack of integers. */
public class Stack {

    // The capacity of this stack
    private final static int MAX_SIZE = 100;

    // The elements of this stack
    private int[] elements = new int[MAX_SIZE];

    // The location in this stack just following the stack's top element.
    private int top;

    ...

    /** Returns a string representing this stack, in the form of [e1, e2, e3, ...],
     *  where the e's are the stack elements. */
    public String toString() {
```

```
// Some class (client code)
Stack stack = new Stack();
stack.push(10); stack.push(20); stack.push(30);
stack.push(40); stack.push(50);
System.out.println("Stack: " + stack);
```

```
// Output
Stack: [10, 20, 30, 40, 50]
```

Basic idea:

```
public String toString() {
   String s = "["
   for (int i = 0; i < top; i++) {
      s = s + elements[i] + " "
   }
   return s + "] ";
}
```

Wasteful!

Each iteration creates a new String object (`s`).

`toString()`: Implementation

```
/** Represents a stack of integers. */
public class Stack {

    // The capacity of this stack
    private final static int MAX_SIZE = 100;

    // The elements of this stack
    private int[] elements = new int[MAX_SIZE];

    // The location in this stack just following the stack's top element.
    private int top;

    ...

    /** Returns a string representing this stack, in the form of [e1, e2, e3, ...],
     *  where the e's are the stack elements. */
    public String toString() {
        StringBuilder s = new StringBuilder("[");
        for (int i = 0; i < (top-1); i++) {
            s.append(elements[i] + ", ");
        }
        return s.append(elements[top-1] + "]").toString();
    }

    ...
```

```
// Some class (client code)
Stack stack = new Stack();
stack.push(10); stack.push(20); stack.push(30);
stack.push(40); stack.push(50);
System.out.println("Stack: " + stack);
```

```
// Output
Stack: [10, 20, 30, 40, 50]
```

Solution: `StringBuilder`

A Java class whose objects are
mutable sequences of characters

Makes string building iterations more efficient (memory-wise):

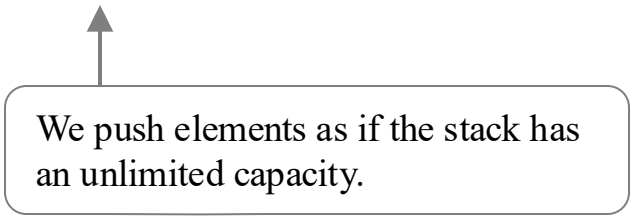
Unlike `String` objects, which are immutable, `StringBuilder` objects can be changed using methods like `append`, `insert`, or `delete`.

Best practice

When building a sequence of characters which is potentially large, use a `StringBuilder` object.

Unlimited capacity: Abstraction

```
/** Represents a stack of integers. */
public class Stack {

    // The capacity of this stack
    private final static int MAX_SIZE = 100;

    // The elements of this stack
    private int[] elements = new int[MAX_SIZE];

    // The location in this stack just following the stack's top element.
    private int top;

    ...
    /** Pushes the given element onto the stack's top. */
    public void push(int e) {
```




```
// Some class (client code)
Stack stack = new Stack();
// Can potentially lead to stack overflow:
stack.push(10); stack.push(20); stack.push(30);
stack.push(40); stack.push(50); ...
...
```

Unlimited capacity: Implementation

```
/** Represents a stack of integers. */
public class Stack {

    // The capacity of this stack
    private final static int MAX_SIZE = 100;

    // The elements of this stack
    private int[] elements = new int[MAX_SIZE];

    // The location in this stack just following the stack's top element.
    private int top;

    ...

    /** Pushes the given element onto the stack's top. */
    public void push(int e) {
        if (top == elements.length - 1) {
            resize();
        }
        elements[top++] = e;
    }

    // Doubles the stack's capacity. */
    private void resize() {
        int[] newElements = new int[elements.length * 2];
        // Creates a new array. Copies elements.size elements from
        //  location 0 in elements to the new array, starting at location 0.
        //  Sets the elements array of this stack to the new array.
        System.arraycopy(elements, 0, newElements, 0,
                         elements.length);
        elements = newElements;
    }

    ...
```

```
// Some class (client code)
Stack stack = new Stack();
// Can potentially lead to stack overflow:
stack.push(10); stack.push(20); stack.push(30);
stack.push(40); stack.push(50);
...
```

Handling unlimited capacity

When the stack reaches capacity, we double its capacity

Best practice

Similar techniques can be used to feature an unlimited capacity for any data structure that uses an array implementation.

Stack: Efficiency

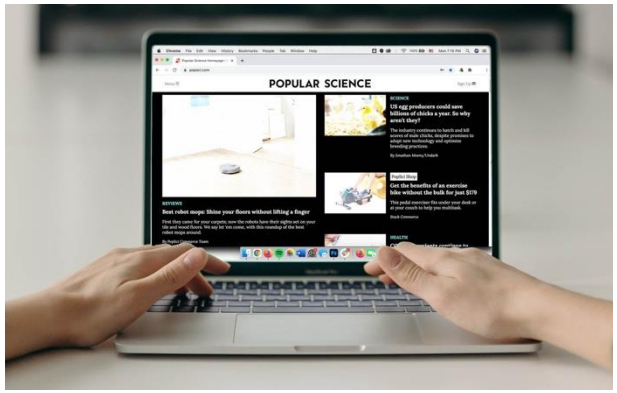
In a stack of N elements:

Push: $O(1)$

Pop: $O(1)$

Add, subtract, neg: $O(1)$

Stack application examples



Browsing

The URL's that the user visits are pushed onto a stack;

When clicking "back", a pop operation returns the page that was visited last before the current one

Editing

The user's editing operations are pushed onto a stack;

When clicking "undo", the most recent operation is popped and undone

Compilation

Simplifying compilers.

Compilation (Java, Python, C#)

Prog.java

```
7 * (4 + 2) / (5 – 3)
```

Prog.bin

```
1100101001011011
0001010100011001
1000101001000010
1100100001000011
0000101001011000
1000101011011010
0000101001011000
0100101001011010
1100101001011011
1100100001000011
0000101001011000
...
```

Executable code

The challenge

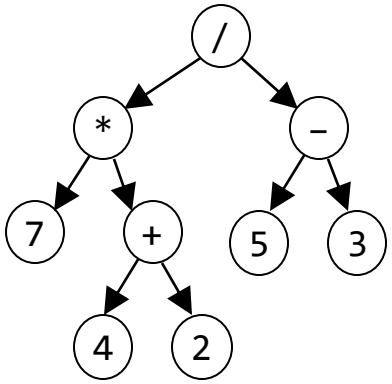
Breaking the compilation task into independent and modular steps

Compilation (Java, Python, C#)

Prog.java

```
7 * (4 + 2) / (5 - 3)
```

parser



Parse tree

Source code

javac Prog.java

code generator

Prog.class

```
push 7
push 4
push 2
add
mult
push 5
push 3
sub
div
```

Bytecode

(operates on a stack)

java Prog

translator

Prog.bin

```
1100101001011011
0001010100011001
1000101001000010
1100100001000011
0000101001011000
1000101011011010
0000101001011000
0100101001011010
1100101001011011
1100100001000011
0000101001011000
...
```

Executable code

`javac Prog.java`

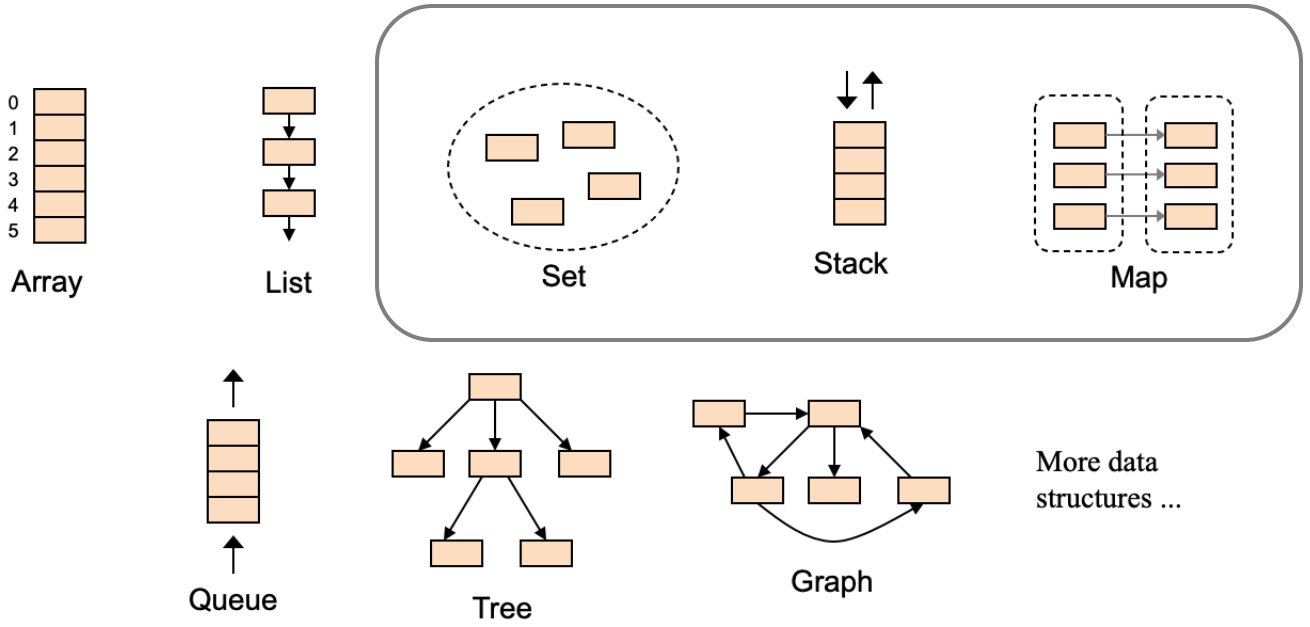
(translates Java code into Bytecode)

`java Prog`

(translates Bytecode into binary code, and executes it)

Data structures

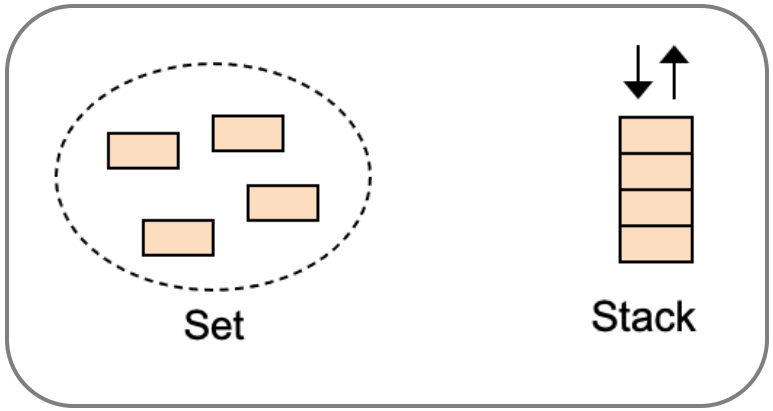
Data structure: A way of organizing, processing, storing, and retrieving data. Different data structures are suited to different kinds of applications, and different kinds of efficiency.

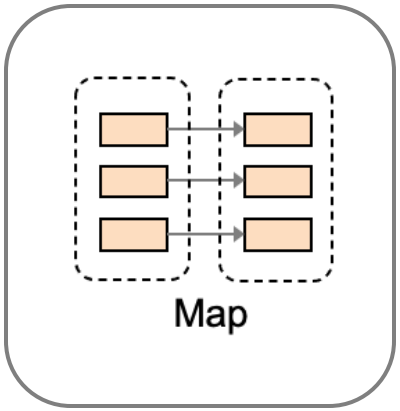


- All widely used in computer science
- All can be implemented using *arrays* or *lists*

Lecture plan

Last week

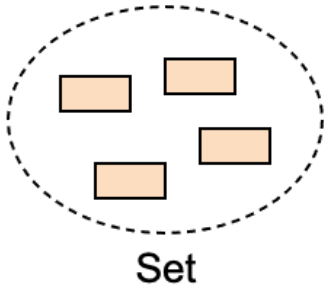


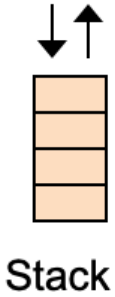


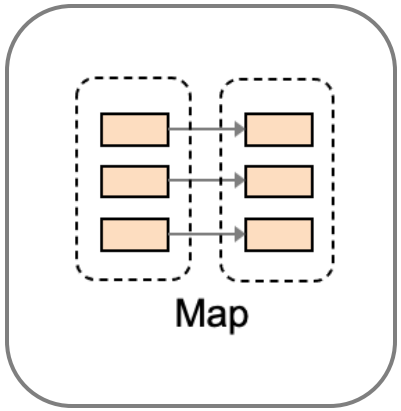
Today

Lecture plan

Last week







Today

Methodology

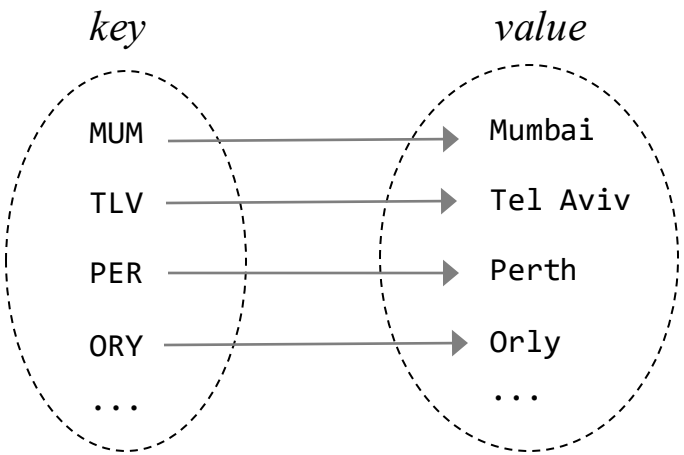
Theory: The mapping algorithm

Applications: Fast storage / retrieval.

Map

Map: a set of <*key, value*> pairs

(<"MUM", "Mumbai">, <"TLV", "Tel Aviv">, ...)



40,000+ airport codes

IATA CODES

Location	Airport	Location	Airport	Location	Airport
Aalborg	AAL	Glasgow	GLA	Okinawa	OKA
Aarhus	AAR	Glasgow, Prestwick	PIK	Oklahoma City	OKC
Abadan	ABD	Goma	GOM	Omaha	OMA
Abakan	ABA	Goodnew Bay	GNU	Ontario	ONT
Aberdeen	ABR	Goose Bay	YYR	Oran	ORN
Aberdeen	ABZ	Goroka	GKA	Orebro	ORB
Abha	AHB	Gothenburg	GOT	Orlando	MCO
Abidjan	ABJ	Gran Canaria	LPA	Osaka	OSA
Abilene	ABI	Grand Cayman	GCM	Osaka, Itami	ITM
Abu Dhabi	AUH	Grand Fork	GFK	International	KIX
Abuja	ABV	Grand Rapids	GRR	Oshkosh	OSH
Acapulco	ACA	Grand Turk	GDT	Osijek	OSI
Acarigua	AGV	Grande Prairie	YQU	Oslo	OSL
Accra	ACC	Grayling	KGX	Oslo, Fornebu Airport	FBU
Adak Island	ADK	Graz	GRZ	Oslo, Gardermoen	GEN
Adana	ADA	Great Falls	GTF	Osorno	ZOS
Addis Ababa	ADD	Green Bay	GRB	Ostend	OST
Adelaide	ADL	Greensboro	GSO	Ottawa	YOW
Aden	ADE	Greenville	GLH	Ouagadougou	OUA
Adler	AER	Greenville	GSP	Oxnard/Ventura	OXR
Agadir	AGA	Greenville	PGV	Pago Pago	PPG
Aguascaliente	AGU	Grenada	GND	Palembang	PLM
Ahmedabad	AMD	Grenoble	GNB	Palermo	PMO
Ajaccio	AJA	Groningen	GRQ	Palm Springs	PSP
Akiachak	KKI	Guadalajara	GDL	Palma de Mallorca	PMI
Akiak	AKI	Guam	GUM	Panama City (FL.)	PFN
Akita	AXT	Guangzhou	CAN	Panama City	PTY
Akron/Canton	CAK	Guatemala City	GUA	Panama City, Paitilla	PAC
Al-Baha	ABT	Guayaquil	GYE	Papeete	PPT
Albany	ABY	Guernsey	GCI	Paphos	PFO
Albany	ALB	Guilin	KWL	Paramaribo	PBM
Albert Bay	YAL	Guiyang	KWE	Hoop	ORG
Albuquerque	ABQ	Gulfport	GPT	Paris	PAR
Aleandroupolis	AXD	Gunnison	GUC	Paris, Charles de Gaulle	CDG
Aleppo	ALP	Hagen	HAG	Paris, Le Bourget	LBG
Alexandria	ESF	Haikou	HAK	Paris, Orly	ORY
Al-Fujairah	FJR	Hail	HAS	Pasco	PSC
Algiers	ALG	Hakodate	HKD	Pau	PUF
Alicante	ALC	Halifax	YHZ	Penang	PEN
Allakaket	AET	Hall Beach	YUX	Pensacola	PNS
Allentown	ABE	Hamburg	HAM	Peoria	PIA
Alma Ata	ALA	Hamilton	YHM	Pereira	PEI
Alor Setar	AOR	Williamsburg	PHF	Perpignan	PGF
Altoona	AOO	Hangzhou	HGH	Perryville	KPV
Amarillo	AMA	Hanoi	HAN	Perth	PER
Amchitka	AHT	Hanover	HAJ	Pescara	PSR
Amman	AMM	Harare	HRE	Peshawar	PEW

Map algorithm

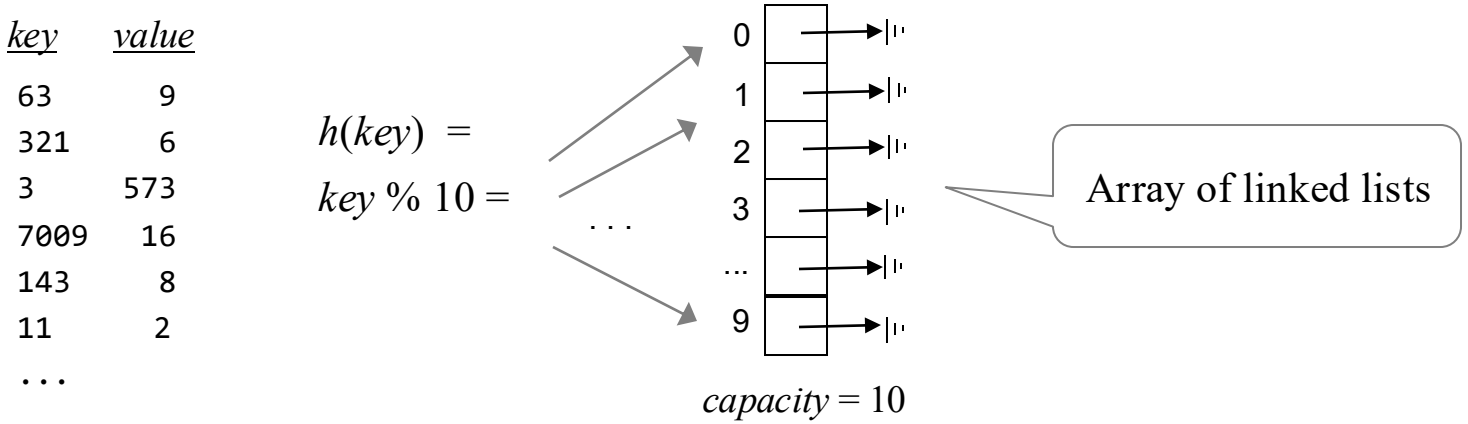
key	*value*
63	9
321	6
3	573
7009	16
143	8
11	2
...	

Without loss of generality

We'll illustrate a map that manages <*int*, *int*> mappings;

Later we'll generalize to mapping *any data type* on *any data type*.

Map algorithm



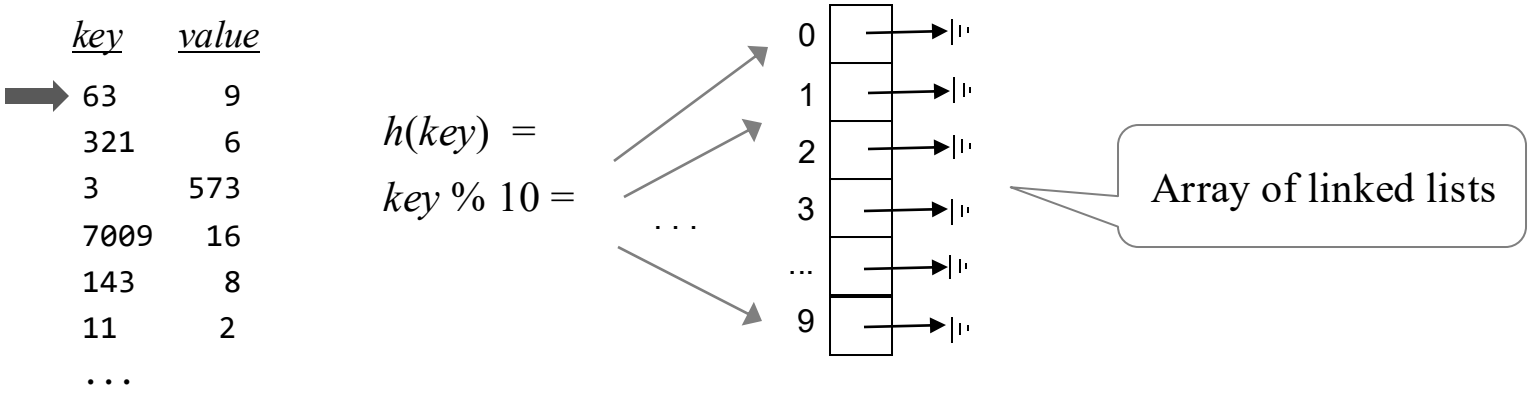
$h(x)$

"Hash" function: Designed to distribute x evenly among the various array elements

("hashing" is a synonym of "scrambling")

$x\%c$ where c is some constant is a commonly-used hashing function.

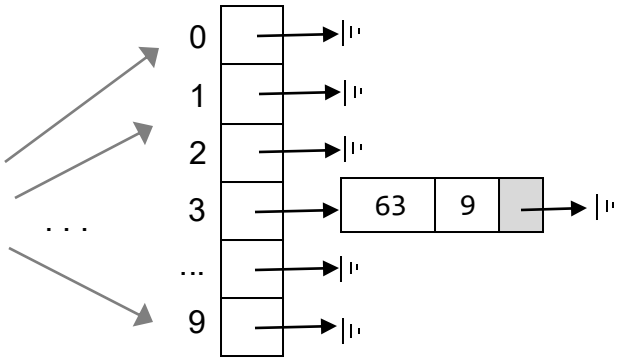
Map algorithm



Map algorithm

key	*value*
63	9
321	6
3	573
7009	16
143	8
11	2
...	

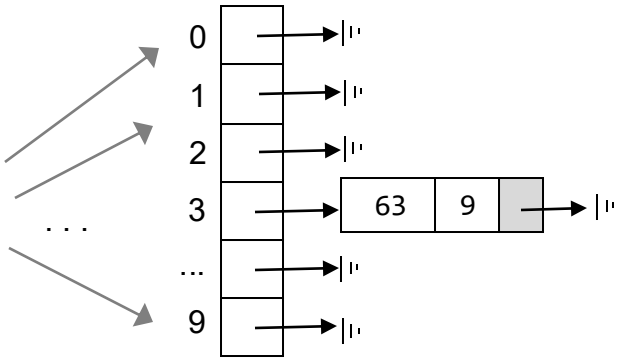
$h(key) = key \% 10 =$



Map algorithm

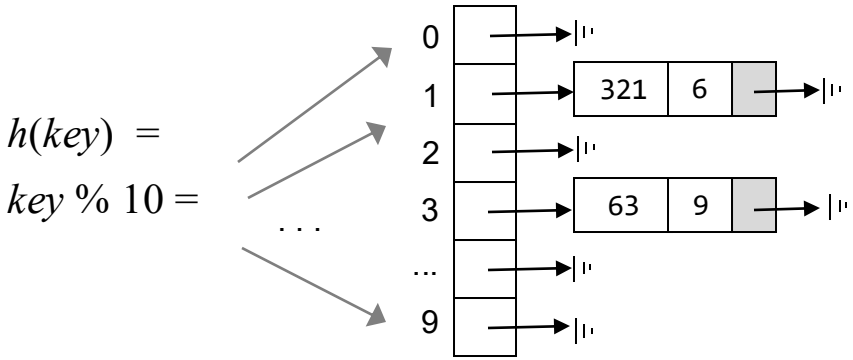
key	*value*
63	9
321	6
3	573
7009	16
143	8
11	2
...	

$h(key) = key \% 10 =$



Map algorithm

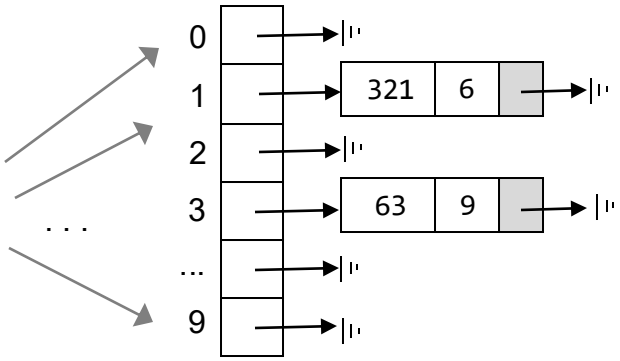
key	*value*
63	9
321	6
3	573
7009	16
143	8
11	2
...	



Map algorithm

key	*value*
63	9
321	6
3	573
7009	16
143	8
11	2
...	

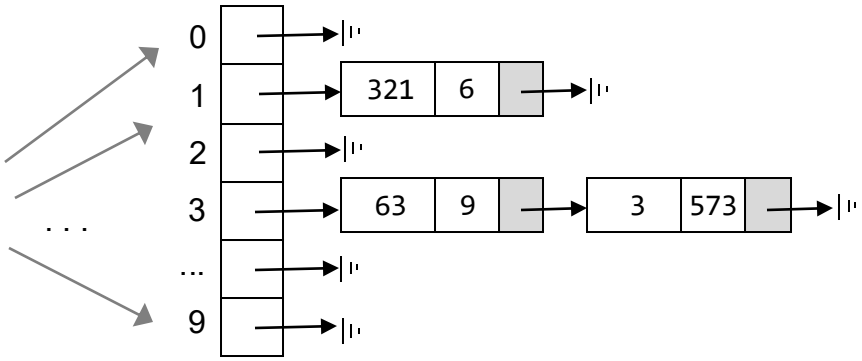
$h(key) = key \% 10 =$



Map algorithm

key	*value*
63	9
321	6
3	573
7009	16
143	8
11	2
...	

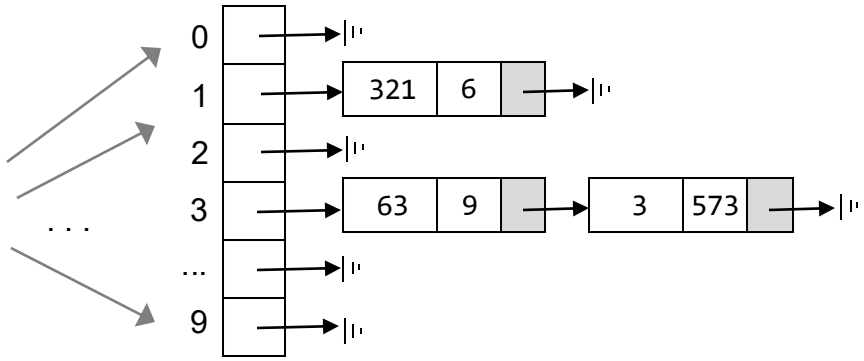
$h(key) = key \% 10 =$



Map algorithm

key	*value*
63	9
321	6
3	573
7009	16
143	8
11	2
...	

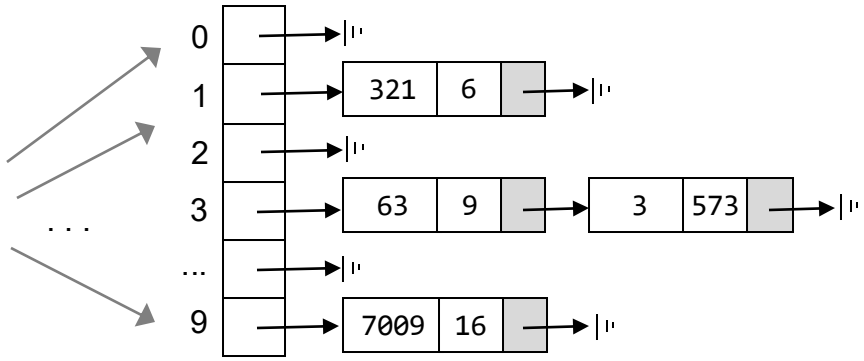
$h(key) = key \% 10 =$



Map algorithm

key	*value*
63	9
321	6
3	573
7009	16
143	8
11	2
...	

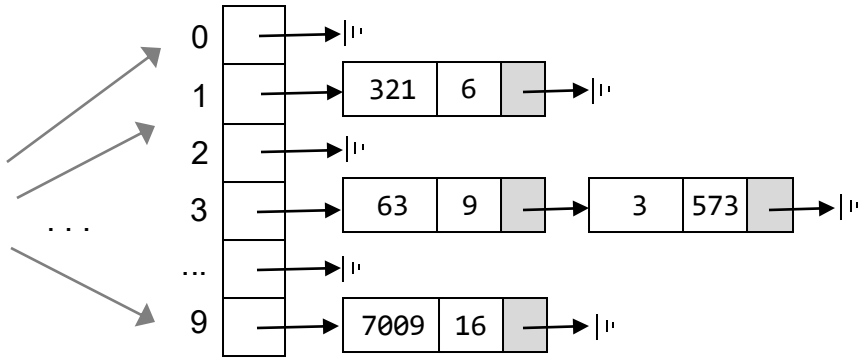
$h(key) = key \% 10 =$



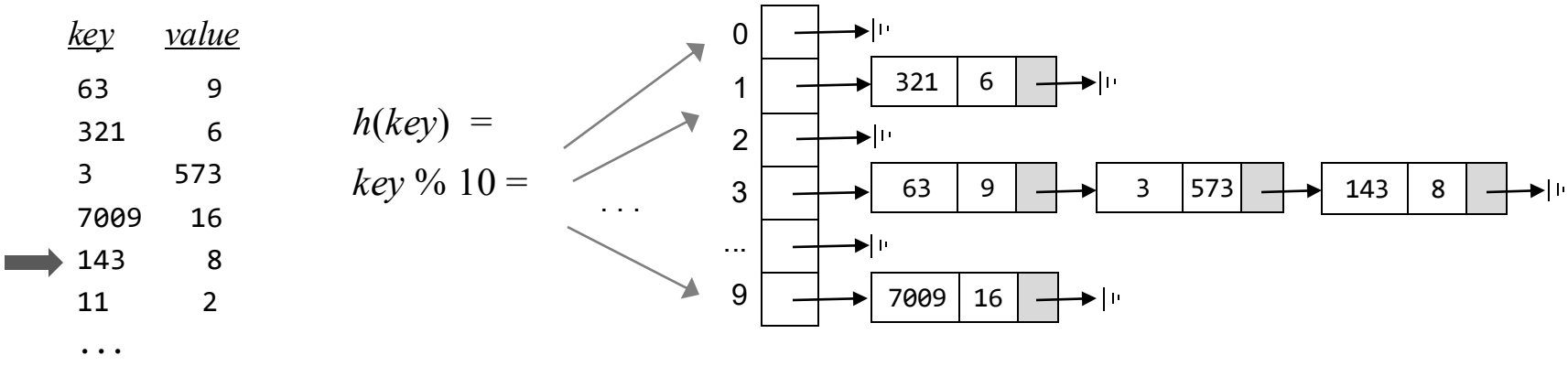
Map algorithm

key	*value*
63	9
321	6
3	573
7009	16
143	8
11	2
...	

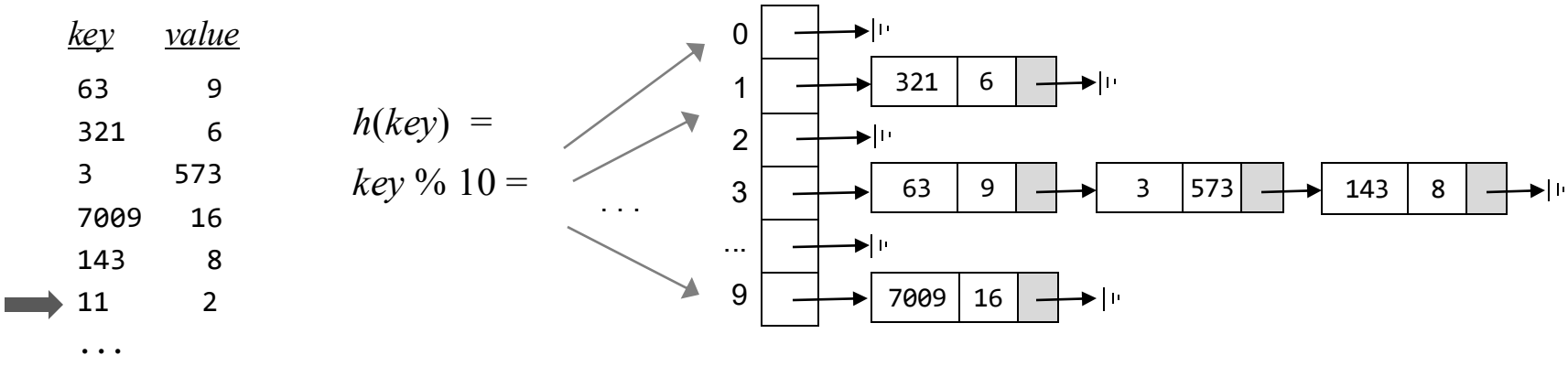
$h(key) = key \% 10 =$



Map algorithm



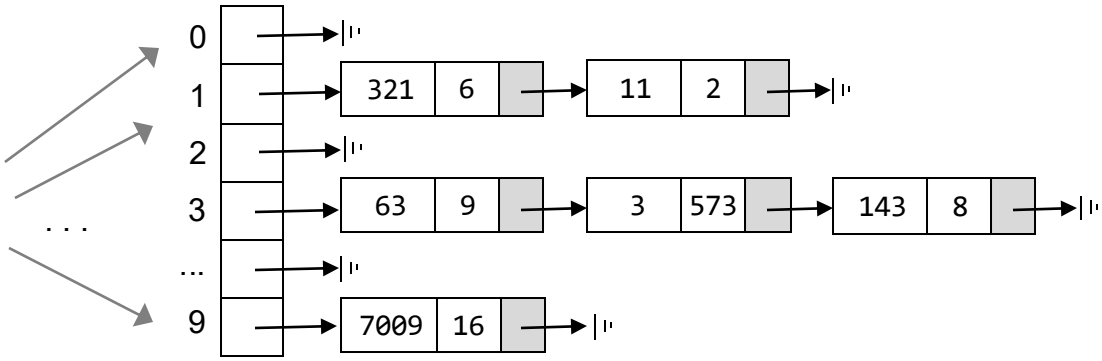
Map algorithm



Map algorithm

key	*value*
63	9
321	6
3	573
7009	16
143	8
11	2
...	

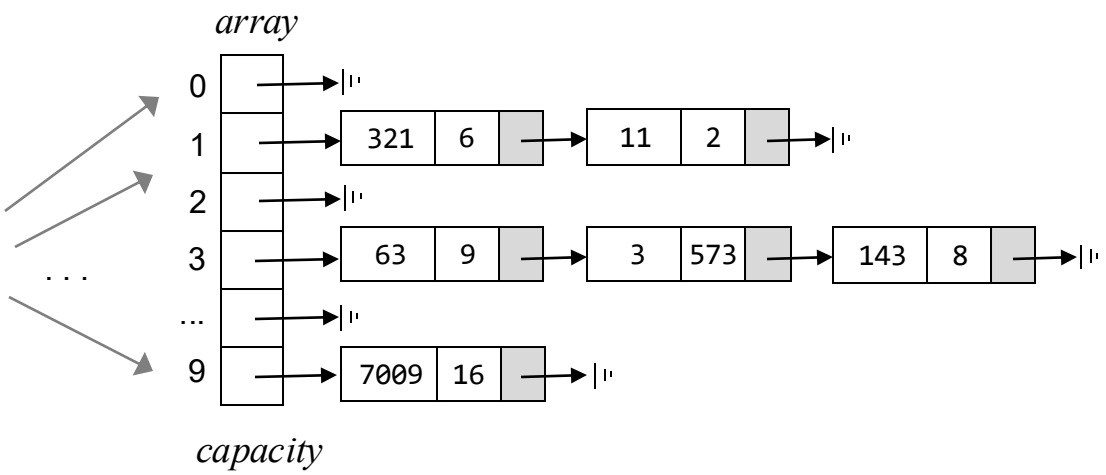
$h(key) = key \% 10 =$



Map algorithm

key	*value*
63	9
321	6
3	573
7009	16
143	8
11	2
...	

$h(key) = key \% 10 =$



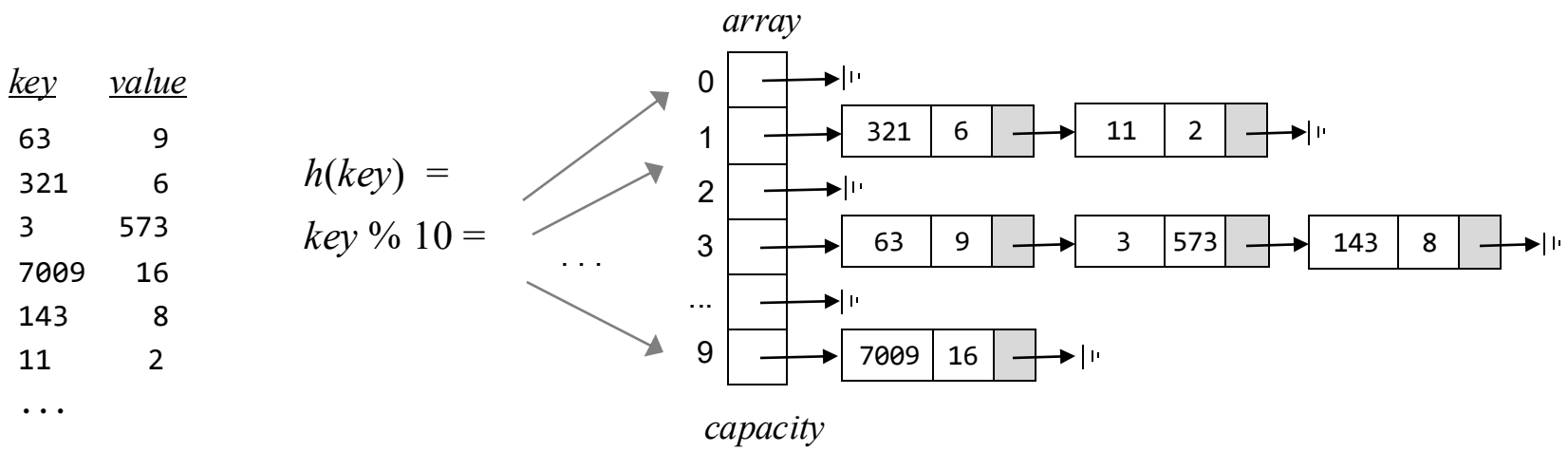
Capacity:

A design parameter;

We can choose 10, or any other value;

The larger is capacity, the smaller are the lists.

Map algorithm



put(*key* , *value*)

- Compute *i* = *h*(*key*)
- Add <*key* , *value*> to the *array*[*i*] list

***O*(1)**

value = **get**(*key*)

- Compute *i* = *h*(*key*)
- Search the *array*[*i*] list

***O*(*n*)**, where *n* is the longest list

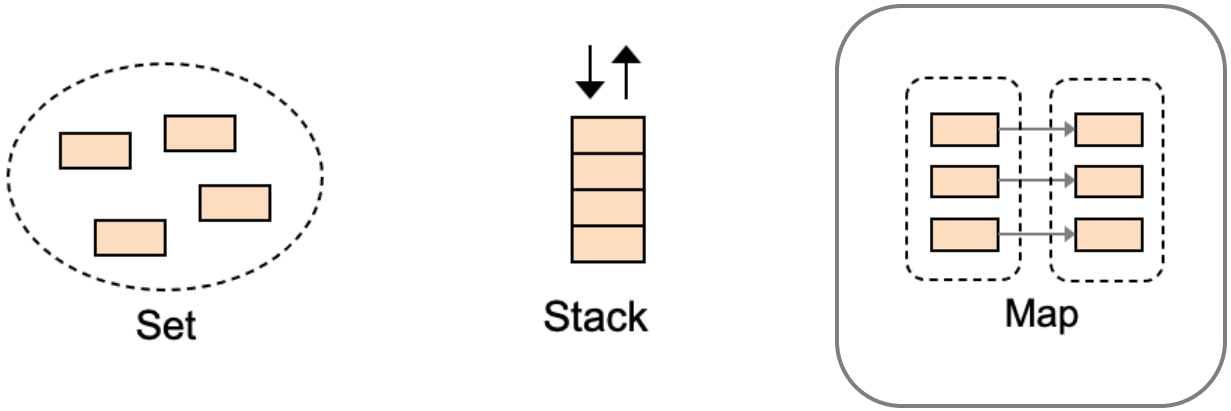
Algorithm properties

The larger is *capacity*, the smaller is *n*

Accessing array elements is $O(1)$

Bottom line: Excellent algorithm for storing / retrieving <*key* , *value*> pairs.

Lecture plan



Methodology

Theory: The mapping algorithm

Applications: Fast storage / retrieval.

next lecture