CS3500: Object-Oriented Design
Spring 2014

Class 8
1.31.2014

Northeastern University
College *of* Computer and Information Science

Today...

- Review Iterator & Data Abstraction
- Queue
- Abstraction function
- Representation invariant
- Binary Search
- Total Order

Iterators

[Lewis & Chase]

- An *iterator* is an object that provides the means to iterate over a collection.
- Provide methods that allow the user to acquire and use each element in a collection in turn.

```
//An iterator over a collection.
public interface Iterator<E> {
  // Returns true if the iteration has more
  // elements.
  public boolean hasNext ();

  // Returns the next element in the iteration.
  public E next ();

  // Removes from the underlying collection the last
  // element returned by the iterator (optional).
  public void remove ();
}
```

Assignment 4

- Implement MyMap
- Due: Friday, January 31, 2014 at 11:59 pm

```
String alice = "Alice";
String bob = "Bob";
String carol = "Carol";
String dave = "Dave";
Integer one = 1;
Integer two = 2;

MyMap<Integer, String> f0 = MyMap.empty();
MyMap<Integer, String> f7
                  = f0.include(one, dave);
f7 = f7.include(two, dave);
f7 = f7.include(two, bob);
f7 = f7.include(one, alice);

f7.size() is ???

f7.get(one) is ???
f7.get(two) is ???
```

Testing Iterator

```
f0 = MyMap.empty();
f1 = f0.include(one, alice);
f2 = f1.include(two, bob);
f5 = f2.include(one, carol);

f = f5;
m = f0;
it = f.iterator();
count = 0;
while (it.hasNext()) {
    Integer k = it.next();
    m = m.include(k, f.get(k));
    count = count + 1;
}
assertTrue("iterator [(1 Carol)(2 Bob)(1 Alice)]",
        f5.equals(m));
assertFalse("iteratorSanity [(1 Carol)(2 Bob)(1 Alice)]",
        it.hasNext());
assertTrue("Iterator count [(1 Carol)(2 Bob)(1 Alice)]",
        f.size() == count);
```

Testing Iterator

```
f0 = MyMap.empty();
f1 = f0.include(one, alice);
f2 = f1.include(two, bob);
f5 = f2.include(one, carol);

f = f5;
m = f0;
count = 0;
for (Integer k : f) {
    m = m.include(k, f.get(k));
    count = count + 1;
}
assertTrue("iterator [(1 Carol)(2 Bob)(1 Alice)]",
        f5.equals(m));
assertTrue("Iterator count [(1 Carol)(2 Bob)(1 Alice)]",
        f.size() == count);
```

```
MyMap<String,Integer> f0 = MyMap.empty();
MyMap<String,Integer> f1
    = f0.include ("Aaron", 1);
MyMap<String,Integer> f2
    = f1.include ("Barb",  2);
MyMap<String,Integer> f3
    = f2.include ("Carl",  3);
MyMap<String,Integer> f4
    = f3.include ("Barb",  4);
MyMap<String,Integer> f5
    = f4.include ("Aaron", 5);
```

```java
/**
 * A comparator for Integer values.
 */
private static class UsualIntegerComparator implements Comparator<Integer> {
    /**
     * Compares its two arguments for order.
     * @param m first Integer to compare
     * @param n second Integer to compare
     * @return Returns a negative integer, zero, or a positive integer as m is
     *         less than, equal to, or greater than n
     */
    public int compare(Integer m, Integer n) {
        return m.compareTo(n);
    }
    /**
     * Is this <code>Comparator</code> same as the given object
     * @param o the given object
     * @return true if the given object is an instance of this class
     */
    public boolean equals(Object o) {
        return (o instanceof UsualIntegerComparator);
    }
    /**
     * There should be only one instance of this class = all are equal
     * @return the hash code same for all instances
     */
    public int hashCode() {
        return (this.toString().hashCode());
    }
    /**
     * @return name of class
     */
    public String toString() {
        return "UsualIntegerComparator";
    }
}
```

Data Abstraction

- What is data abstraction?
    - A type of abstraction that allows us to introduce new types of data objects.
- What must we define with a new data type?
    - set of objects
    - set of operations characterizing the behavior of the objects
- What do we gain from data abstraction?
    - Separation between how the objects behave and how the objects are implemented
- Representation

Queue

- Similar to list
- First In, First Out (FIFO)

- Enqueue
- Dequeue

Immutable Queue Algebraic Specs

```
empty   :                -> Queue
enqueue: Queue x int     -> Queue
dequeue: Queue           -> Queue
first   : Queue          -> int
isEmpty: Queue           -> boolean
```

Immutable Queue Algebraic Specs

```
empty  :               -> Queue
enqueue: Queue x int -> Queue
dequeue: Queue         -> Queue
first  : Queue         -> int
size   : Queue         -> int
isEmpty: Queue         -> boolean
```

```
Queue.dequeue(Queue.enqueue(q,i))  = q  if(Queue.isEmpty(q))
Queue.dequeue(Queue.enqueue(q,i))
   = Queue.enqueue(Queue.dequeue(q), i) if(!Queue.isEmpty(q))
Queue.first(Queue.enqueue(q,i)) = i              if(Queue.isEmpty(q))
Queue.first(Queue.enqueue(q,i)) = Queue.first(q) if(!Queue.isEmpty(q))
Queue.size(Queue.empty()) = 0
Queue.size(Queue.enqueue(q,i)) = 1 + Queue.size(q)
Queue.isEmpty(Queue.empty()) = true
Queue.isEmpty(Queue.enqueue(q,i) = false
Queue.empty().toString() = ""
Queue.enqueue(q,i).toString() = q.toString() +"(" + i + ")"
```

```
equals if elements in same order
```

Immutable Queue Algebraic Specs with Dynamic Methods

```
public static method:
empty   :      -> Queue

public dynamic methods:
enqueue: int -> Queue
dequeue:     -> Queue
first  :     -> int
size   :     -> int
isEmpty:     -> boolean

q.enqueue(i).dequeue()
   = q                                  if(q.isEmpty())
q.enqueue(i).dequeue()
   = q.dequeue().enqueue(i)             if(!q.isEmpty())

q.enqueue(i).first()
   = i              if(q.isEmpty())
q.enqueue(i).first()
   = q.first()      if(!q.isEmpty())

q.size(q.empty()) = 0
q.size(q.enqueue(i)) = 1 + q.size()

Queue.empty().isEmpty() = true
q.enqueue(i).isEmpty() = false

Queue.empty().toString() = ""
q.enqueue(i).toString() = q.toString() +"(" + i + ")"
```

Mutable Queue Specs

```
public static method:
empty   :                -> Queue

public dynamic methods:
enqueue: int ->          adds int to end of
                         queue
dequeue:     ->          removes first int from
                         queue
first   :    -> int      returns first int in
                         queue
size    :    -> int      returns size of queue
                         (number of ints)
isEmpty:     -> boolean  returns whether Queue
                         is empty
```

Mutable Queue Specs

```
/**
 * enqueue the given int at the end of this queue
 * Requires: any valid queue and a valid int
 * Modifies: the current queue is one element bigger,
 * the new element is added at the logical end of the queue
 */
public void enqueue(int k)

/**
 * remove the int at the front of this queue
 * Requires: any valid queue
 * Modifies: the current queue is one element shorter,
 * the int at the logical front of the queue is removed
 */
public void dequeue()

/**
 * produce the int at the front of this queue
 * Requires: any valid queue
 * Effect: returns the int at the logical front of the queue
 */
public int first()


/**
 * produce the size this queue
 * Requires: any valid queue
 * Effect: returns the size of the queue
 */
public int size()
```

Mutable Queue with ArrayList

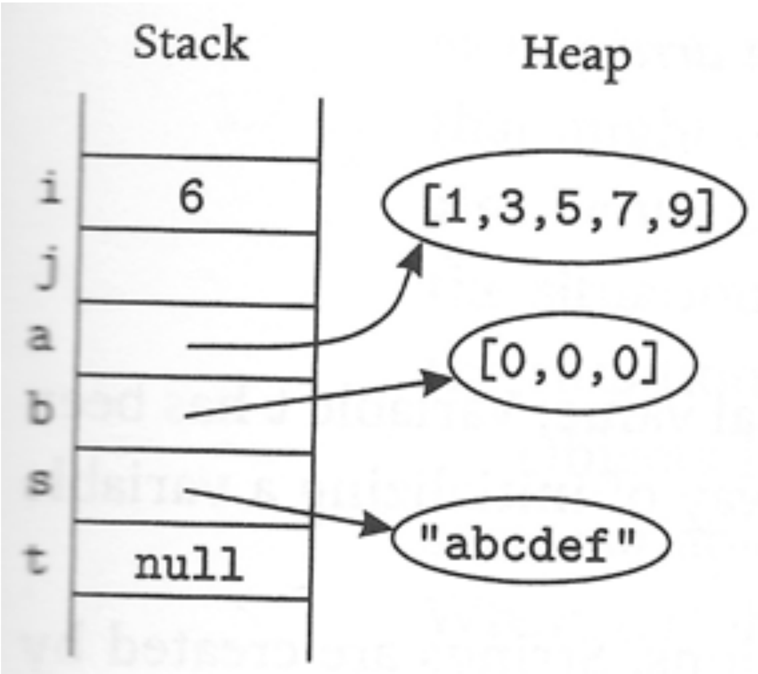
- first element of the queue at index 0
- most recently added element at index 0

```java
public class Queue {
  ArrayList<Integer> q;
  private Queue() {
    this.q = new ArrayList<Integer>();
  }
  public static Queue empty(){
    return new Queue();
  }
  public void enqueue(int i){
    this.q.add(new Integer(i));
  }
  public int first(){
    if (this.q.isEmpty()) {
      throw new RuntimeException("No first in an empty queue");
    }
    else {
      return this.q.get(0).intValue();
    }
  }
  public void dequeue(){
    if (this.q.isEmpty()) {
      throw new RuntimeException("Nothing to remove from an empty queue");
    }
    else {
      this.q.remove(0);
    }
  }
  public boolean isEmpty() {
    return this.q.isEmpty();
  }
  public Integer size() {
    return this.q.size();
  }
}
```

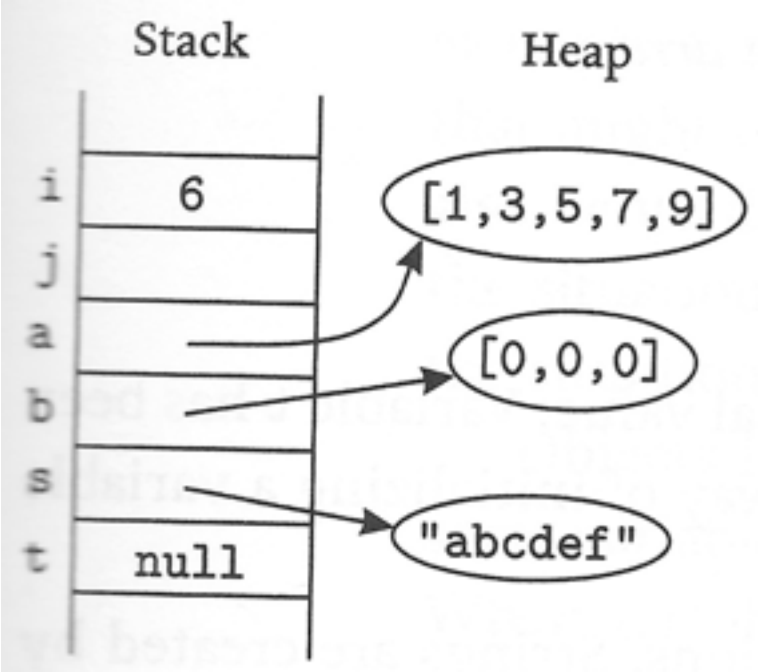
```
public class Queue {
  ArrayList<Integer> q;
  private Queue() {
    this.q = new ArrayList<Integer>();
  }
  public static Queue empty(){
    return new Queue();
  }
  public void enqueue(int i){
    this.q.add(0, new Integer(i));
  }
  public int first(){
    if (this.q.isEmpty()) {
      throw new RuntimeException("No first in an empty queue");
    }
    else {
      return this.q.get(q.size() - 1).intValue();
    }
  }
  public void dequeue(){
    if (this.q.isEmpty()) {
      throw new RuntimeException("Nothing to remove from an empty queue");
    }
    else {
      this.q.remove(q.size() - 1);
    }
  }
  public boolean isEmpty() {
    return this.q.isEmpty();
  }
  public int size() {
    return this.q.size().intValue();
  }
}
```

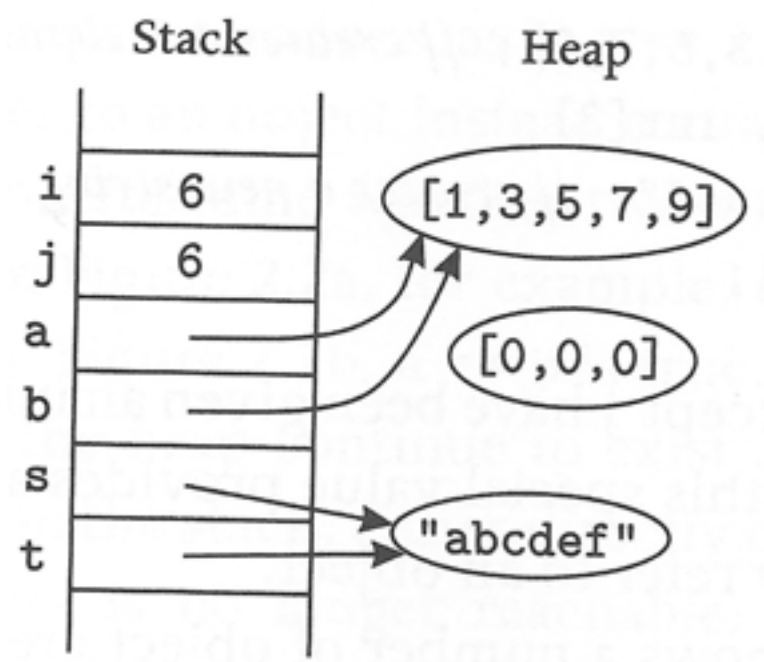
```
public class Queue {
  ArrayList<Integer> q;
  private Queue() {
    this.q = new ArrayList<Integer>();
  }
  public static Queue empty(){
    return new Queue();
  }
  public void enqueue(int i){
    this.q.add(0, new Integer(i));
  }
  public int first(){
    if (this.q.isEmpty()) {
      throw new RuntimeException("No first in an empty queue");
    }
    else {
      return this.q.get(q.size() - 1).intValue();
    }
  }
  public void dequeue(){
    if (this.q.isEmpty()) {
      throw new RuntimeException("Nothing to remove from an empty queue");
    }
    else {
      this.q.remove(q.size() - 1);
    }
  }
  public boolean isEmpty() {
    return this.q.isEmpty();
  }
  public int size() {
    return this.q.size().intValue();
  }
}
```

```
int i = 6;
int j; //uninitialized
int [] a = {1, 3, 5, 7, 9}; //
creates a 5-element array
int [] b = new int[3];
String s = "abcdef";
String t = null;
```

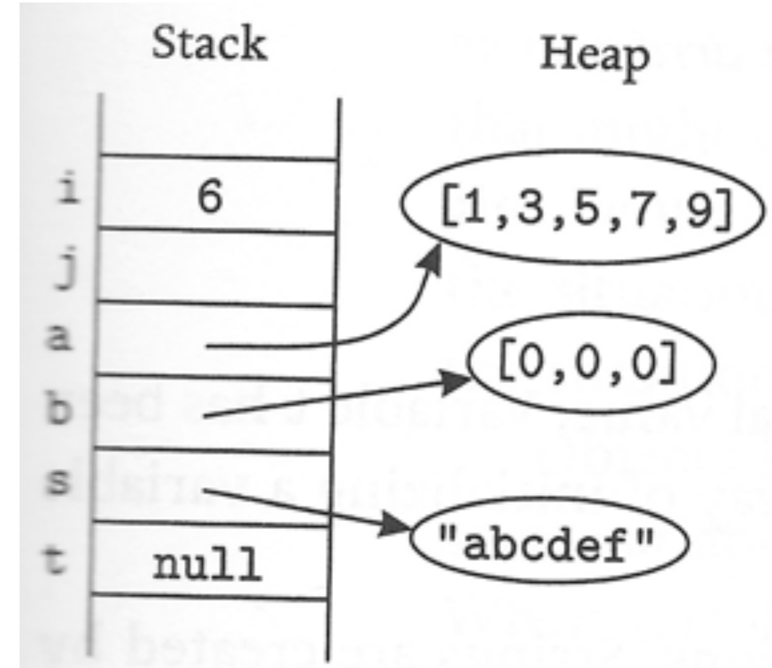


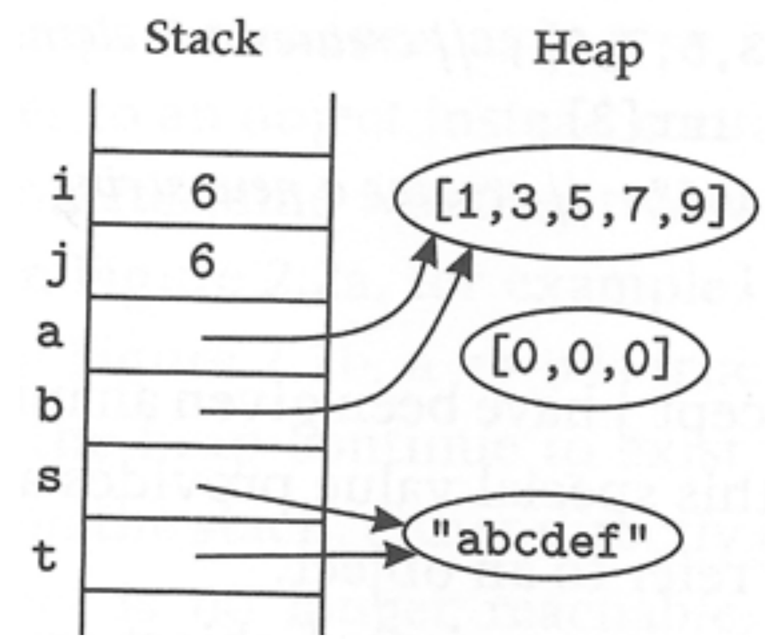
```
int i = 6;
int j; //uninitialized
int [] a = {1, 3, 5, 7, 9}; //
creates a 5-element array
int [] b = new int[3];
String s = "abcdef";
String t = null;

j = i;
b = a;
t = s;
```

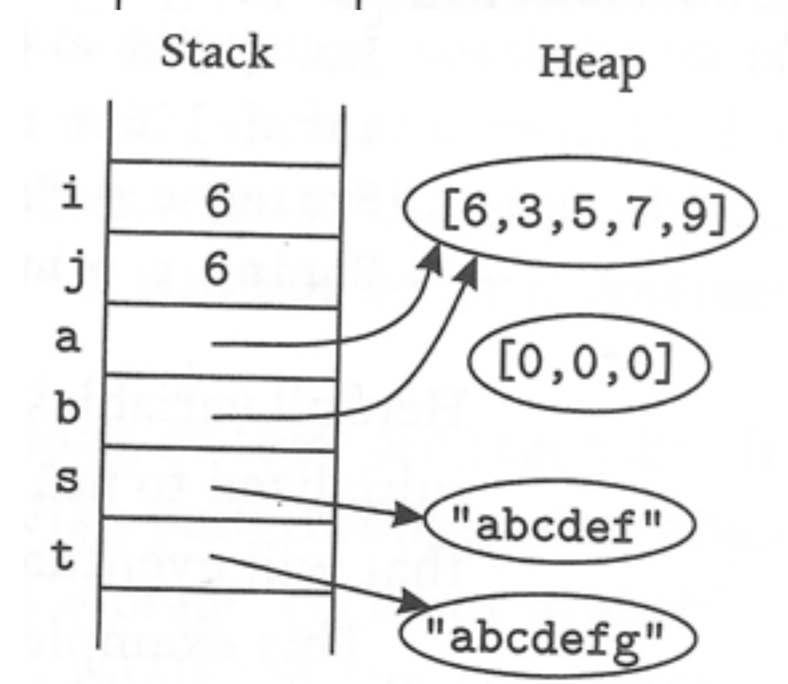




```
int i = 6;

int j; //uninitialized

int [] a = {1, 3, 5, 7, 9}; // creates a 5-element array

int [] b = new int[3];

String s = "abcdef";

String t = null;


j = i;

b = a;

t = s;


t = t +"g";

a[0] = i;
```







Understanding an implementation

- abstraction function
- representation invariant

Abstraction Function

[Liskov]

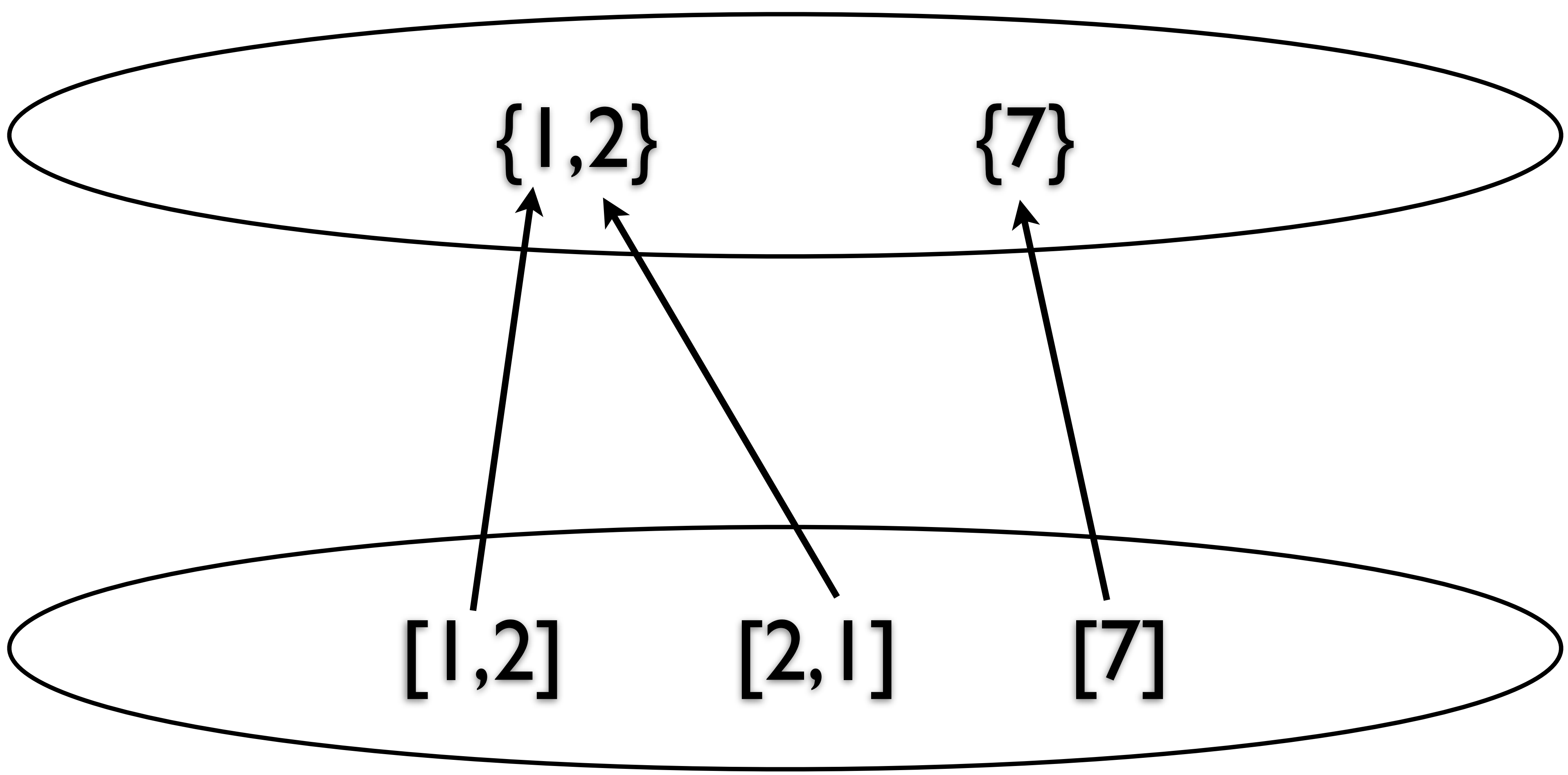
This relationship can be defined by a function called the abstraction function that maps from the instance variables that make up the rep of an object to the abstract object being represented:

$$AF : C \rightarrow A$$

Specifically, the abstraction function AF maps from a concrete state (i.e., the state of an object of the class C) to an abstract state (i.e., the state of an abstract object). For each object c belonging to C, AF(c) is the state of the abstract object $a \in A$ that c represents.

IntSet Abstraction Function

[Liskov]



AF(c) = {c.els[i].intValue | 0<=i<c.els.size}

MySet Recipe Implementation Abstraction Function

$$AF(MySet.empty()) = \{\}$$

$$AF(MySet.insert(s0, k0)) = \left\{ \begin{array}{ll} AF(s0) & |MySet.contains(s0, k0) \\ k0 + AF(s0) & |else \end{array} \right\}.$$

Northeastern University
College *of* Computer and Information Science

MySet Recipe Implementation Abstraction Function

$$AF(Empty()) = \{\}$$

$$AF(Insert(s0, k0)) = \left\{ \begin{array}{ll} AF(s0) & |s0.contains(k0) \\ k0 + AF(s0) & |else \end{array} \right\}.$$

Immutable Queue Abstraction Function

Immutable Queue Abstraction Function

$$AF(Queue.empty()) = \text{``''}$$

$$AF(Queue.enqueue(q, i)) = q.toString() + \text{``(''} + i + \text{``)''}$$

Immutable Queue with Dynamic Methods Abstraction Function

Immutable Queue with Dynamic Methods Abstraction Function

$$AF(Queue.empty()) = \text{``''}$$

$$AF(q.enqueue(i)) = q.toString() + \text{``(''} + i + \text{``)''}$$

Mutable Queue Abstraction Function

Mutable Queue Implemented as ArrayList Abstraction Function

$$AF(Queue.empty()) = \text{``''}$$

$$AF(c) = \{\text{``(''} + c.q.get(i).intValue() + \text{``)''} \mid 0 <= i < c.q.size()\}$$

Mutable Queue Abstraction Function

```
// A typical Queue of integers is
// {k0, k1, ..., kn}
// with k0 as the first element added, and
// kn as the last element added
//
// The abstraction function is
// AF(queue) =
//   { queue.q[0] = k0, queue.q[q.size() - 1] = kn |
//     for queue = {k0, k1, ..., kn}}


// For a queue created by adding elements k0, k1, k2, …
//   in this order
// the abstraction function is
//   AF(queue) = {k0, k1, k2, …}
//   where queue.q[i] = ki for 0 <= i < queue.q.size()
```

Rep Invariant

A statement of a property that all legitimate objects satisfy is called a *representation invariant*, or *rep invariant.* A rep invariant I is a predicate

$$I : C \rightarrow \text{boolean}$$

that is true of legitimate objects.

IntSet Rep Invariant

```
// The rep invariant is:
//   c.els != null &&
//   all elements of c.els are Integers &&
//   there are no duplicates in c.els
```

Mutable Queue Implemented as ArrayList Rep Invariant

Mutable Queue Implemented as ArrayList Rep Invariant

```
// The rep invariant is:
//   c.q != null &&
//   all elements of c.q are Integers
```

Class Invariant

Class Invariant

- An assertion about an object's state that is true for the lifetime of that object.

How much to include in rep invariant? [Liskov p.107]

There is an issue concerning how much to say in a rep invariant. <u>A rep invariant should express all constraints on which the operations depend.</u> A good way to think of this is to imagine that the operations are to be implemented by different people who cannot talk to one another; the rep invariant must contain all the constraints that these various implementors depend on. However, it need not state additional constraints.

Implementing Rep Invariant

`repOk`

IntSet repOk
[Liskov]

```
// The rep invariant is:
//  c.els != null &&
//  all elements of c.els are Integers &&
//  there are no duplicates in c.els

public boolean repOk() {
  if (els == null) return false;
  for (int i = 0; i < els.size(); i++) {
    Object x = els.get(i);
    if (!(x instanceof Integer)) return false;
    for (int j = i+1; j < els.size(); j++)
      if(x.equals(els.get(j))) return false;
  }
  return true;
}
```

`clone` method

Binary Search

Binary Search

If a set S is represented by a sorted linear sequence, then we can determine whether x is an element of S in logarithmic time by using binary search.

Binary Search

1	3	7	11	19	22	25	33	41	49	51	57	60

Search for 51

Binary Search

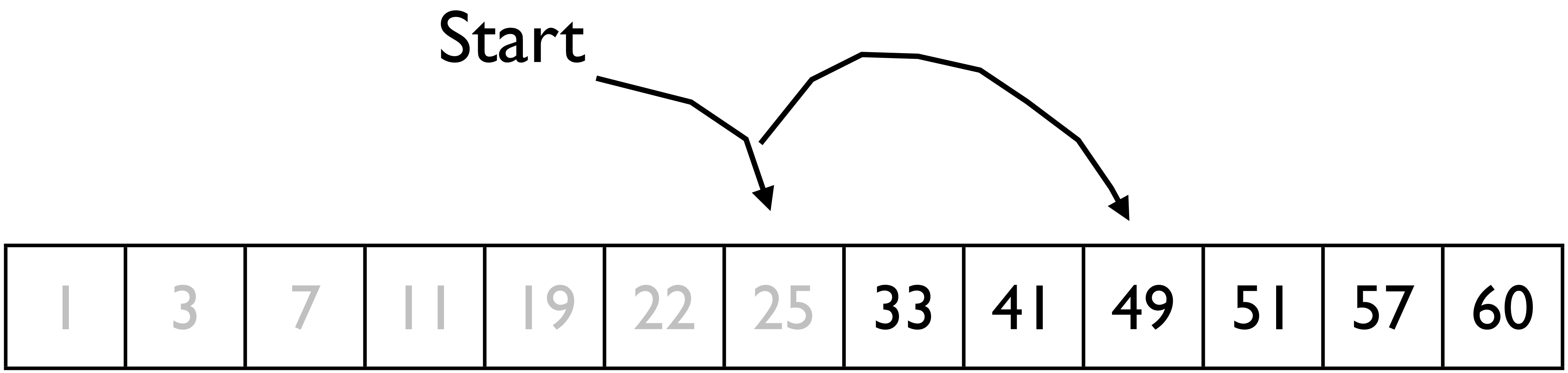
Start

1	3	7	11	19	22	25	33	41	49	51	57	60

Search for 51

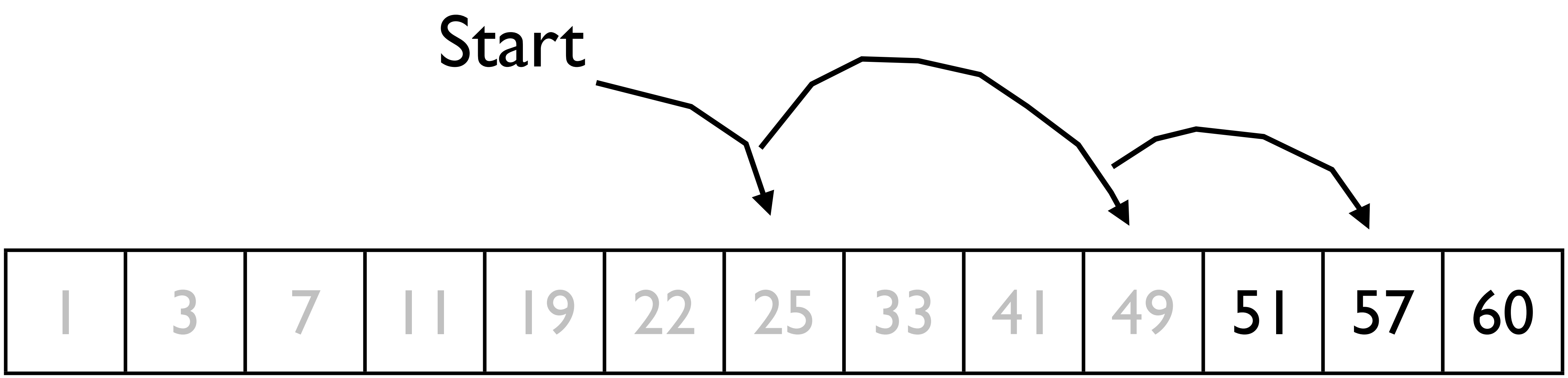
Northeastern University
College *of* Computer and Information Science

Binary Search



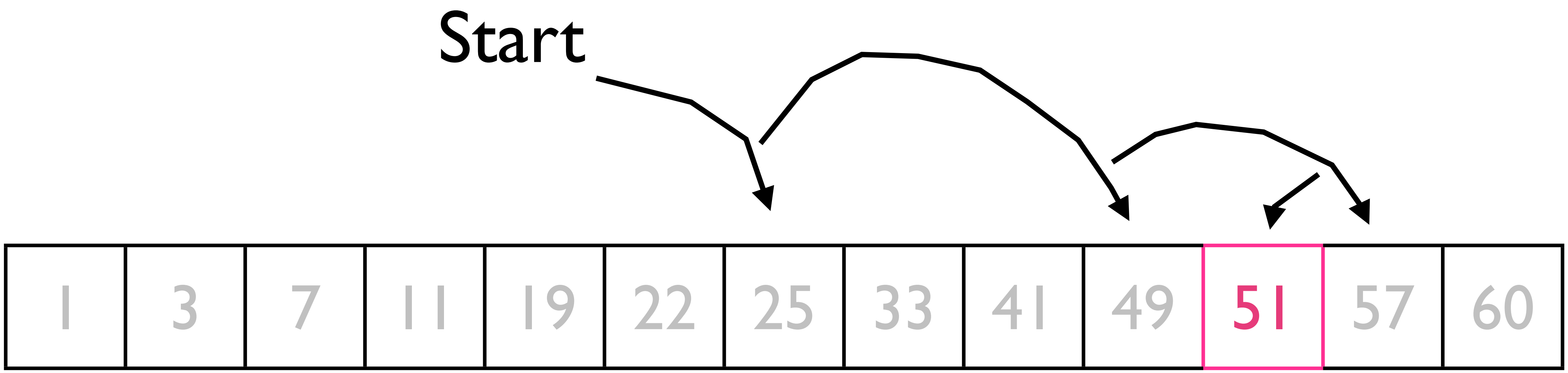
Search for 51

Binary Search



Search for 51

Binary Search



Search for 51

FOUND

Binary Search

1	3	7	11	19	22	25	33	41	49	51	57	60

Search for 53

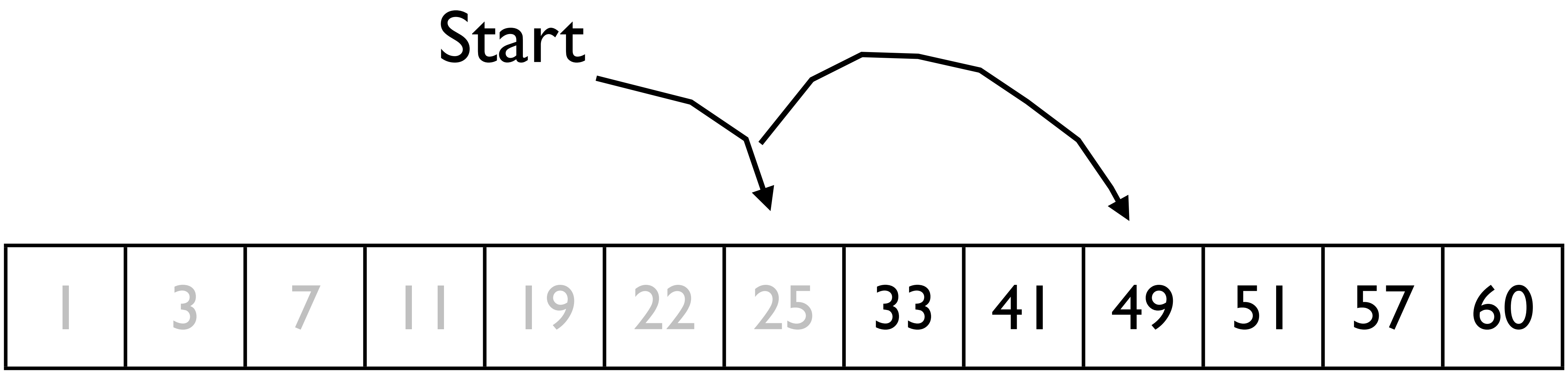
Binary Search

Start

1	3	7	11	19	22	25	33	41	49	51	57	60

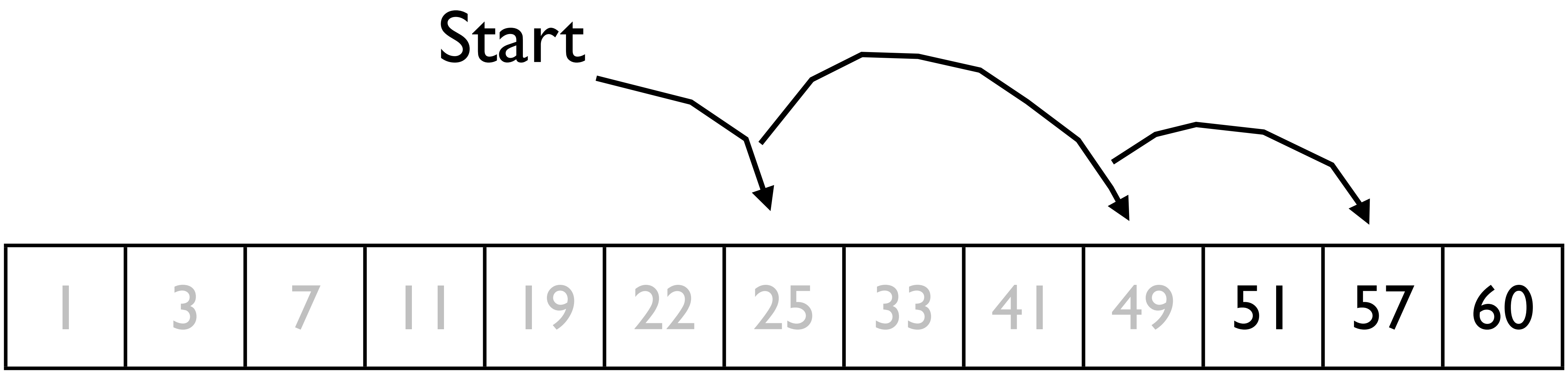
Search for 53

Binary Search



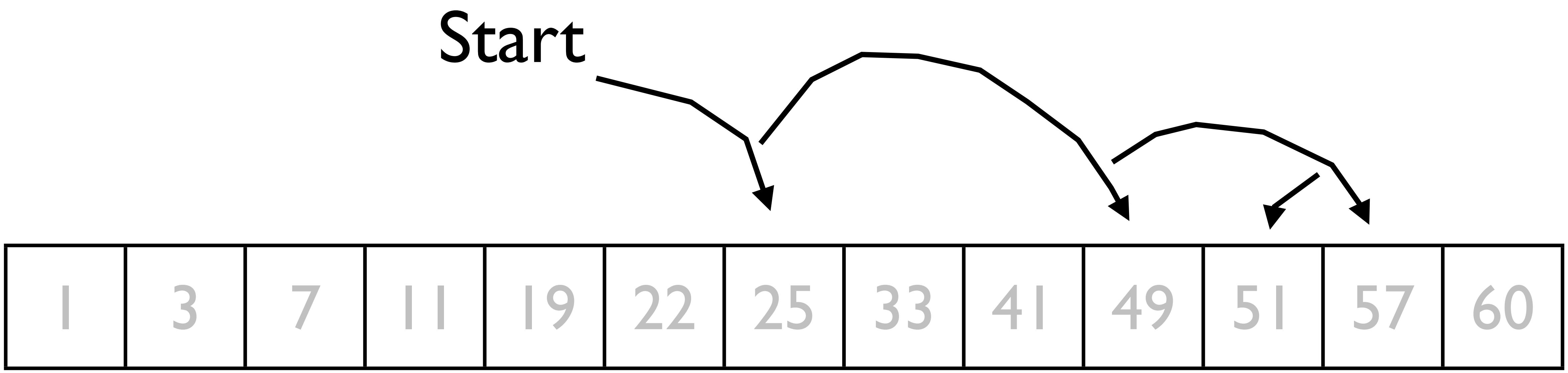
Search for 53

Binary Search



Search for 53

Binary Search



Search for 53

NOT FOUND

```java
/**
 * PRE: min <= max
 * PRE: 0 <= min, max <= data.length
 * @param data sorted array of ints
 * @param min min index to search
 * @param max max index to search
 * @param target value searching for in data
 * @return index of target in data,
 *       or -1 if target is not in data
 */
public static int binarySearch(int[] data, int min, int max,
                               int target){




}
```