University of BRISTOL

Programming and Algorithms II

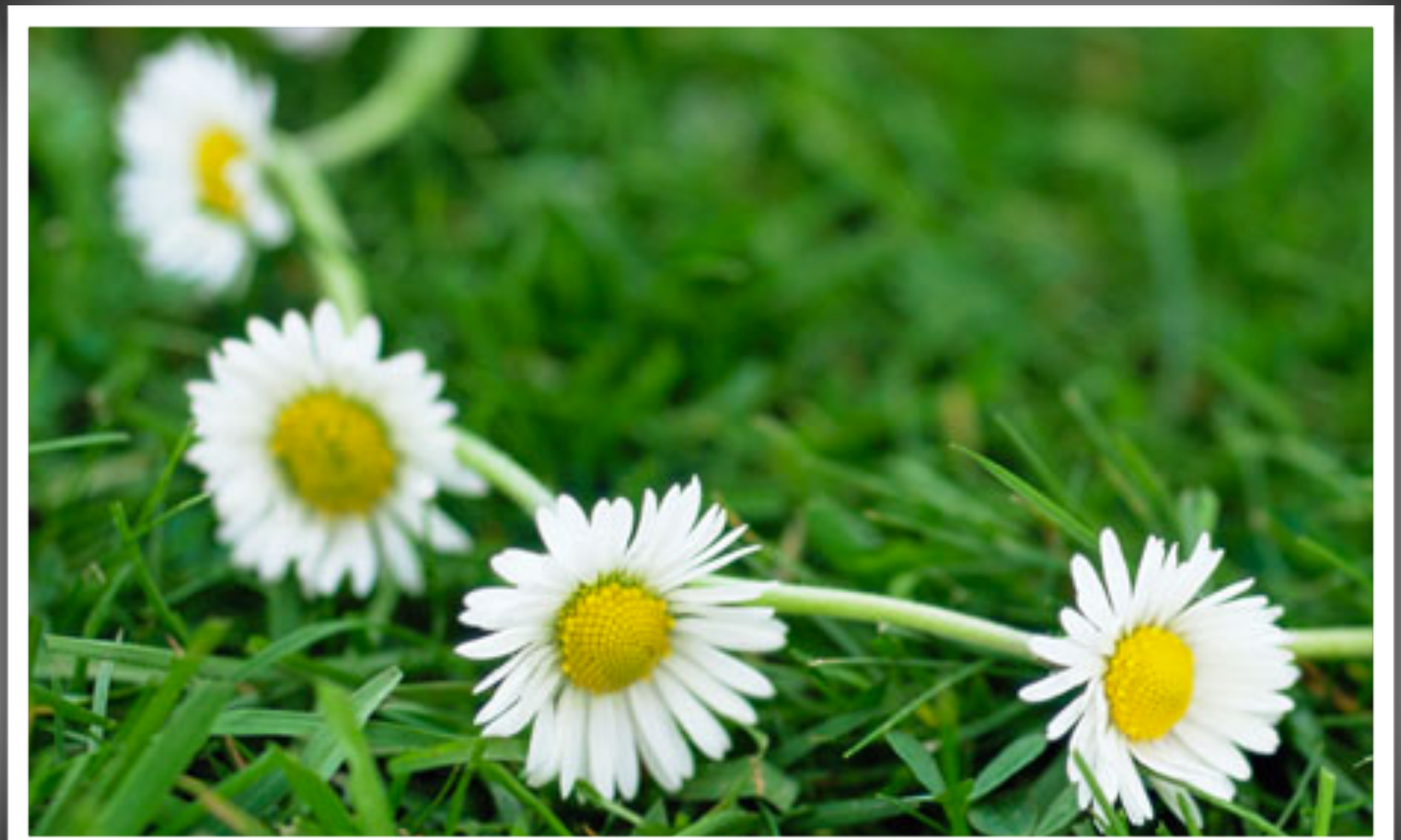
Lecture 7: Abstract Data Types II

Nicolas Wu

nicolas.wu@bristol.ac.uk

Department of Computer Science
University of Bristol

Linked Lists



Linked Lists

- Linked lists are an extremely versatile structure
- Let's take them for a test-run and use them to implement stacks
- While we're at it, let's do it to a generic implementation that doesn't suffer from the ugliness of the ArrayStack version
- We'll call it ListStack, since it's based on Lists

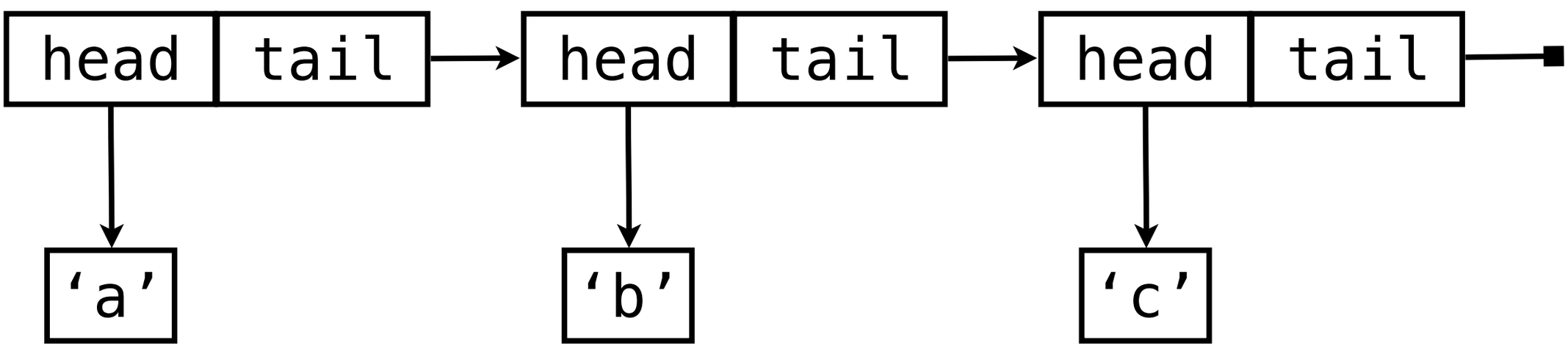
Linked Lists

- Since Stack is an interface, we can instantiate one just as we would a class
- Multiple implementations mean we get to pick-and-choose the one that suits us best in different circumstances

```
Stack<Integer> stack1 = new ArrayStack<Integer>();
Stack<Integer> stack2 = new ListStack<Integer>();
```

Linked Lists

- A linked list is a structure made up of *links*
- A link is a pair of references:
  - *head* points to a data element
  - *tail* points to another link



Linked Lists

- A Link can be implemented by a class

```
class Link<X> {
  public final X head;
  public final Link<X> tail;

  public Link(X head, Link<X> tail) {
    this.head = head;
    this.tail = tail;
  }
}
```

Linked Lists

- A LinkedStack contains a single link that points to the top of the stack

```
class LinkedStack<X> implements Stack<X> {
  private Link<X> xs = null;
  …

}
```

Linked Lists

- A LinkedStack contains a single link that points to the top of the stack

```
// abs : LinkedStack → [X]
// abs this = list (this.xs)
//   where
//     list null = []
//     list link = link.head : list link.tail
class LinkedStack<X> implements Stack<X> {
  private Link<X> xs = null;
  …

}
```

- The abstraction function pushes us further into the functional world

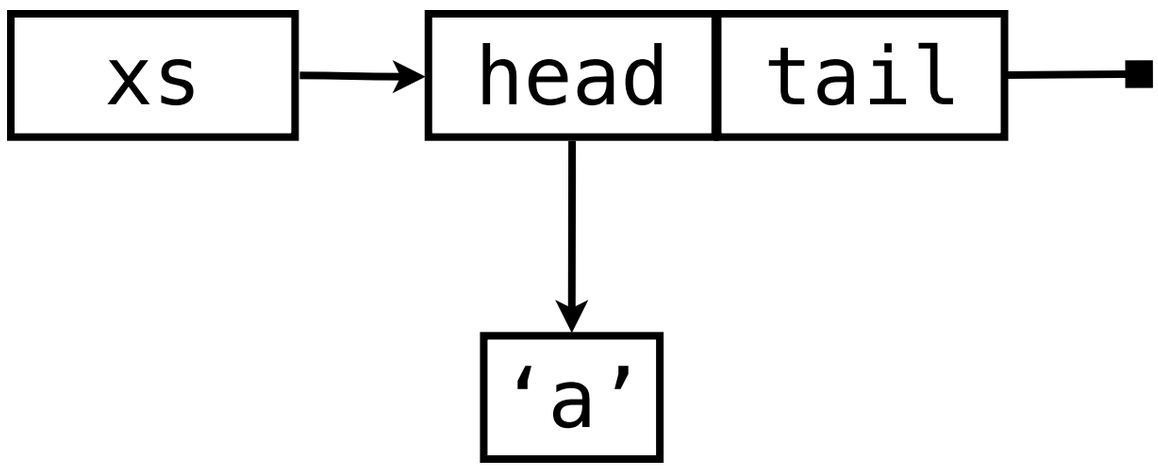
Linked Lists

```
// abs : LinkedStack → [X]
// abs this = list (this.xs)
//   where
//     list null = []
//     list link = link.head : list link.tail
```

[]

Linked Lists

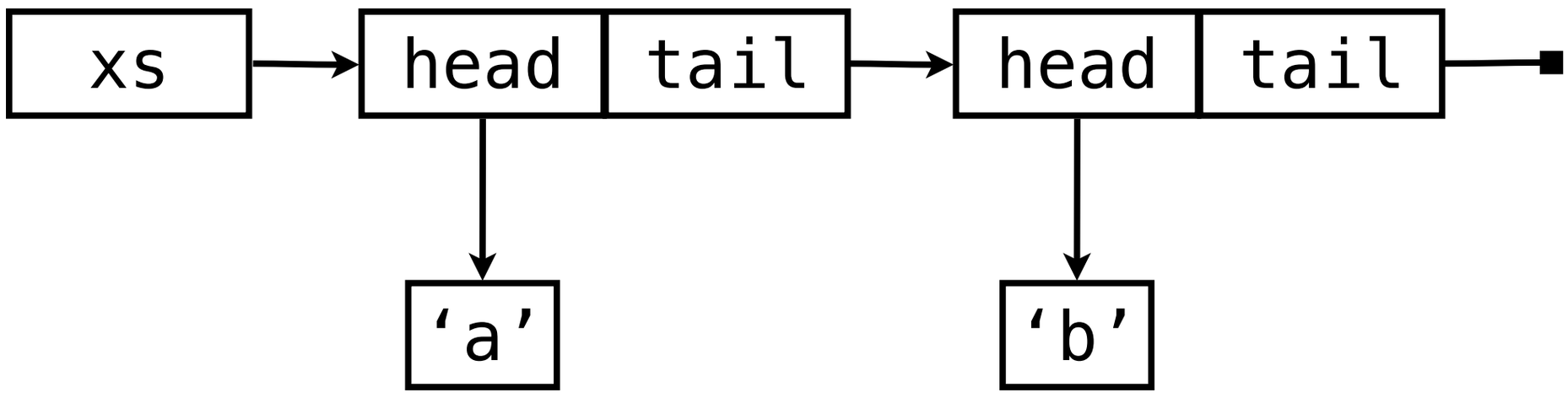
```
// abs : LinkedStack → [X]
// abs this = list (this.xs)
//   where
//     list null = []
//     list link = link.head : list link.tail
```



‘a’ : []

Linked Lists

```
// abs : LinkedStack → [X]
// abs this = list (this.xs)
//   where
//     list null = []
//     list link = link.head : list link.tail
```



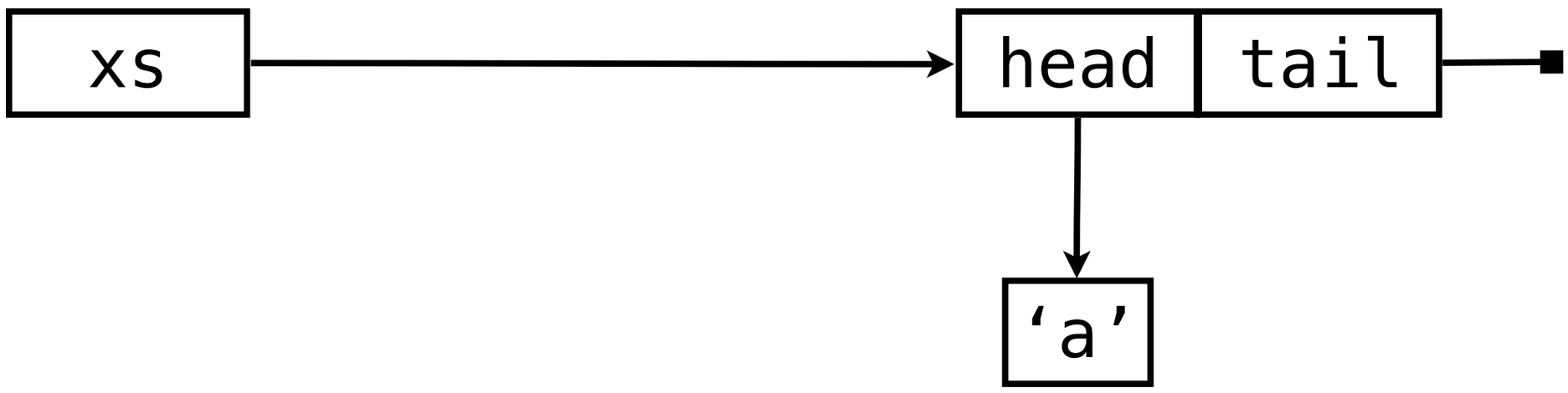
‘a’ : ‘b’ : []

Linked Lists

```
// pre:  true
// post: xs = x:xs0
public void push(X x) {
    xs = new Link(x, xs);
}
```

Linked Lists

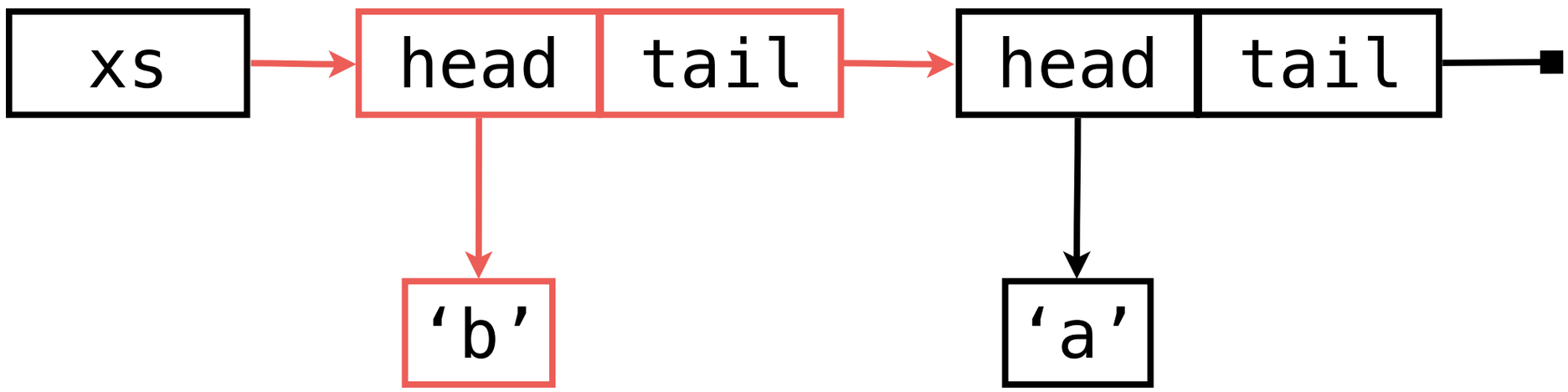
```
// pre:  true
// post: xs = x:xs0
public void push(X x) {
    xs = new Link(x, xs);
}
```



‘a’ : []

Linked Lists

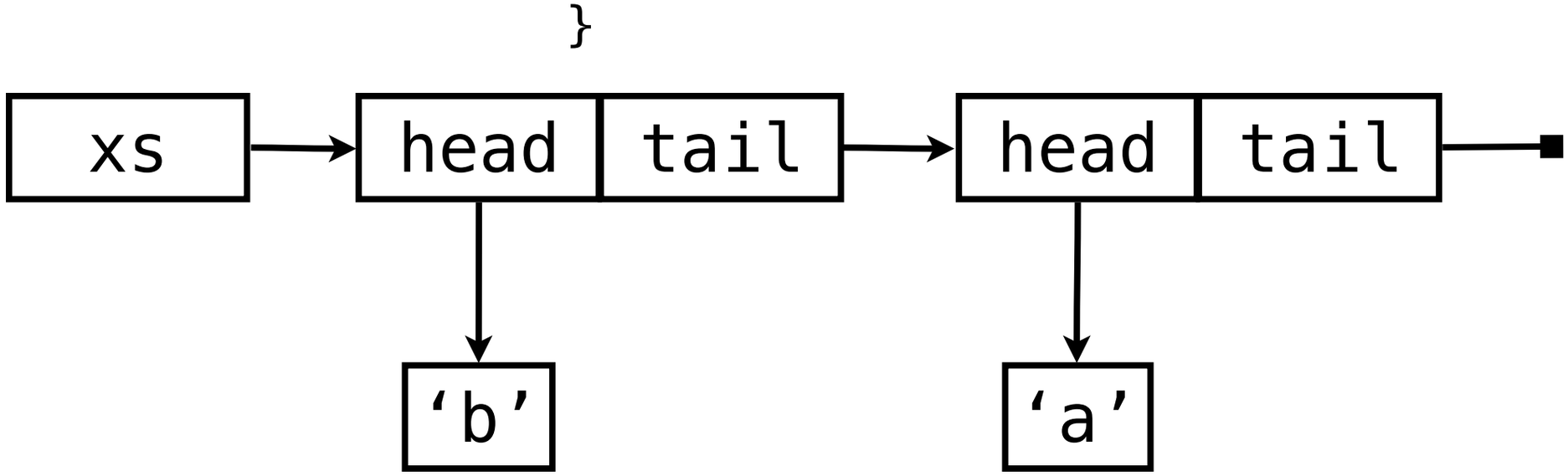
```
// pre:  true
// post: xs = x:xs0
public void push(X x) {
    xs = new Link(x, xs);
}
```



‘b’ : ‘a’ : []

Linked Lists

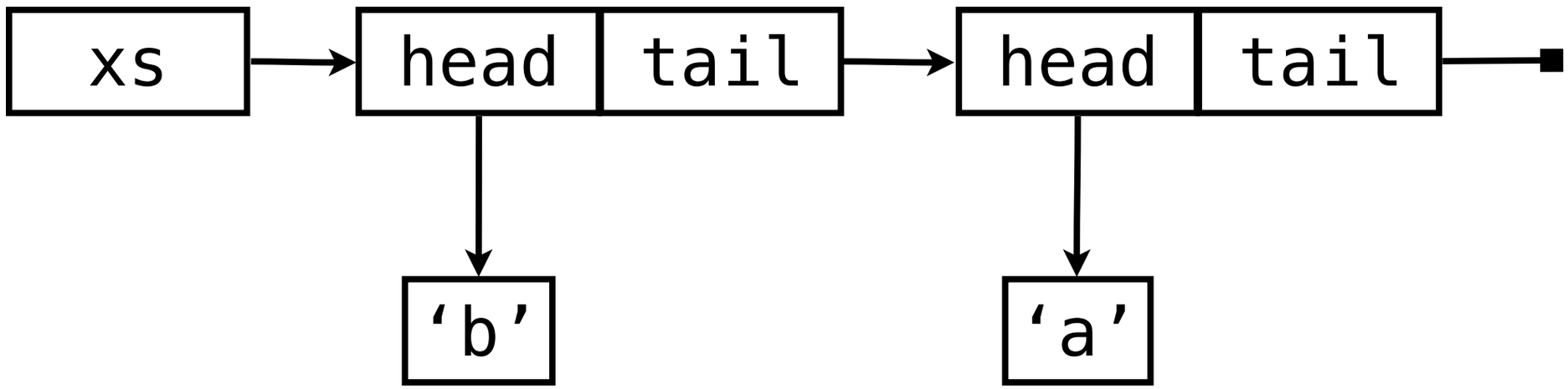
```
// pre:    xs ≠ []
// post:   x:xs = xs0
// return: x
public X pop() {
  assert xs != null;
  X x = xs.head;
  xs = xs.tail;
  return x;
}
```



‘b’ : ‘a’ : []

Linked Lists

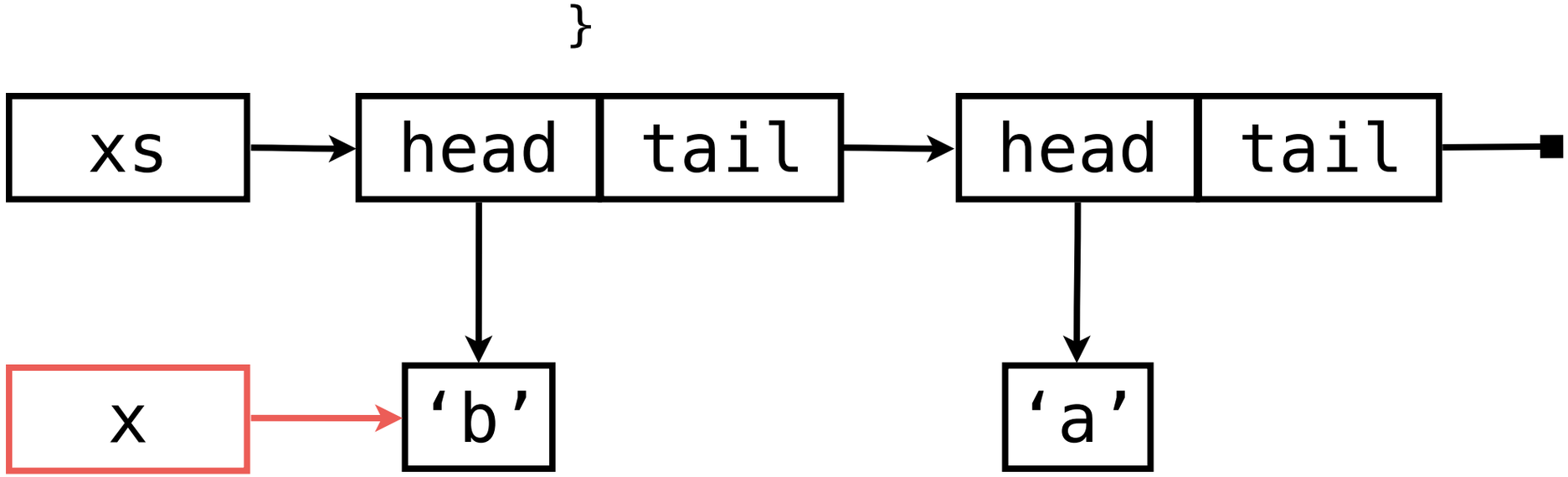
```
// pre:    xs ≠ []
// post:   x:xs = xs0
// return: x
public X pop() {
  assert xs != null;
  X x = xs.head;
  xs = xs.tail;
  return x;
}
```



‘b’ : ‘a’ : []

Linked Lists

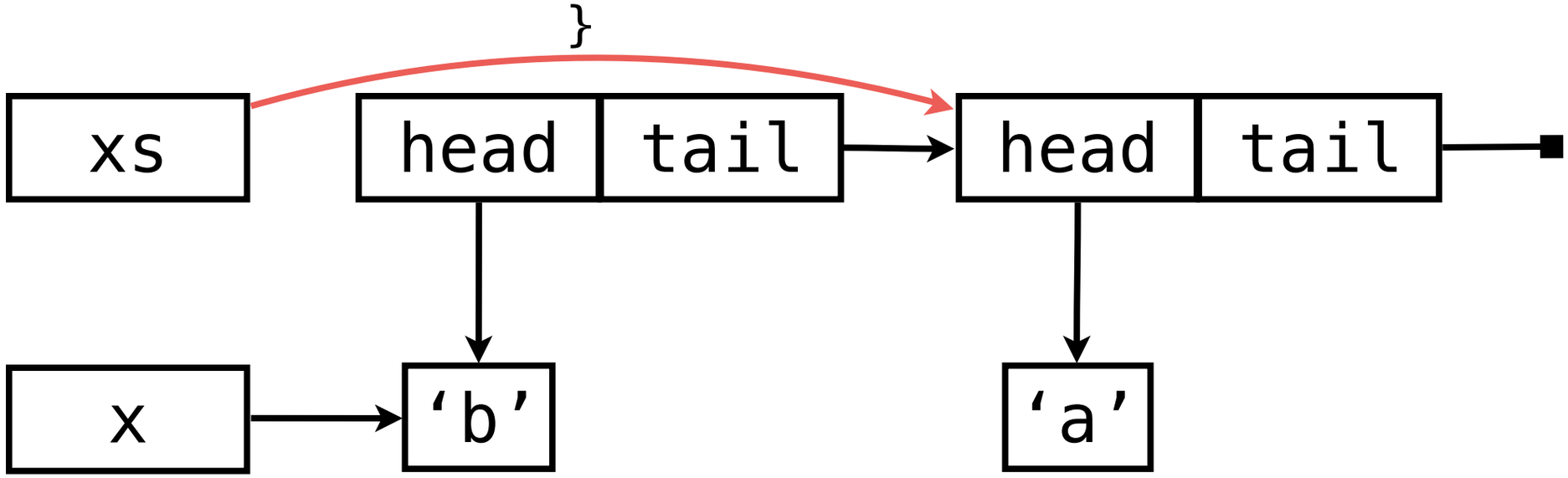
```
// pre:    xs ≠ []
// post:   x:xs = xs0
// return: x
public X pop() {
  assert xs != null;
  X x = xs.head;
  xs = xs.tail;
  return x;
}
```



‘b’ : ‘a’ : []

Linked Lists

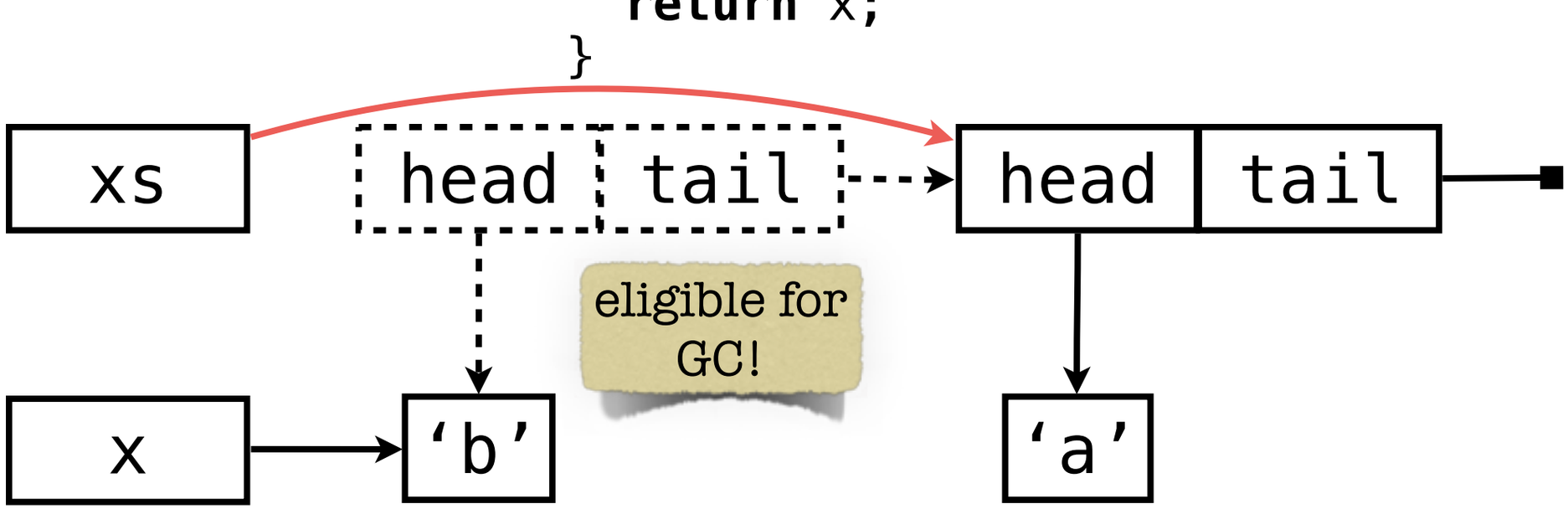
```
// pre:    xs ≠ []
// post:   x:xs = xs0
// return: x
public X pop() {
  assert xs != null;
  X x = xs.head;
  xs = xs.tail;
  return x;
}
```



'a' : []

Linked Lists

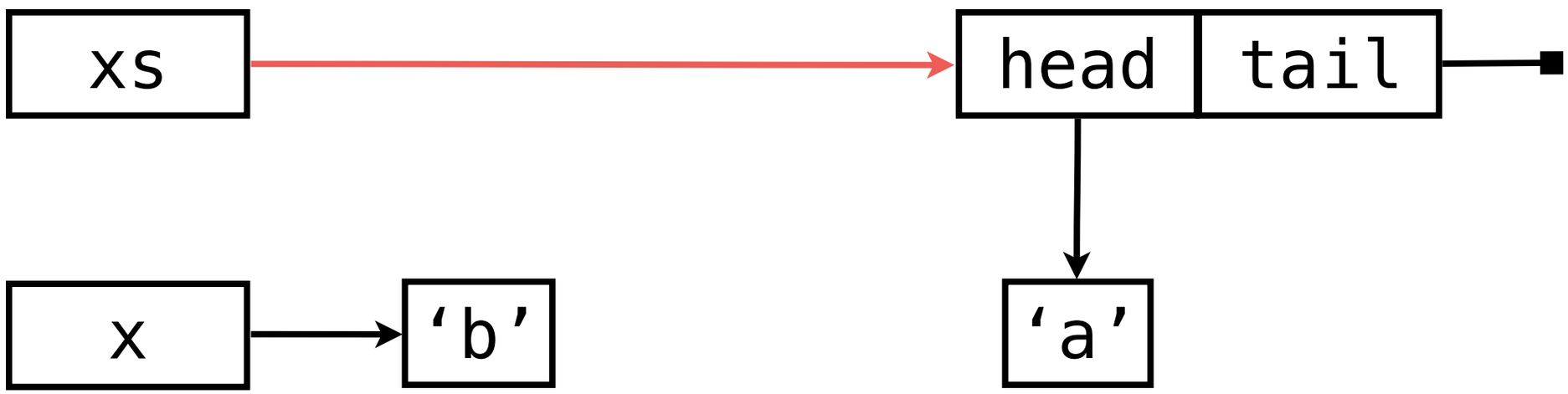
```
// pre:    xs ≠ []
// post:   x:xs = xs0
// return: x
public X pop() {
  assert xs != null;
  X x = xs.head;
  xs = xs.tail;
  return x;
}
```



'a' : []

Linked Lists

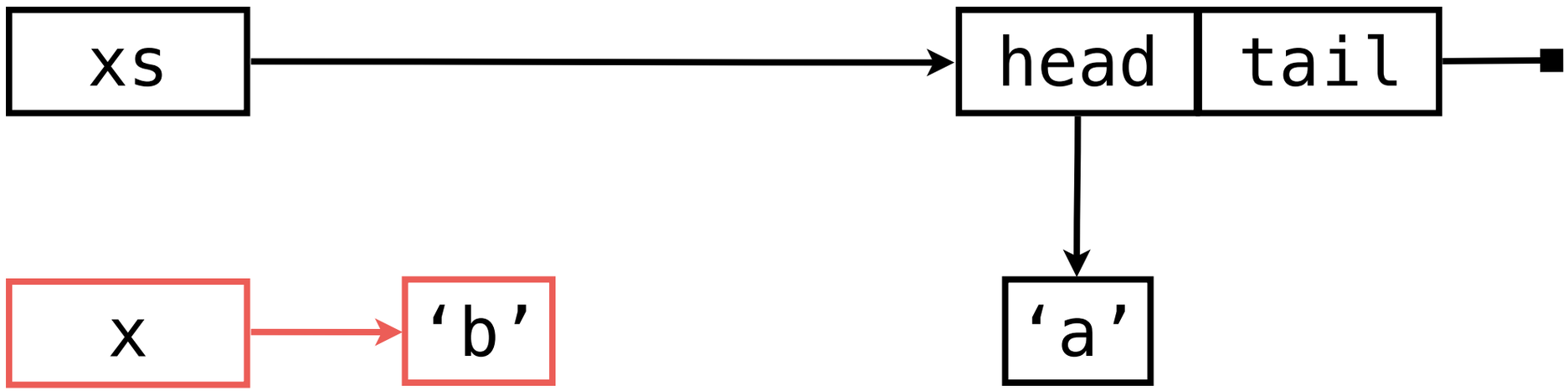
```
// pre:    xs ≠ []
// post:   x:xs = xs0
// return: x
public X pop() {
  assert xs != null;
  X x = xs.head;
  xs = xs.tail;
  return x;
}
```



'a' : []

Linked Lists

```
// pre:    xs ≠ []
// post:   x:xs = xs0
// return: x
public X pop() {
  assert xs != null;
  X x = xs.head;
  xs = xs.tail;
  return x;
}
```



‘a’ : []

Linked Lists

```
// abs: LinkedStack<X> → [X]
// abs this = list (this.xs)
//   where
//     list null = []
//     list link = link.head : list link.tail
class LinkedStack<X> implements Stack<X> {
  private Link<X> xs = null;

  public void push(X x) {
    xs = new Link<X>(x, xs);
  }

  public X pop() {
    assert xs != null;
    X x = xs.head;
    xs = xs.tail;
    return x;
  }

  public X peek() {
    assert xs != null;
    return xs.head;
  }

  public boolean empty() {
    return (xs == null);
  }
}
```

Linked Lists

```
// abs: LinkedStack<X> → [X]
// abs this = list (this.xs)
//   where
//     list null = []
//     list link = link.head : list link.tail
class LinkedStack<X> implements Stack<X> {
  private Link<X> xs = null;

  public void push(X x) {
    xs = new Link<X>(x, xs);
  }

  public X pop() {
    assert xs != null;
    X x = xs.head;
    xs = xs.tail;
    return x;
  }

  public X peek() {
    assert xs != null;
    return xs.head;
  }

  public boolean empty() {
    return (xs == null);
  }
}
```

Q. Is this code any good?

Linked Lists

```
// abs: LinkedStack<X> → [X]
// abs this = list (this.xs)
//   where
//     list null = []
//     list link = link.head : list link.tail
class LinkedStack<X> implements Stack<X> {
  private Link<X> xs = null;

  public void push(X x) {
    xs = new Link<X>(x, xs);
  }

  public X pop() {
    assert xs != null;
    X x = xs.head;
    xs = xs.tail;
    return x;
  }

  public X peek() {
    assert xs != null;
    return xs.head;
  }

  public boolean empty() {
    return (xs == null);
  }
}
```

Q. Is this code any good?

A. Well it meets the spec

Linked Lists

```
// abs: LinkedStack<X> → [X]
// abs this = list (this.xs)
//   where
//     list null = []
//     list link = link.head : list link.tail
class LinkedStack<X> implements Stack<X> {
  private Link<X> xs = null;

  public void push(X x) {
    xs = new Link<X>(x, xs);
  }

  public X pop() {
    assert xs != null;
    X x = xs.head;
    xs = xs.tail;
    return x;
  }

  public X peek() {
    assert xs != null;
    return xs.head;
  }

  public boolean empty() {
    return (xs == null);
  }
}
```

Q. Is this code any good?

A. Well it meets the spec

But it has nulls!

The Zen of Purity

A tale of a youngling and his Master

Master, I thought nulls were evil!

Indeed, they are a great evil

Then why did you show me evil?

Because there is much evil in the world

Can the evil be avoided?

Yes, the truth lies in the way of purity

Teach it to me!

I will teach it, though others call it folly

- First, we must construct a list ADT

```
// state: xs : [X]
interface List<X> {
  // pre:    xs ≠ []
  // post:   xs = xs0
  // return: head xs
  public X head();

  // pre:    xs ≠ []
  // post:   xs = xs0
  // return: tail xs
  public List<X> tail();

  // post:   xs = xs0
  // return: xs ≡ []
  public boolean empty();
}
```

- Now we start with the base case: empty lists

```
// ABS:
//   abs : Empty<X> -> [X]
//   abs this = []
class Empty<X> implements List<X> {
  public X head()         { return null; }
  public List<X> tail()   { return null; }
  public boolean empty() { return true; }
}
```

But Master! I see nulls!

Patience, youngling, enlightenment will come

- The other case is for adding an element to a list

```
// ABS:
//  abs : Cons<X> -> [X]
//  abs this = this.head : abs (this.tail)
class Cons<X> implements List<X> {
  private final X head;
  private final List<X> tail;
  public Cons(X head, List<X> tail) {
    this.head = head;
    this.tail = tail;
  }

  public X head()          { return head; }
  public List<X> tail()    { return tail; }
  public boolean empty() { return false; }
}
```

- Finally, we can implement a Stack with no nulls

```
// ABS:
//   abs : ListStack<X> -> [X]
//   abs this = abs (this.xs)
abstract class ListStack<X> implements Stack<X> {
  private List<X> xs = new Empty<X>();

  public void push(X x) {
    xs = new Cons<>(x, xs);
  }

  public X pop() {
    assert !xs.empty();
    X x = xs.head();
    xs = xs.tail();
    return x;
  }

  public X peek() {
    assert !xs.empty();
    X x = xs.head();
    return x;
  }

  public boolean empty() {
    return (xs.empty());
  }
}
```

I see that we have hidden nulls so others will not have to see the evil we have done

That is correct, we have mastered evil so that others need not

But Master! Must evil always exist?

No, there is yet a purer way to follow

teach it to me!

I will teach it, though others call it folly

Youngling, where do you think the nulls come from in List?

That is easy! The Empty class!

That is correct, but why are they there?

Because there is no way to implement head or tail

Why must they be implemented?

The interface required them!

Then we must change the interface

- We must make head and tail safer by using Optional

```
import java.util.Optional;

// state: xs : [X]
interface ListSafe<X> {
  // pre:    xs ≠ []
  // post:   xs = xs0
  // return: head xs
  public Optional<X> head();

  // pre:    xs ≠ []
  // post:   xs = xs0
  // return: tail xs
  public Optional<List<X>> tail();

  // post:   xs = xs0
  // return: xs ≡ []
  public boolean empty();
}
```

- Now we can eradicate the evil nulls

```
import java.util.Optional;
// ABS:
//   abs : Empty<X> -> [X]
//   abs this = []
class EmptySafe<X> implements ListSafe<X> {
  public Optional<X> head()        { return Optional.empty(); }
  public Optional<List<X>> tail() { return Optional.empty(); }
  public boolean empty()           { return true; }
}
```

- We must remember to return optional values instead

```
// ABS:
//  abs : Cons<X> -> [X]
//  abs this = this.head : abs (this.tail)
class ConsSafe<X> implements ListSafe<X> {
  private final X head;
  private final List<X> tail;
  public ConsSafe(X head, List<X> tail) {
    this.head = head;
    this.tail = tail;
  }

  public Optional<X> head()       { return Optional.of(head); }
  public Optional<List<X>> tail() { return Optional.of(tail); }
  public boolean empty()          { return false; }
}
```

But Master! Stack doesn't use Optionals!

This is true

Must I change that too?

Yes, your interface was unsafe, it too neds to change

That sounds like a lot of work!

The way of purity can be hard

Must it always be this way?

No, there are better languages than Java

Sets

Sets

- A *set* is a mathematical collection of objects

```
data Set a
```

- An empty set contains no elements

```
∅ : Set a
```

- A singleton set contains one element

```
{–} : a → Set a
```

- Two sets can be combined with their union

```
(∪) : Set a → Set a → Set a
```

Sets

- We often abuse notation and write

```
{ x1, x2, … , xn } = {x1} ∪ {x2} ∪ … ∪ {xn}
```

- Elements can be checked for membership

```
(∈) : a → Set a → Bool
x ∈ ∅           = False
x ∈ {y}         = x ≡ y
x ∈ (us ∪ vs) = (x ∈ us) ∨ (x ∈ vs)
```

- The negation is convenient too

```
x ∉ us = ¬(x ∈ us)
```

Sets

- The subset relation holds if all elements in one set are contained in another

```
(⊆) : Set a → Set a → Bool
us ⊆ vs = ∀ u . u ∈ us ⇒ u ∈ vs
```

- Equality of sets can be determined in terms of subsets

```
(≡) : Set a → Set a → Bool
us ≡ vs = us ⊆ vs ∧ vs ⊆ us
```

Sets

- As usual, we give the state space and initial state of the ADT

```
// state:  xs : Set X
// init:   xs = ∅
interface Set<X> {
  …
}
```

Sets

- Now some methods: *insert*, *delete*, *empty*, *contains* and *size*

```
// state:  xs : Set X
// init:   xs = ∅
interface Set<X> {

  // pre:   x ∉ xs
  // post:  xs = {x} ∪ xs₀
  public void insert(X x);

  …
}
```

Sets

- The *empty*, *contains* and *size* methods are quite straightforward

```
// post:   xs = ∅
public void empty();

// post:   xs = xs0
// return: x ∈ xs
public boolean contains(X x);

// post:   xs = xs0
// return: |xs|
public int size();
```

Sets

- Finding the exact pre and post combination can be a subtle art: consider *delete*

```
public void delete(X x);
```

Q. Guess the problem:

```
// pre:  true
// post: x ∉ xs
```

Sets

- Finding the exact pre and post combination can be a subtle art: consider *delete*

```
public void delete(X x);
```

Q. Guess the problem:

```
// pre:  true
// post: x ∉ xs
```

A. We could add things to xs

Sets

- Finding the exact pre and post combination can be a subtle art: consider *delete*

```
public void delete(X x);
```

Q. Guess the problem:

```
// pre:  true
// post: x ∉ xs

// pre:  true
// post: x ∉ xs ∧ xs ⊆ xs0
```

A. We could add things to xs

Sets

- Finding the exact pre and post combination can be a subtle art: consider *delete*

```
public void delete(X x);
```

Q. Guess the problem:

```
// pre:  true
// post: x ∉ xs

// pre:  true
// post: x ∉ xs ∧ xs ⊆ xs0
```

A. We could add things to xs

A. We can set xs = ∅

Sets

- Finding the exact pre and post combination can be a subtle art: consider *delete*

```
public void delete(X x);
```

Q. Guess the problem:

```
// pre:  true
// post: x ∉ xs

// pre:  true
// post: x ∉ xs ∧ xs ⊆ xs0

// pre:  true
// post: x ∉ xs ∧ xs0 = {x} ∪ xs
```

A. We could add things to xs

A. We can set xs = ∅

Sets

- Finding the exact pre and post combination can be a subtle art: consider *delete*

```
public void delete(X x);
```

Q. Guess the problem:

```
// pre:  true
// post: x ∉ xs

// pre:  true
// post: x ∉ xs ∧ xs ⊆ xs0

// pre:  true
// post: x ∉ xs ∧ xs0 = {x} ∪ xs
```

A. We could add things to xs

A. We can set xs = ∅

A. Was x in xs_0?

Sets

- Finding the exact pre and post combination can be a subtle art: consider *delete*

```
public void delete(X x);
```

Q. Guess the problem:

```
// pre:  true
// post: x ∉ xs

// pre:  true
// post: x ∉ xs ∧ xs ⊆ xs0

// pre:  true
// post: x ∉ xs ∧ xs0 = {x} ∪ xs

// pre:  x ∈ xs
// post: x ∉ xs ∧ xs0 = {x} ∪ xs
```

A. We could add things to xs

A. We can set xs = ∅

A. Was x in xs_0?

Sets

- Finding the exact pre and post combination can be a subtle art: consider *delete*

```
public void delete(X x);
```

Q. Guess the problem:

```
// pre:  true
// post: x ∉ xs

// pre:  true
// post: x ∉ xs ∧ xs ⊆ xs0

// pre:  true
// post: x ∉ xs ∧ xs0 = {x} ∪ xs

// pre:  x ∈ xs
// post: x ∉ xs ∧ xs0 = {x} ∪ xs
```

A. We could add things to xs

A. We can set xs = ∅

A. Was x in xs_0?

A. None?!

Sets

```
// state:  xs : Set X
// init:   xs = ∅
interface Set<X> {
  // pre:   x ∉ xs
  // post:  xs = {x} ∪ xs0
  public void insert(X x);

  // pre:  x ∈ xs
  // post: x ∉ xs ∧ xs0 = {x} ∪ xs
  public void delete(X x);

  // post:   xs = ∅
  public void empty();

  // post:   xs = xs0
  // return: x ∈ xs
  public boolean contains(X x);

  // post:   xs = xs0
  // return: |xs|
  public int size();
}
```

- Specify the state
- Set initialisation
- Preconditions and postconditions
- Return values

Concrete Sets

Concrete Sets

```
class ArrayIntSet implements IntSet {
  private int[] values;
  private int size;
  private final int N = 100;

  …
}
```

Concrete Sets

- The abstraction function returns an ADT

```
abs : Set<X> → Set X
abs this = { this.values[i] | 0 ≤ i < this.size }
```

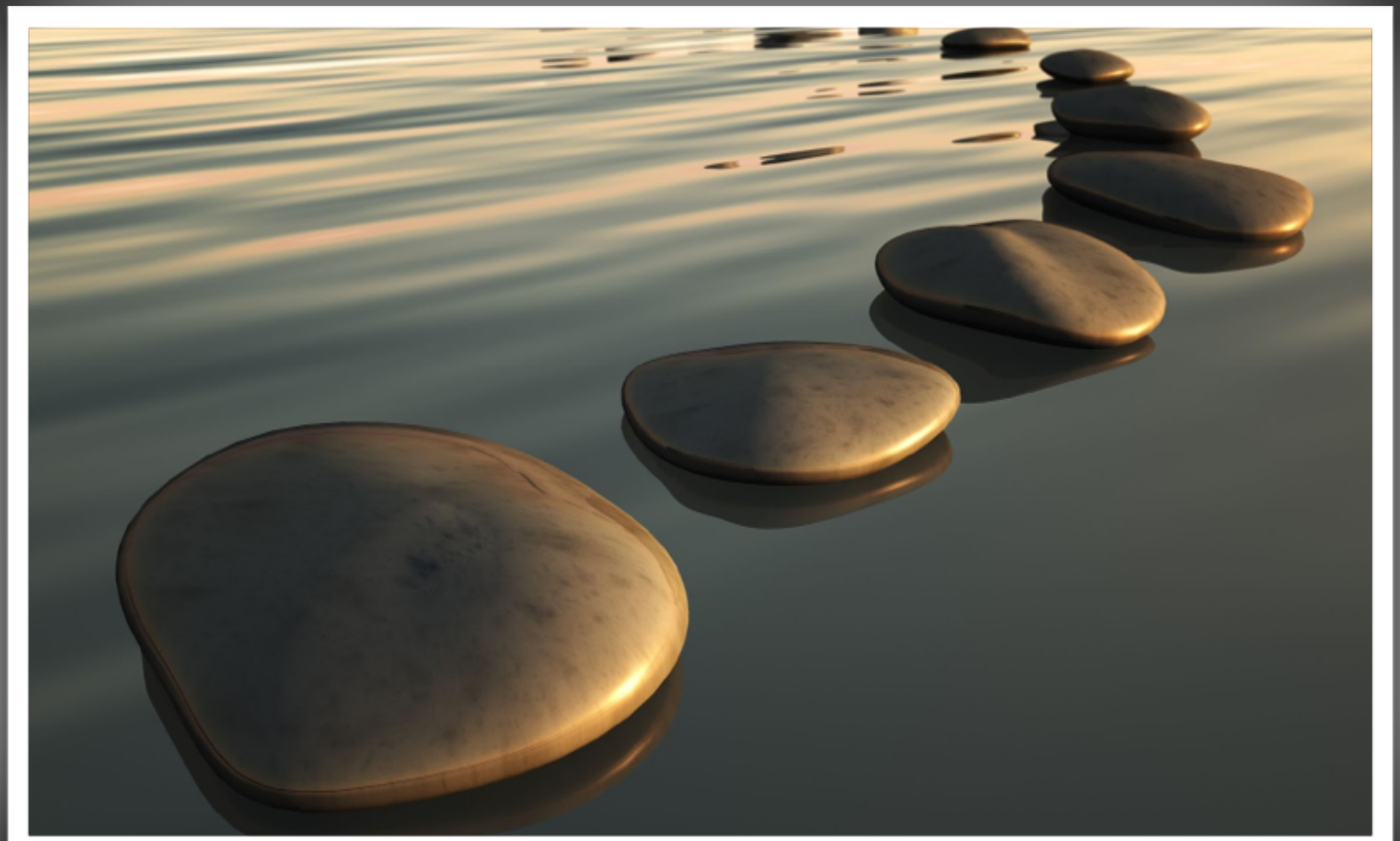
- Constraints on the concrete are given by the datatype invariant

```
dti : ArrayIntSet → Bool
dti this = this.size < this.N
```

Concrete Sets

```
// ABS: abs(values, size) = { values[i] | 0 ≤ i < size }
// DTI: size < N
class ArraySet<X> implements Set<X> {
  private X[] values;
  private int size;
  private final int N = 100;

  public ArraySet() {
    values = (X[]) new Object[N];
    size = 0;
  }

  @Override
  public void insert(X x) {
    values[size] = x;
    size = size + 1;
  }

  @Override
  public void delete(X x) {
    for (int i=0; i < size; i = i+1) {
      if (values[i].equals(x)) {
        values[i] = values[size-1];
        size = size - 1;
        break;
      }
    }
  }

  @Override
  public void clear() {
    size = 0;
  }

  @Override
  public boolean contains(X x) {
    boolean contains = false;
    for (X value : values) {
      if (value.equals(x)) {
        contains = true;
        break;
      }
    }
    return contains;
  }

  @Override
  public int size() {
    return size;
  }
}
```

Iterators



Iterators

- The Stack and Set interfaces are good abstractions for collecting values
- How would we go about saving all of the elements they contain into a file?
- We might think about pulling out all elements one-by-one, but there will surely be a lot of repeated code

Iterators

- More generally, we want to be able to somehow *iterate* over all the elements
- Java has an *Iterator* interface to do this

```
interface Iterator<E> {
  public boolean hasNext();
  public E next();
}
```

- This is well-documented in the javadocs

Q. How should we document the behaviour of an Iterator?

Iterators

- Using an iterator used to be a matter of using a while loop

```
Iterator<Integer> xs = …;
int sum = 0;
while (xs.hasNext()) {
  int x = xs.next();
  sum = sum + x;
}
```

- Java introduced notation that makes this all easier

```
Iterator<Integer> xs = …;
int sum = 0;
for (int x : xs) {
  sum = sum + x;
}
```

Iterators

- Using an iterator used to be a matter of using a while loop

```
Iterator<Integer> xs = …;
int sum = 0;
while (xs.hasNext()) {
  int x = xs.next();
  sum = sum + x;
}
```

- Java introduced notation that makes this all easier

```
Iterator<Integer> xs = …;
int sum = 0;
for (int x : xs) {
  sum = sum + x;
}
```

NB. Autoboxing at work, automatically casting Integer into int

Iterators

- An ADT can communicate its contents without exposing (too much) of its internal structure by using an *Iterator*

```
interface Iterator<E> {
  public boolean hasNext();
  public E next();
}
```

- In order to provide an iterator, the ADT must implement the *Iterable* interface

```
interface Iterable<E> {
  public Iterator<E> iterator ();
}
```

Iterators

- Let's see how the ArrayStack can implement Iterator
- First, we need to import some packages, which we add to the top of our file:

```
import java.lang.Iterable;
import java.util.Iterator;
```

- We also add *Iterable* to the list of implemented interfaces

```
class ArraySet<X> implements Set<X>, Iterable<X> {
  …
}
```

Iterators

- Next we add the methods that an *Iterable* implements

```
class ArraySet<X> implements Set<X>, Iterable<X> {
  @Override
  public Iterator<X> iterator() {
    ...
  }
}
```

Q. We need access to the **values** array, so how should our iterator be defined?

A. We could create a class and pass in the array as a parameter

A. We can use an **anonymous inner class**

Iterators

- The Iterator can be returned as a new object whose class we define in-place: it has access to everything in its context

```
class ArraySet<X> implements Set<X>, Iterable<X> {
  …
  @Override
  public Iterator<X> iterator() {
    return new Iterator<X>() {
      …
      public X next () {
        …


      }
      public boolean hasNext() {
        …
      }
    };
  }
}
```

Iterators

- The Iterator can be returned as a new object whose class we define in-place: it has access to everything in its context

```
class ArraySet<X> implements Set<X>, Iterable<X> {
  …
  @Override
  public Iterator<X> iterator() {
    return new Iterator<X>() {
      private int index = 0;
      public X next () {
        …


      }
      public boolean hasNext() {
        …
      }
    };
  }
}
```

Iterators

- The Iterator can be returned as a new object whose class we define in-place: it has access to everything in its context

```
class ArraySet<X> implements Set<X>, Iterable<X> {
  …
  @Override
  public Iterator<X> iterator() {
    return new Iterator<X>() {
      private int index = 0;
      public X next () {
        …


      }
      public boolean hasNext() {
        return (index < size);
      }
    };
  }
}
```

Iterators

- The Iterator can be returned as a new object whose class we define in-place: it has access to everything in its context

```
class ArraySet<X> implements Set<X>, Iterable<X> {
  …
  @Override
  public Iterator<X> iterator() {
    return new Iterator<X>() {
      private int index = 0;
      public X next () {
        X x = values[index];
        index = index + 1;
        return x;
      }
      public boolean hasNext() {
        return (index < size);
      }
    };
  }
}
```

Iterators

- The Iterator can be returned as a new object whose class we define in-place: it has access to everything in its context

```
class ArraySet<X> implements Set<X>, Iterable<X> {
  …
  @Override
  public Iterator<X> iterator() {
    return new Iterator<X>() {
      private int index = 0;
      public X next () {
        X x = values[index];
        index = index + 1;
        return x;
      }
      public boolean hasNext() {
        return (index < size);
      }
    };
  }
}
```

Q. What happens if the Set changes during an iteration?

Iterators

- The Iterator can be returned as a new object whose class we define in-place: it has access to everything in its context

```
class ArraySet<X> implements Set<X>, Iterable<X> {
  …
  @Override
  public Iterator<X> iterator() {
    return new Iterator<X>() {
      private int index = 0;
      public X next () {
        X x = values[index];
        index = index + 1;
        return x;
      }
      public boolean hasNext() {
        return (index < size);
      }
    };
  }
}
```

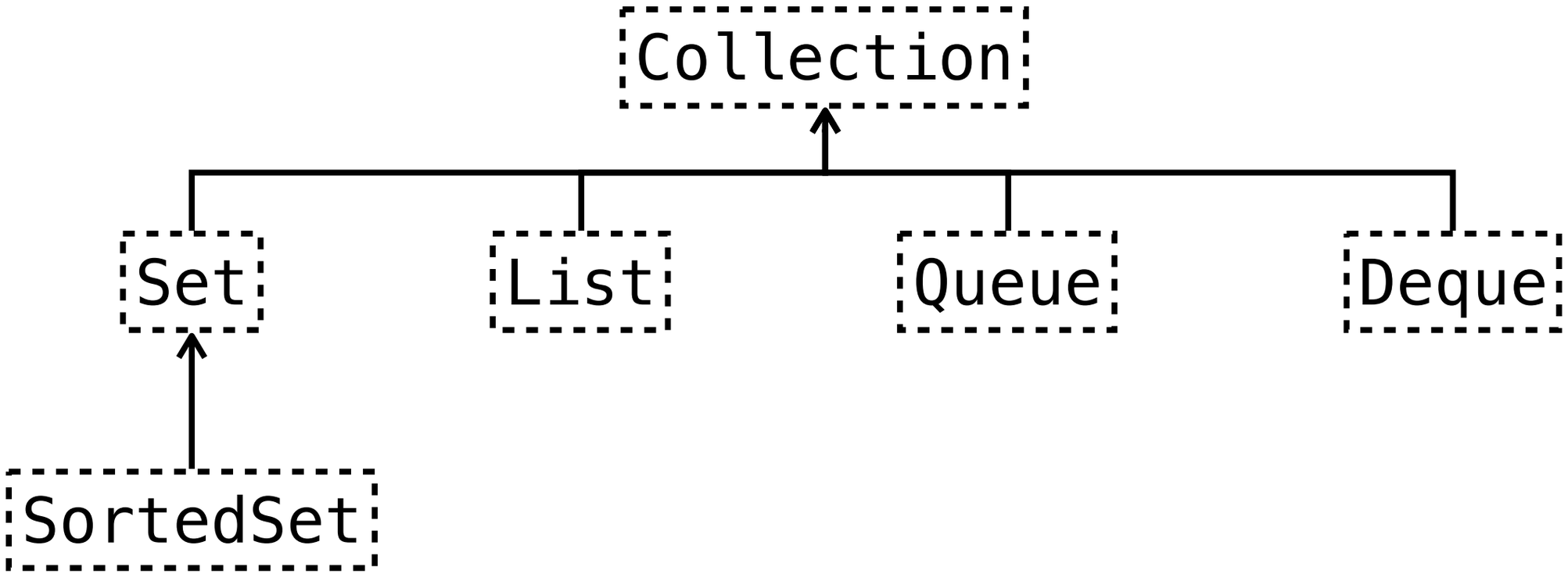
Q. What happens if the Set changes during an iteration?

A. Bad things. This isn't allowed

Collections

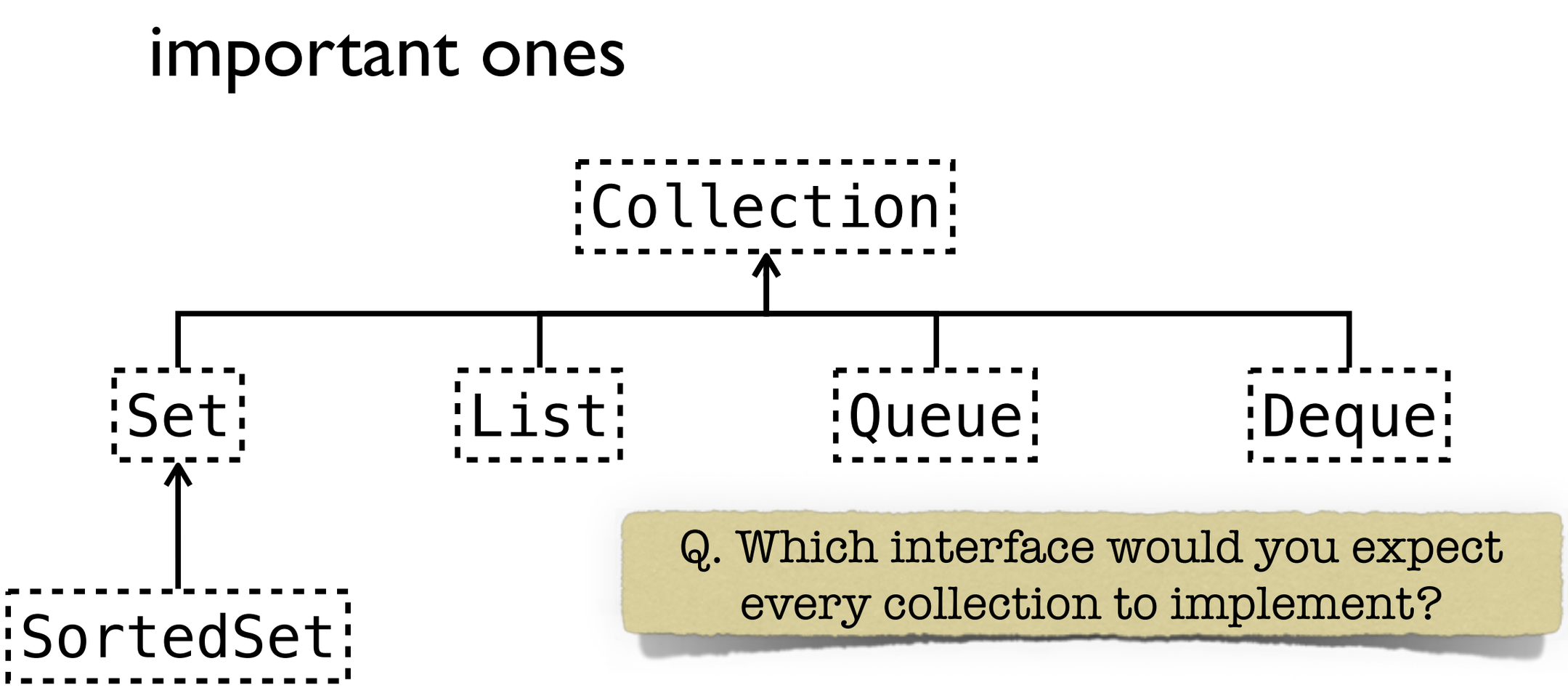
Collections

- Java comes with a *huge* library of classes
- The *Collection* interface is one of the most important ones



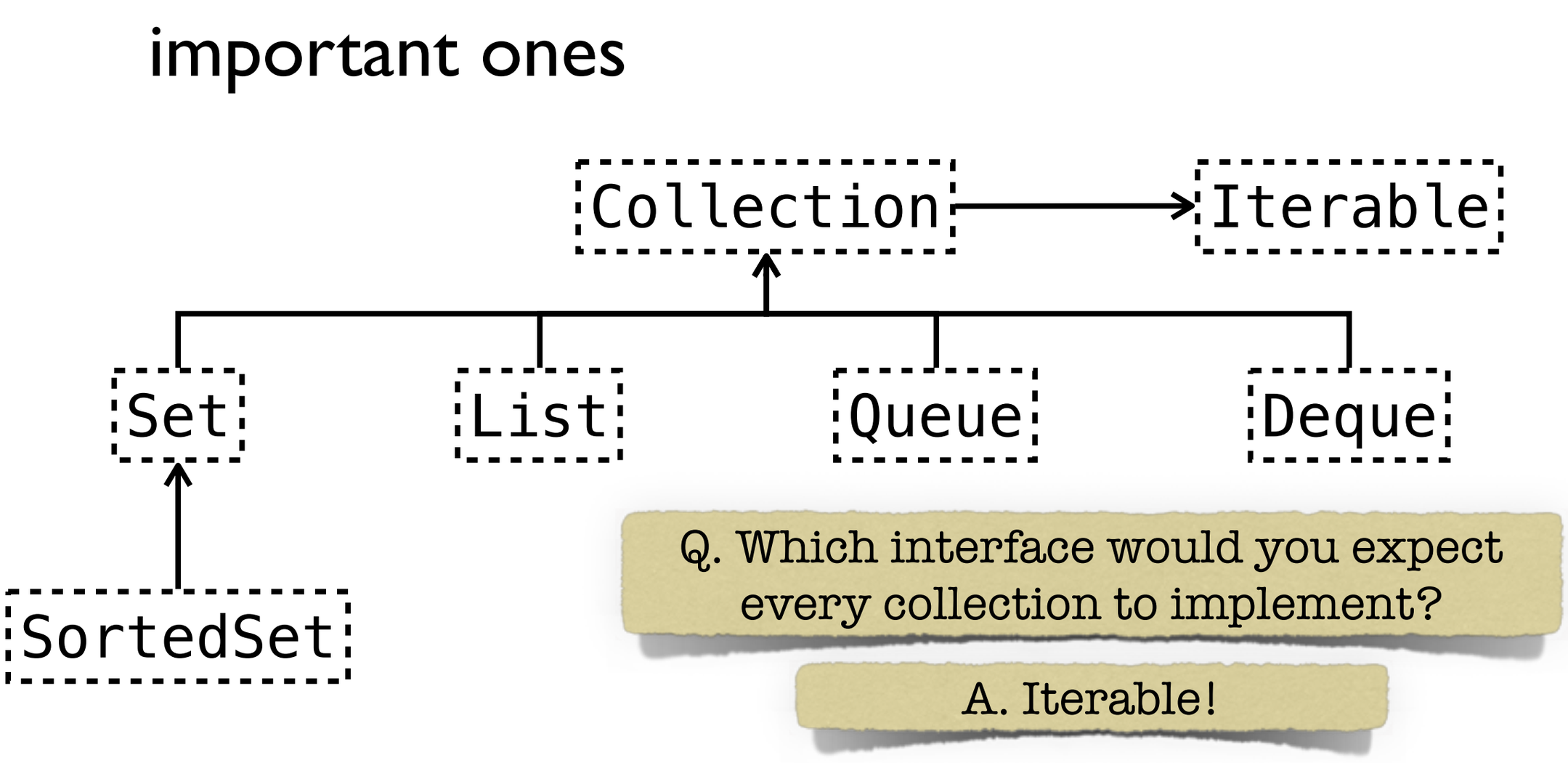
Collections

- Java comes with a *huge* library of classes
- The *Collection* interface is one of the most important ones



Collections

- Java comes with a *huge* library of classes
- The *Collection* interface is one of the most important ones



Collections

- How would we sort a Collection?
- Not all collections are inherently sortable: a Set isn't naturally sorted
- However, a List *can* be sorted
- Moreover, we can make an ArrayList from any Collection

Collections

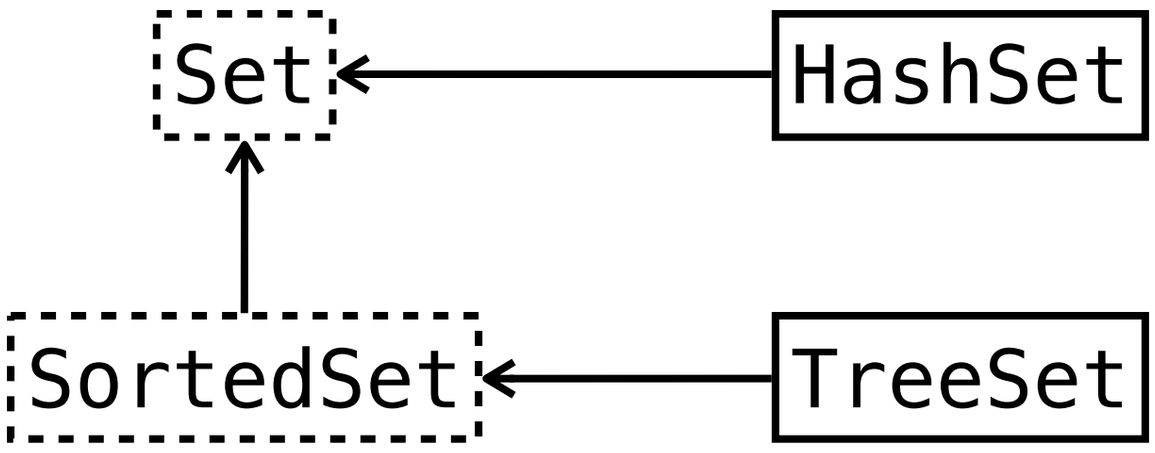
- To make an ArrayList from a Collection, we simply use the constructor

```
List<X> list = new ArrayList<X>(c);
Collections.sort(list);
return list;
```

- However, you might want to ask yourself if you want to use a structure that's always sorted, and use an *Iterator* instead

Collections

- The Set interface has two main implementations



- A HashSet is much faster than a TreeSet, but isn't sorted: iterating through it will produce elements in random order