

CSCI 136:
Data Structures
and
Advanced Programming

Lecture 19-2
Ordered Structures

Instructor: Dan Barowy

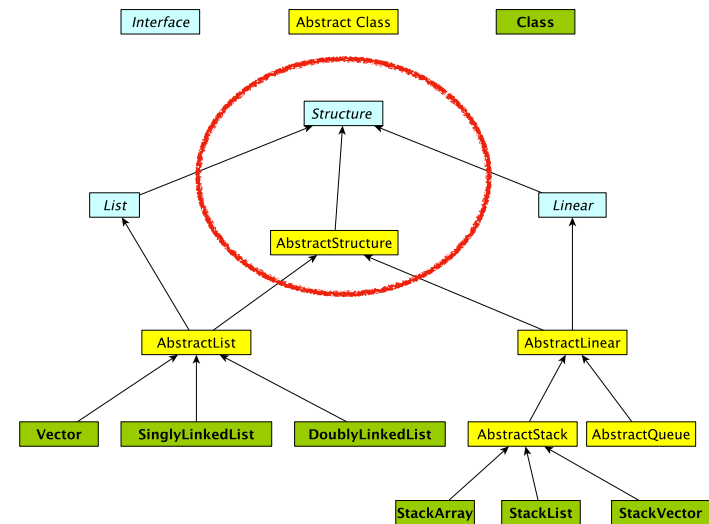
Williams

Outline

1. Ordered structures
2. Binary search

Ordered structures

structure5 Stack implementations



structure in structure5

A **structure** is an interface for a "traversable" collection of objects. In other words, it represents a class that **contains** some number of elements, and those elements can be **iterated**, **added**, and **removed**. **Membership** and **size** can also be checked.

Most of the data structures we discuss in this class implement **structure**.

structure in structure5

```
public interface Structure<E> extends Iterable<E>
{
    public int size();
    public boolean isEmpty();
    public void clear();
    public boolean contains(E value);
    public void add(E value);
    public E remove(E value);
    public java.util.Enumeration elements();
    public Iterator<E> iterator();
    public Collection<E> values();
}
```

Question for you

Why is a **structure** interface a **good idea**? What **benefit** do we get from having it?

One reason

Suppose we write a **method** that takes a **structure**. We could give it an instance of **any data structure** that implements the **structure** interface.

E.g., we could **iterate** over the elements and print them because **all structures** have the **iterator()** method.

What about **order**?

Does the `structure` interface require that elements be **ordered**?

`structure` in `structure5`

```
public interface Structure<E> extends Iterable<E>
{
    public int size();
    public boolean isEmpty();
    public void clear();
    public boolean contains(E value);
    public void add(E value);
    public E remove(E value);
    public java.util.Enumeration elements();
    public Iterator<E> iterator();
    public Collection<E> values();
}
```

What about **order**?

Does the `structure` interface require that elements be **ordered**?

No.

What about **order**?

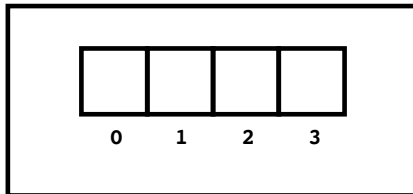
Is it possible to create a data structure to which you **add elements one at a time in any order**, but which **keeps elements in sorted order**?

Yes.

Should be possible

Simple inductive proof.

Start with an **empty data structure**.



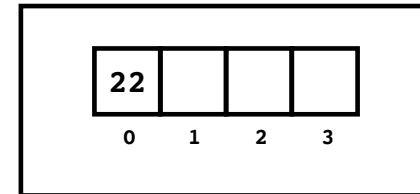
FooStructure

Is FooStructure ordered? **Yes.**

Should be possible

Simple inductive proof.

Add an element (any element). **Insert it in order.**



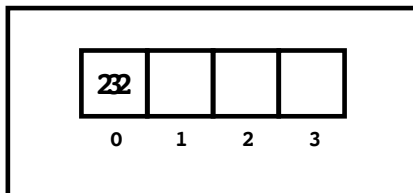
FooStructure

Is FooStructure ordered? **Yes.**

Should be possible

Simple inductive proof.

Add another element (any element). **Insert it in order.**



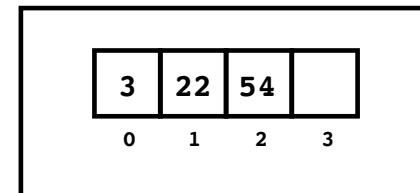
FooStructure

Is FooStructure ordered? **Yes.**

Should be possible

Simple inductive proof.

Add another element (any element). **Insert it in order.**



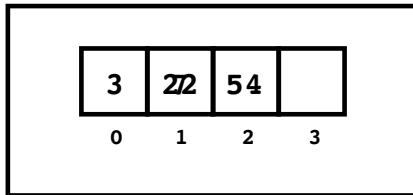
FooStructure

Is FooStructure ordered? **Yes.**

Should be possible

Simple inductive proof.

Add another element (any element). **Insert it in order.**



FooStructure

Is FooStructure ordered? **Yes.**

What about **order**?

Is order a property that **can be enforced** using interfaces?

No. Order is a **data-dependent property**, so there's no way to check whether something is ordered until runtime.

OrderedStructure

Nevertheless, we can **signal our intent** with an interface.

How would we write an `OrderedStructure` interface?

Do its elements need to have **any special property**? (i.e., how would we **compare** them?)

Let's try to write this.

(code)

OrderedVector

Let's try implementing an `OrderedVector`.

(code)

OrderedVector

How do we figure out where `add` should insert?

Binary search to the rescue.

Binary search

100	101	322	365	423	478	499	504
0	1	2	3	4	5	6	7

Want to know **whether** the array contains the value **322**, and if so, what its **index** is.

Binary search is a **divide-and-conquer** algorithm that solves this problem.

Binary search is **fast**: in the **worst case**, it returns an answer in **$O(\log_2 n)$** steps.

Binary search

100	101	322	365	423	478	499	504
0	1	2	3	4	5	6	7

Important precondition: array must be **sorted**.

Binary search


Looking for the value **322**.

100	101	322	365	423	478	499	504
0	1	2	3	4	5	6	7

Binary search

Looking for the value **322**.



100	101	322	365	423	478	499	504
0	1	2	3	4	5	6	7



Binary search

Looking for the value **322**.




100	101	322	365	423	478	499	504
0	1	2	3	4	5	6	7



Binary search

Looking for the value **322**.




100	101	322	365	423	478	499	504
0	1	2	3	4	5	6	7



Binary search

Looking for the value **322**.

100	101	322	365	423	478	499	504
0	1	2	3	4	5	6	7



322 = 365? **no**

322 < 365? **yes**

Binary search

Looking for the value **322**.

100	101	322	365	423	478	499	504
0	1	2	3	4	5	6	7

↑ ↑ ↑

Binary search

Looking for the value **322**.

100	101	322	365	423	478	499	504
0	1	2	3	4	5	6	7

↑ ↑ ↑

322 = 101? **no**

322 < 101? **no**

322 > 101? **yes**

Binary search

Looking for the value **322**.

100	101	322	365	423	478	499	504
0	1	2	3	4	5	6	7

 ↑ ↑ ↑

Binary search

Looking for the value **322**.

100	101	322	365	423	478	499	504
0	1	2	3	4	5	6	7

 ↑ ↑ ↑

322 = 322? **yes**

return 2

Recap & Next Class

Today we learned:

Ordered structures

Next class:

Trees