# CSCI 136:
# Data Structures
# and
# Advanced Programming

Lecture 19

Ordered Structures

Instructor: Dan Barowy

Williams

## Outline

1. Mid-semester eval.
2. Resubmission procedure
3. Ordered structures
4. Infix to postfix algorithm

## Mid-semester evaluation

## Resubmission procedure

## Resubmission procedure

Remember: the goal of this course is mastery.

## Resubmission procedure

Allows you to earn **up to 50% of the lost points**.

E.g., **if you got a 50%** on the midterm, **you can get a 75%** on resubmission.

Midterm is 20% of your final grade. **This is worth doing!**

## Resubmission procedure

1. You have **two weeks** from tomorrow (your exam will be in my box by tomorrow).
2. Resubmission **must include both** the **original work** and the **new submission**.
3. Must be accompanied by an **explanation document**, written in plain English.

## Resubmission procedure

Explanation document **must identify**:

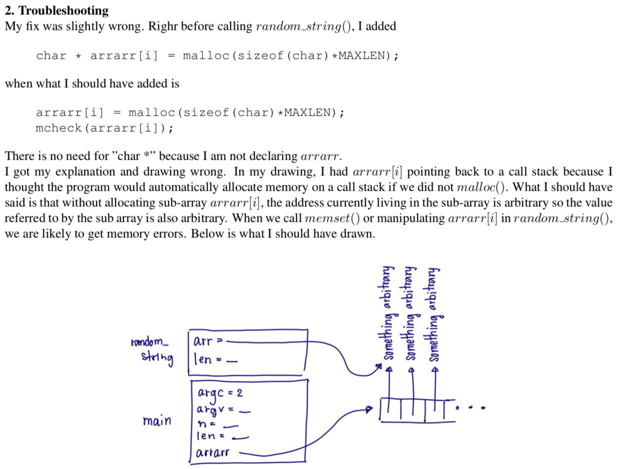
1. **What** the mistake is.
2. **How** you fixed the mistake.
3. **Why** the new version is correct.

# Resubmission procedure

Please submit this **on paper**
(put it in my box in the CS department).

# Resubmission procedure

Sample:



**2. Troubleshooting**
My fix was slightly wrong. Righr before calling *random_string*(), I added

```
char * arrarr[i] = malloc(sizeof(char)*MAXLEN);
```

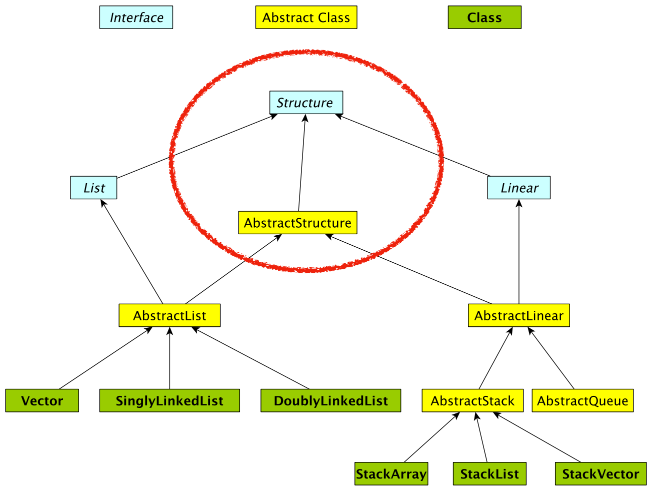
when what I should have added is

```
arrarr[i] = malloc(sizeof(char)*MAXLEN);
mcheck(arrarr[i]);
```

There is no need for ”char *” because I am not declaring *arrarr*.
I got my explanation and drawing wrong. In my drawing, I had *arrarr*[*i*] pointing back to a call stack because I thought the program would automatically allocate memory on a call stack if we did not *malloc*(). What I should have said is that without allocating sub-array *arrarr*[*i*], the address currently living in the sub-array is arbitrary so the value referred to by the sub array is also arbitrary. When we call *memset*() or manipulating *arrarr*[*i*] in *random_string*(), we are likely to get memory errors. Below is what I should have drawn.

# 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.

Is order a property that **could 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`

Nonetheless, 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.

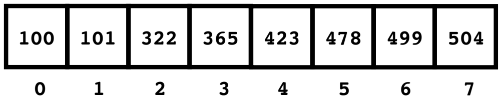
## Binary search

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

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

## Binary search

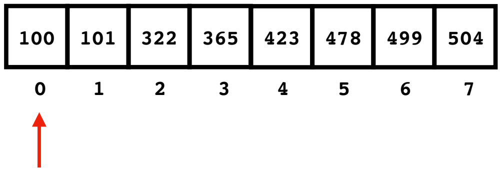
Looking for the value **322**.



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

## Binary search

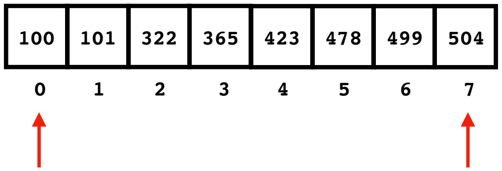
Looking for the value **322**.



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

## Binary search

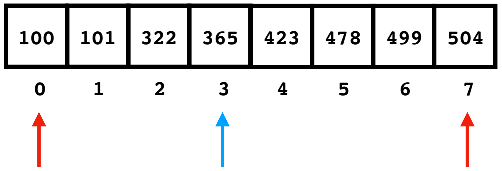
Looking for the value **322**.



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

## Binary search

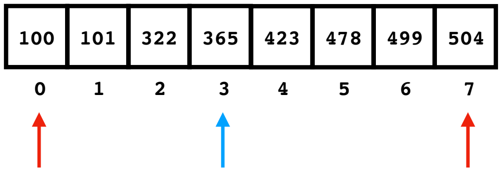
Looking for the value **322**.



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

## Binary search

Looking for the value **322**.



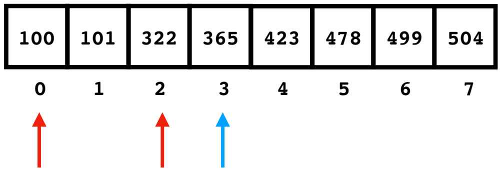
| 100 | 101 | 322 | 365 | 423 | 478 | 499 | 504 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

**322** = 365? **no**

**322** < 365? **yes**

## Binary search

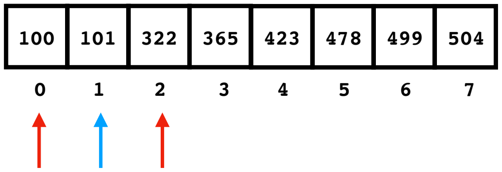
Looking for the value **322**.



| 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 |

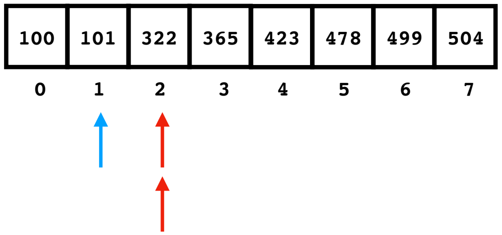
**322** = 101? **no**

**322** < 101? **no**

**322** > 101? **yes**

## Binary search

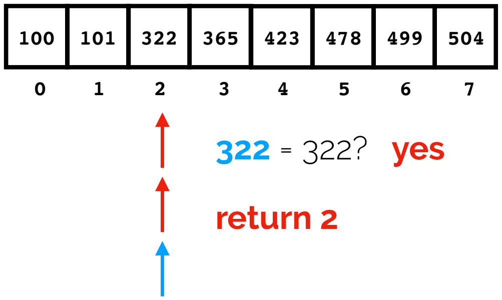
Looking for the value **322**.



| 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:

More about ordered structures,

Shunting yard,

Trees