CSCI 136:
Data Structures
and
Advanced Programming

Lecture 19

Search

Instructor: Dan Barowy

Williams

Topics

- Iterators
- Binary search
- How to resubmit work in this course

Your to-dos

1. Read **before Mon**: Bailey, Ch 11.
2. Lab 6 (partner lab), **due Tuesday 4/12 by 10pm**.

Announcements

Colloquium on Friday.

Friday, April 8 @ 2:35pm

Wege Hall – TCL 123

Perception and Context in Data Visualization

Jordan Crouser, Smith College

Visual analytics is the science of combining interactive visual interfaces and information visualization techniques with automatic algorithms to support analytical reasoning through human-computer interaction. People use visual analytics tools and techniques to synthesize information and derive insight from massive, dynamic, ambiguous, and often conflicting data… and we exploit all kinds of perceptual tricks to do it! In this talk, we'll explore concepts in decision-making, human perception, and color theory as they apply to data-driven communication. Whether you're an aspiring data scientist or you're just curious about the mechanics of how data visualization works under the hood, stop by and take your pre-attentive processing for a spin.

Announcements

- **ACM TechTalk: "Visual Data Analysis: Why? When? How?"**
  Organized by Prof. Kelly Shaw and CoSSAC.
  Wednesday, April 13 from 7-7:45pm in TBL 211.
  "Extra special snacks" provided by CoSSAC afterward in the Eco Cafe.

Announcements

Please **consider being a TA** next semester (especially for this class!)

Applications **due Friday, April 22**.

https://csci.williams.edu/tatutor-application/

Iterators

What do the following have in common?

```
double[] a
// … initialize a …
double sum = 0.0;
for (int i = 0; i < a.length; i++) {
  sum += a[i];
}
```

```
List<Double> ls = new SinglyLink
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i
  sum += ls.get(i);
}
```

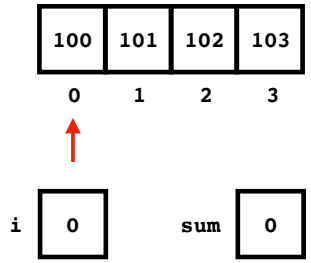
```
Stack<Double> s = new StackVect
// … initialize s …
double sum = 0.0;
while (!s.isEmpty()) {
  sum += s.pop();
}
```

Iteration

Iteration is the **repetition of a process** in order to generate a (possibly unbounded) **sequence of outcomes**. Each repetition of the process is a single iteration, and the outcome of each iteration is then the starting point of the next iteration.

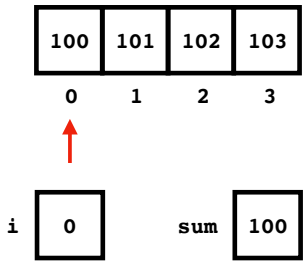
Each program iterates

```
double[] a
// … initialize a …
double sum = 0.0;
for (int i = 0; i < a.length; i++) {
  sum += a[i];
}
```



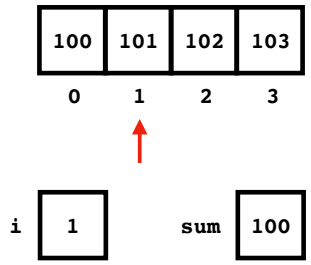
Each program iterates

```
double[] a
// … initialize a …
double sum = 0.0;
for (int i = 0; i < a.length; i++) {
  sum += a[i];
}
```



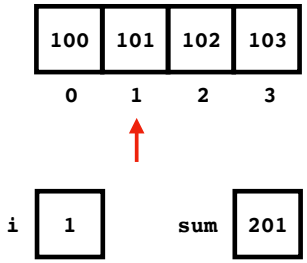
Each program iterates

```
double[] a
// … initialize a …
double sum = 0.0;
for (int i = 0; i < a.length; i++) {
  sum += a[i];
}
```



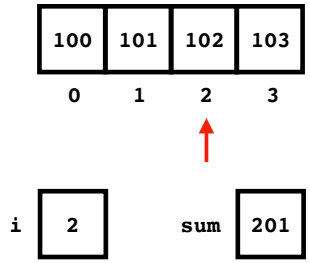
Each program iterates

```
double[] a
// … initialize a …
double sum = 0.0;
for (int i = 0; i < a.length; i++) {
  sum += a[i];
}
```



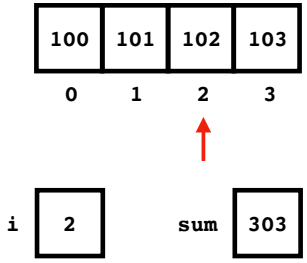
Each program iterates

```
double[] a
// … initialize a …
double sum = 0.0;
for (int i = 0; i < a.length; i++) {
  sum += a[i];
}
```



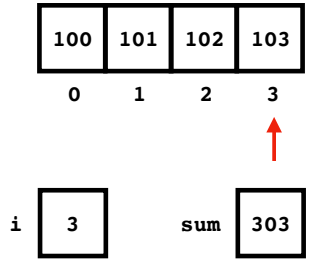
Each program iterates

```
double[] a
// … initialize a …
double sum = 0.0;
for (int i = 0; i < a.length; i++) {
  sum += a[i];
}
```



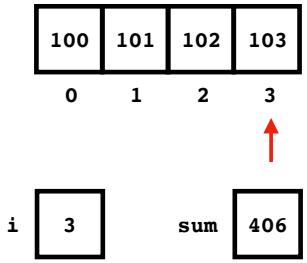
Each program iterates

```
double[] a
// … initialize a …
double sum = 0.0;
for (int i = 0; i < a.length; i++) {
  sum += a[i];
}
```



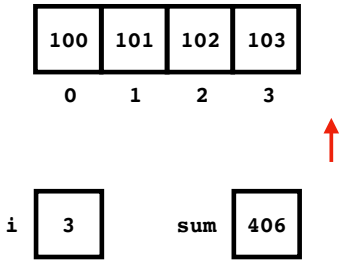
Each program iterates

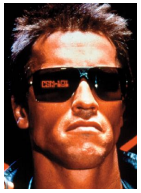
```
double[] a
// … initialize a …
double sum = 0.0;
for (int i = 0; i < a.length; i++) {
  sum += a[i];
}
```



Each program iterates

```
double[] a
// … initialize a …
double sum = 0.0;
for (int i = 0; i < a.length; i++) {
  sum += a[i];
}
```

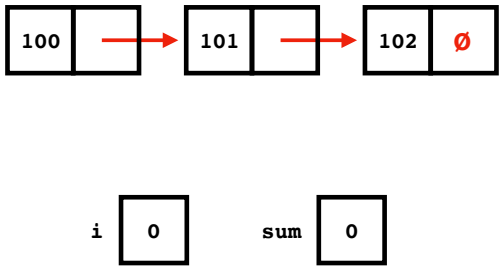




"Iteration is terminated!"

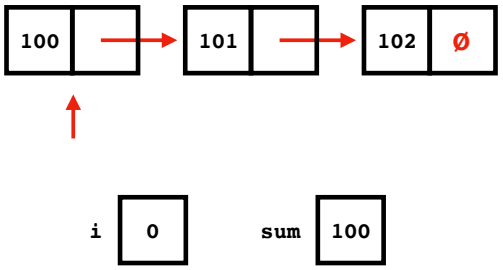
Each program iterates

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



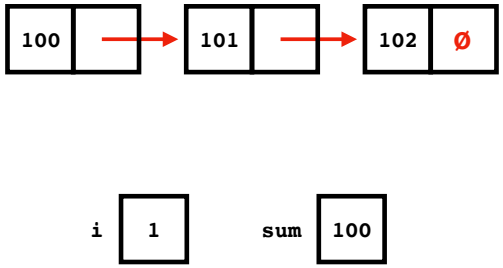
Each program iterates

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



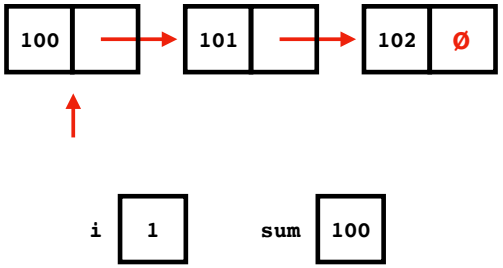
Each program iterates

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



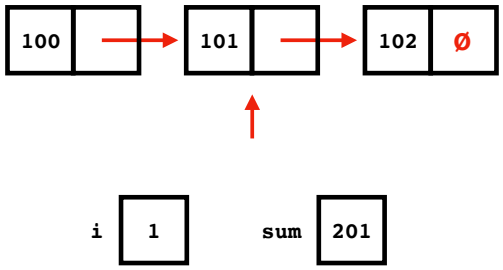
Each program iterates

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



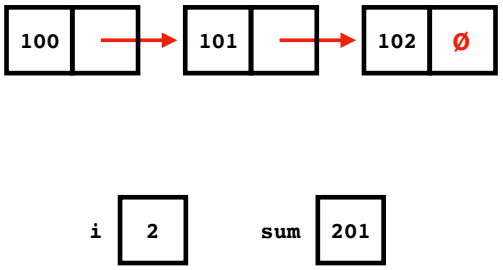
Each program iterates

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



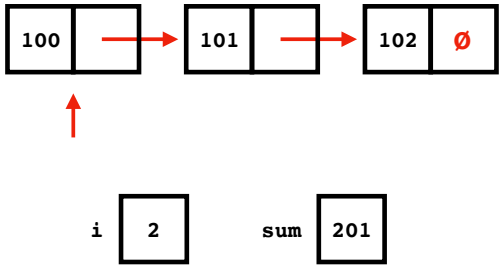
Each program iterates

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



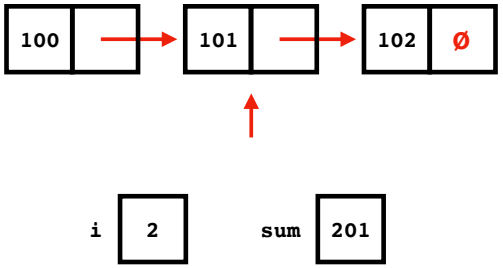
Each program iterates

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



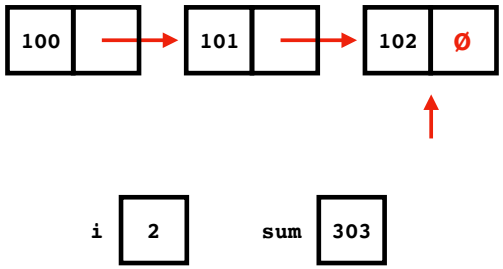
Each program iterates

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



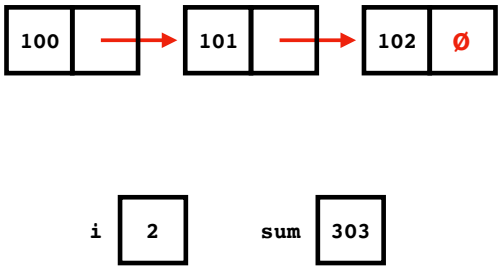
Each program iterates

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



Each program iterates

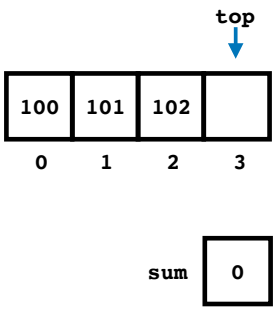
```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



"Iteration is terminated!"

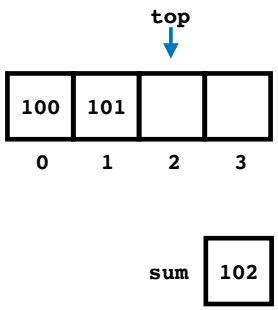
Each program iterates

```
Stack<Double> s = new StackVector<>();
// … initialize s …
double sum = 0.0;
while (!s.isEmpty()) {
  sum += s.pop();
}
```



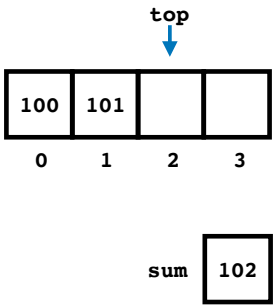
Each program iterates

```
Stack<Double> s = new StackVector<>();
// … initialize s …
double sum = 0.0;
while (!s.isEmpty()) {
  sum += s.pop();
}
```



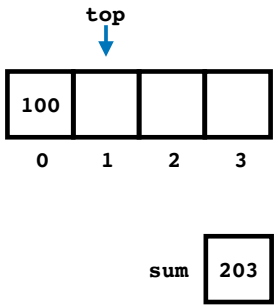
Each program iterates

```
Stack<Double> s = new StackVector<>();
// … initialize s …
double sum = 0.0;
while (!s.isEmpty()) {
  sum += s.pop();
}
```



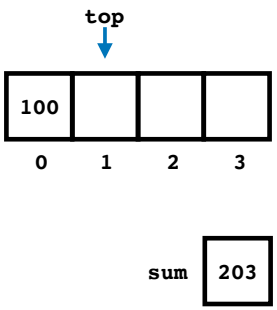
Each program iterates

```
Stack<Double> s = new StackVector<>();
// … initialize s …
double sum = 0.0;
while (!s.isEmpty()) {
  sum += s.pop();
}
```



Each program iterates

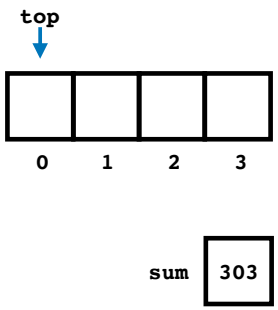
```
Stack<Double> s = new StackVector<>();
// … initialize s …
double sum = 0.0;
while (!s.isEmpty()) {
  sum += s.pop();
}
```



sum 203

Each program iterates

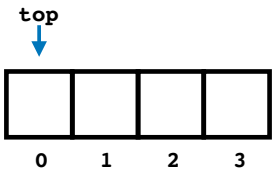
```
Stack<Double> s = new StackVector<>();
// … initialize s …
double sum = 0.0;
while (!s.isEmpty()) {
  sum += s.pop();
}
```



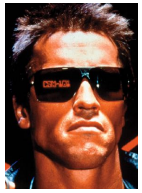
sum 303

Each program iterates

```
Stack<Double> s = new StackVector<>();
// … initialize s …
double sum = 0.0;
while (!s.isEmpty()) {
  sum += s.pop();
}
```



sum 303



"Iteration is terminated!"

Essentially the same algorithm!

```
double[] a
// … initialize a …
double sum = 0.0;
for (int i = 0; i < a.length; i++) {
  sum += a[i];
}
```

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```

```
Stack<Double> s = new StackVector<>();
// … initialize s …
double sum = 0.0;
while (!s.isEmpty()) {
  sum += s.pop();
}
```

But the code looks different.

Problems

- **Different data structures** yield **different code for same algorithm**.
- **Data hiding** potentially causes **efficiency problems**.
- **Inspecting** data structure "from the outside" can **change the state** of a data structure (e.g., `pop()`'ing a `Stack`).

What if I told you that you could solve

all of these problems with **abstraction**?

Iteration abstraction to the rescue.

```
double[] a
// … initialize a …
double sum = 0.0;
for (double d : a) {
  sum += d;
}
```

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (double d : ls) {
  sum += d;
}
```

```
Stack<Double> s = new StackVector<>();
// … initialize s …
double sum = 0.0;
for (double d : s) {
  sum += d;
}
```

Brought to you by **Iterators**.

Iterators are a really good idea.

- Invented by Barbara Liskov in 1974.
- Incidentally, **abstract data types** were also invented by Barbara Liskov in 1974.
- Both debuted in the influential PL called **CLU**.
- Barbara won the **Turing Award in 2008** for this work and more.

How does “for each” work?

```
for (int num : nums) { … }
```

All of these data structures must implement `Iterable<T>`

structure5
Interface Stack<E>

All Superinterfaces:
java.lang.Iterable<E>, Linear<E>, Structure<E>

All Known Implementing Classes:
AbstractStack, StackArray, StackList, StackVector

structure5
Interface List<E>

All Superinterfaces:
java.lang.Iterable<E>, Structure<E>

All Known Implementing Classes:
AbstractList, CircularList, DoublyLinkedList, SinglyLinkedList, Vector

(array is a special case)

What is an `Iterable<T>`?

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

It’s a class that returns an `Iterator<T>`.

What’s an `Iterator<T>`???

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

It’s an object that lets you **iterate through a data structure**.

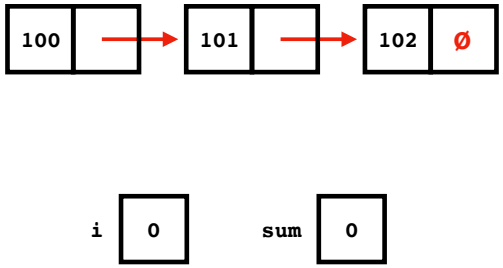
Importantly, `Iterator`s are **stateful**.

Why does statefulness matter? It can **save work**.

Let’s look at `SinglyLinkedList<T>`

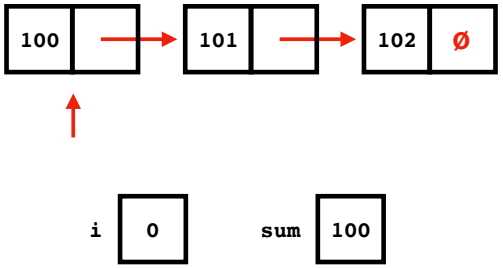
Naive iteration makes O(n) operation O(n^2)!

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



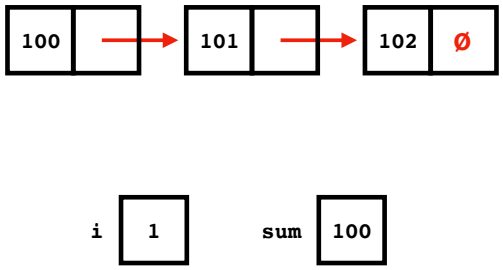
Naive iteration makes O(n) operation O(n^2)!

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



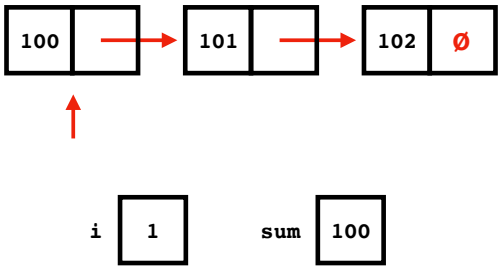
Naive iteration makes O(n) operation O(n^2)!

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



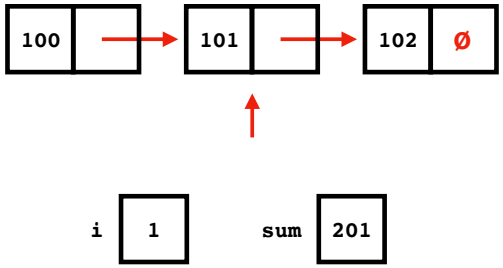
Naive iteration makes O(n) operation O(n^2)!

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



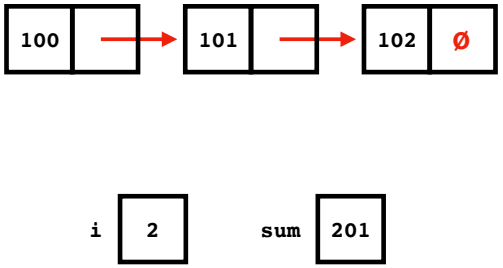
Naive iteration makes O(n) operation O(n^2)!

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



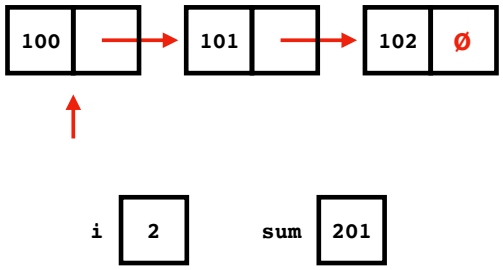
Naive iteration makes O(n) operation O(n^2)!

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



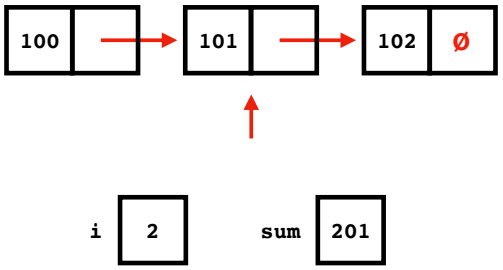
Naive iteration makes O(n) operation O(n^2)!

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



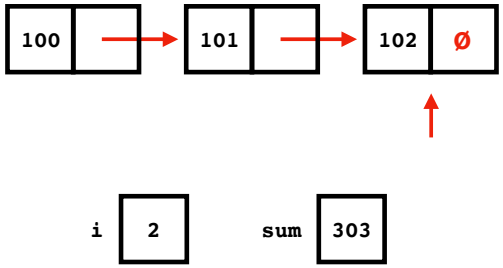
Naive iteration makes O(n) operation O(n^2)!

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



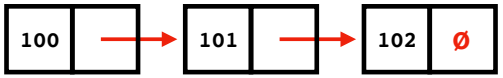
Naive iteration makes O(n) operation O(n²)!

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



Naive iteration makes O(n) operation O(n²)!

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (int i = 0; i < ls.size(); i++) {
  sum += ls.get(i);
}
```



"Iteration is terminated!"

How does `for` use an `Iterator<T>`?

The following code

```
List<Integer> ls = new SinglyLinkedList<>();
// …
for (int i : ls) {
  // … work …
}
```

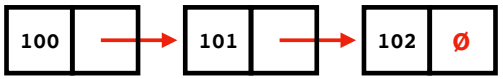
is the moral equvalent to

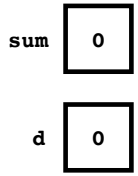
```
List<Integer> ls = new SinglyLinkedList<>();
// …
for (Iterator<Integer> i = ls.iterator(); i.hasNext(); ) {
  int n = i.next()
  // … work …
}
```

1. Get `Iterator<T>`
2. Get next element.
3. If there is a next element, go to 2.

Example.

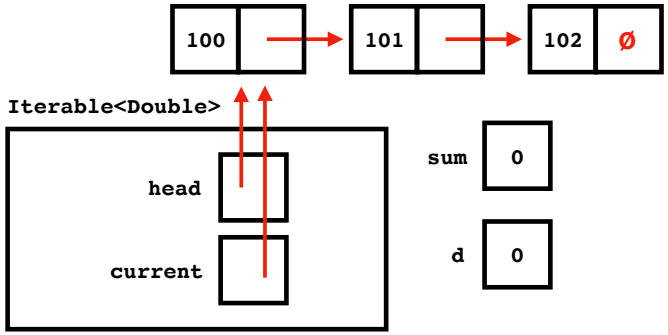
```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (double d : ls) {
  sum += d;
}
```





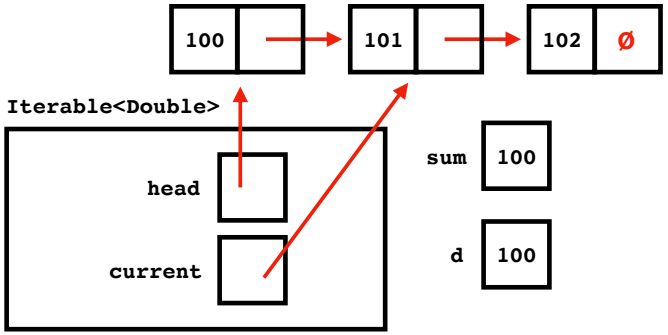
Example.

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (double d : ls) {
  sum += d;
}
```



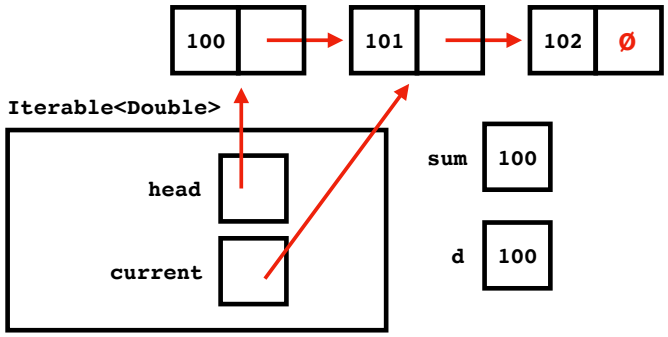
Example.

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (double d : ls) {
  sum += d;
}
```



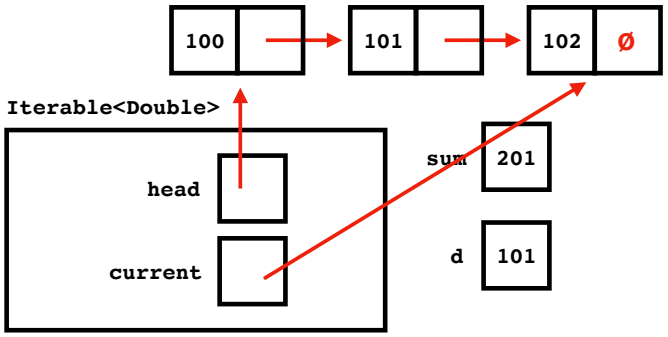
Example.

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (double d : ls) {
  sum += d;
}
```



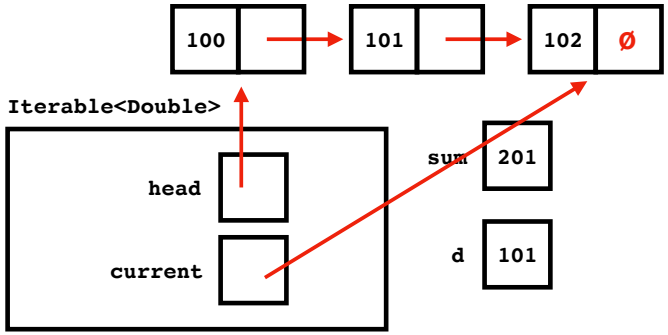
Example.

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (double d : ls) {
  sum += d;
}
```



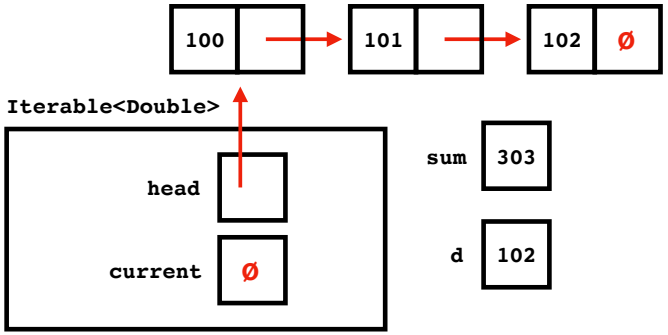
Example.

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (double d : ls) {
  sum += d;
}
```



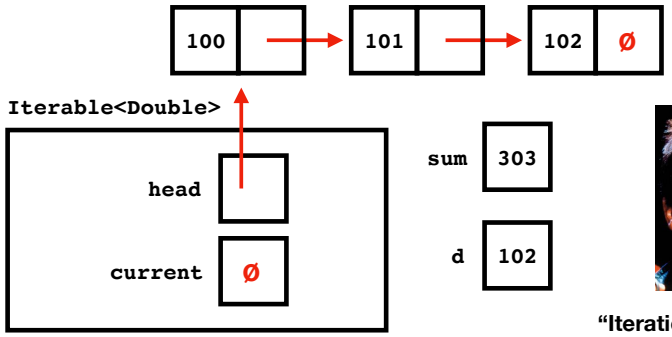
Example.

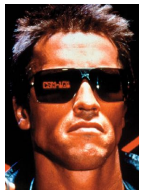
```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (double d : ls) {
  sum += d;
}
```



Example.

```
List<Double> ls = new SinglyLinkedList<>();
// … initialize ls …
double sum = 0.0;
for (double d : ls) {
  sum += d;
}
```





"Iteration is terminated!"

Efficient searching: binary search

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

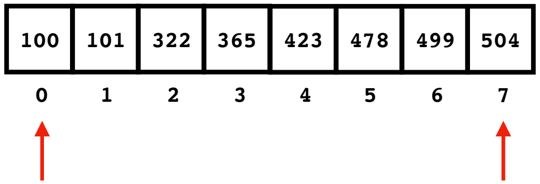
Binary search

Looking for the value **322**.

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

Binary search

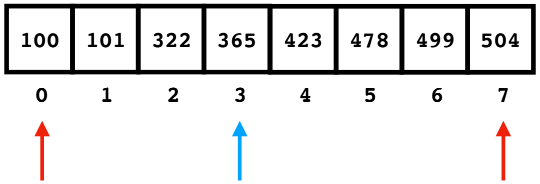
Looking for the value **322**.



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

Binary search

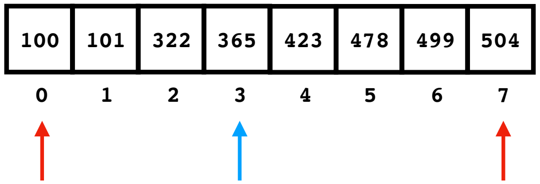
Looking for the value **322**.



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

Binary search

Looking for the value **322**.



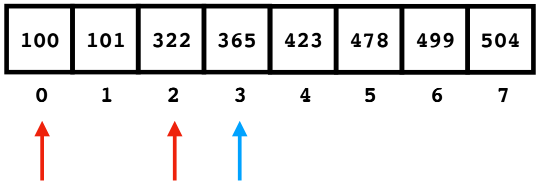
100	101	322	365	423	478	499	504
0	1	2	3	4	5	6	7

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

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 25% of your final grade.
This is worth doing!

Resubmission procedure

1. You have **until the end of reading period**.
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

Resubmit code **electronically**
(i.e., using git).

Resubmit exam **on paper**
(i.e., hand it to me).

Resubmission procedure

Sample:

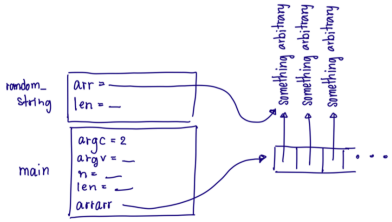
2. Troubleshooting
My fix was slightly wrong. Righr before calling $random\_string()$, I added

```
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.



Recap & Next Class

Today:

Iteration

Binary search

Resubmission procedure

Next class:

Ordered structures