

CSCI 136:  
Data Structures  
and  
Advanced Programming

Lecture 19

Iteration & Search

Instructor: Dan Barowy

**Williams**

## Topics

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

## Your to-dos

1. Better not come to class **on Friday!**
2. Read **before Mon**: Bailey, Ch 12-12.5.
3. Lab 7 (partner lab), **due Tuesday 11/1 by 10pm.**

## Announcements

- CS Colloquium, Fri @ 2:35 in Wege Auditorium  
**Pre-registration info session**  
**+ cookies**

## Announcements

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

Applications **due Friday, Oct 28.**

<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 SinglyLinke
// ... 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();
}
```

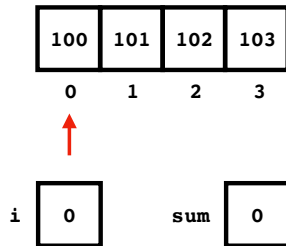


## Iteration

**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.

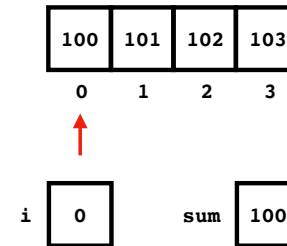
## Each program iterates

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



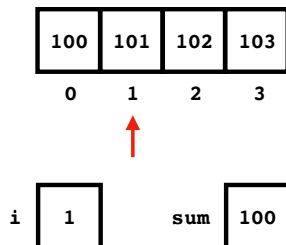
## Each program iterates

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



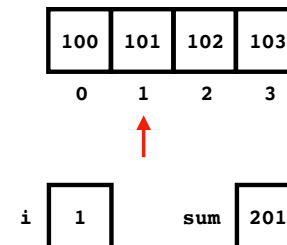
## Each program iterates

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



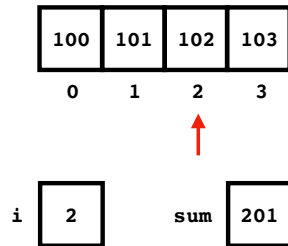
## Each program iterates

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



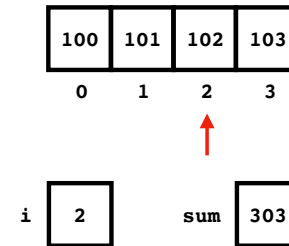
## Each program iterates

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



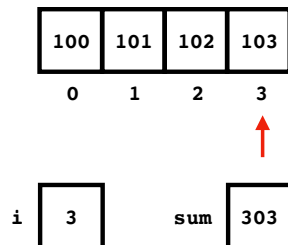
## Each program iterates

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



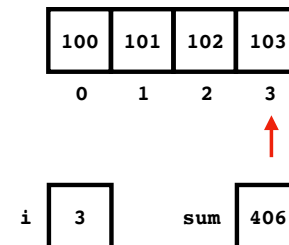
## Each program iterates

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



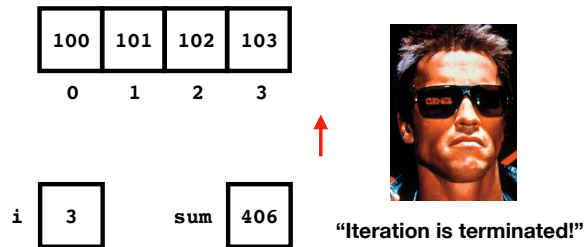
## Each program iterates

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



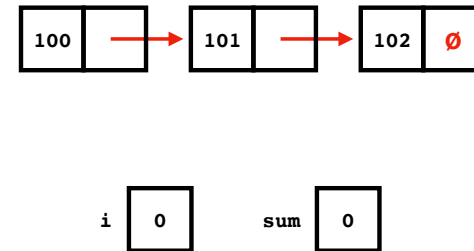
## Each program iterates

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



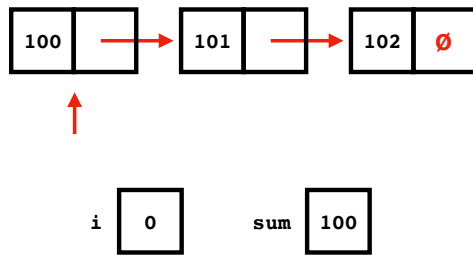
## Each program iterates

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



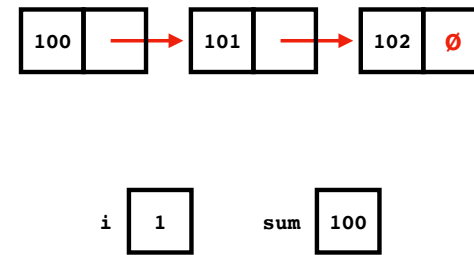
## Each program iterates

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



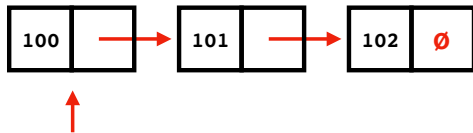
## Each program iterates

```
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

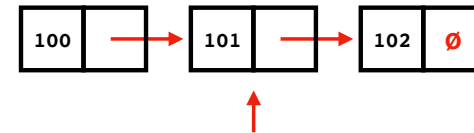
```
List<Double> ls = new SinglyLinkedList<>();  
// ... initialize ls ...  
double sum = 0.0;  
for (int i = 0; i < ls.size(); i++) {  
    → sum += ls.get(i);  
}
```



i 1      sum 100

## Each program iterates

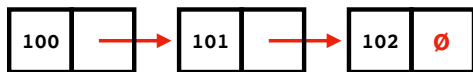
```
List<Double> ls = new SinglyLinkedList<>();  
// ... initialize ls ...  
double sum = 0.0;  
for (int i = 0; i < ls.size(); i++) {  
    → sum += ls.get(i);  
}
```



i 1      sum 201

## Each program iterates

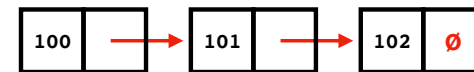
```
List<Double> ls = new SinglyLinkedList<>();  
// ... initialize ls ...  
double sum = 0.0;  
→ for (int i = 0; i < ls.size(); i++) {  
    sum += ls.get(i);  
}
```



i 2      sum 201

## Each program iterates

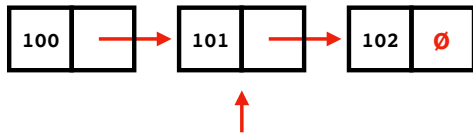
```
List<Double> ls = new SinglyLinkedList<>();  
// ... initialize ls ...  
double sum = 0.0;  
→ for (int i = 0; i < ls.size(); i++) {  
    sum += ls.get(i);  
}
```



i 2      sum 201

## Each program iterates

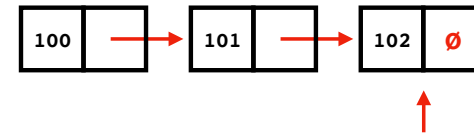
```
List<Double> ls = new SinglyLinkedList<>();  
// ... initialize ls ...  
double sum = 0.0;  
for (int i = 0; i < ls.size(); i++) {  
    sum += ls.get(i);  
}
```



i 2      sum 201

## Each program iterates

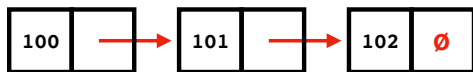
```
List<Double> ls = new SinglyLinkedList<>();  
// ... initialize ls ...  
double sum = 0.0;  
for (int i = 0; i < ls.size(); i++) {  
    sum += ls.get(i);  
}
```



i 2      sum 303

## 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);  
}
```

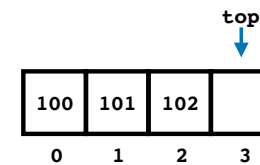


i 2      sum 303

"Iteration is terminated!"

## Each program iterates

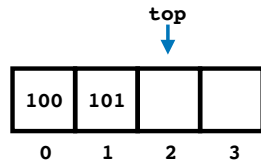
```
Stack<Double> s = new StackVector<>();  
// ... initialize s ...  
double sum = 0.0;  
while (!s.isEmpty()) {  
    sum += s.pop();  
}
```



sum 0

## Each program iterates

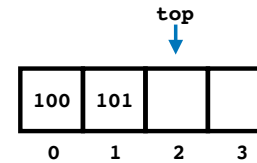
```
Stack<Double> s = new StackVector<>();  
// ... initialize s ...  
double sum = 0.0;  
while (!s.isEmpty()) {  
    sum += s.pop();  
}
```



sum 102

## Each program iterates

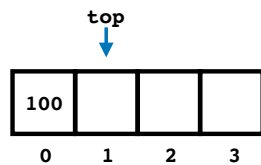
```
Stack<Double> s = new StackVector<>();  
// ... initialize s ...  
double sum = 0.0;  
while (!s.isEmpty()) {  
    sum += s.pop();  
}
```



sum 102

## Each program iterates

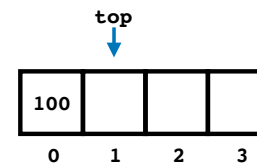
```
Stack<Double> s = new StackVector<>();  
// ... initialize s ...  
double sum = 0.0;  
while (!s.isEmpty()) {  
    sum += s.pop();  
}
```



sum 203

## Each program iterates

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

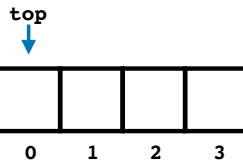


sum 203



## Each program iterates

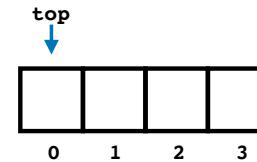
```
Stack<Double> s = new StackVector<>();  
// ... initialize s ...  
double sum = 0.0;  
while (!s.isEmpty()) {  
    sum += s.pop();  
}
```



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>**

```
structures5
Interface Stack<E>
All Superinterfaces:
    java.lang.Iterable<E>, Linear<E>, Structure<E>
All Known Implementing Classes:
    AbstractStack, StackArray, StackList, StackVector
```

```
structures5
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**.

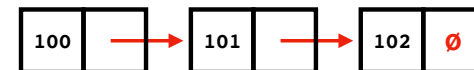
Importantly, `Iterators` are **stateful**.

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

Let's look at `SinglyLinkedList<T>`

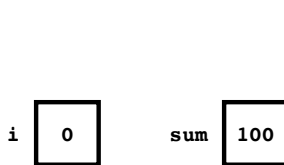
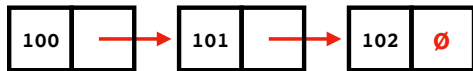
Naive iteration makes  $O(n)$  operation  $O(n^2)$ !

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



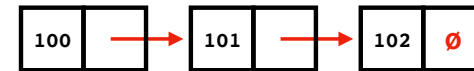
## Naive iteration makes $O(n)$ operation $O(n^2)$ !

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



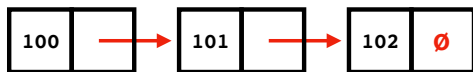
## Naive iteration makes $O(n)$ operation $O(n^2)$ !

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



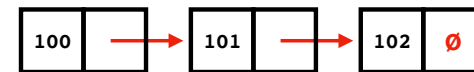
## Naive iteration makes $O(n)$ operation $O(n^2)$ !

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



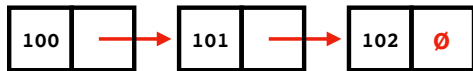
## Naive iteration makes $O(n)$ operation $O(n^2)$ !

```
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 $O(n^2)$ !

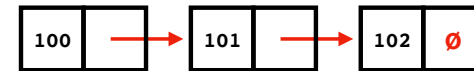
```
List<Double> ls = new SinglyLinkedList<>();  
// ... initialize ls ...  
double sum = 0.0;  
→ for (int i = 0; i < ls.size(); i++) {  
    sum += ls.get(i);  
}
```



i 2      sum 201

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

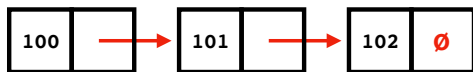
```
List<Double> ls = new SinglyLinkedList<>();  
// ... initialize ls ...  
double sum = 0.0;  
→ for (int i = 0; i < ls.size(); i++) {  
    sum += ls.get(i);  
}
```



↑  
i 2      sum 201

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

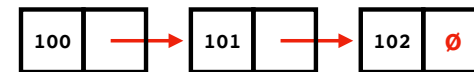
```
List<Double> ls = new SinglyLinkedList<>();  
// ... initialize ls ...  
double sum = 0.0;  
→ for (int i = 0; i < ls.size(); i++) {  
    sum += ls.get(i);  
}
```



↑  
i 2      sum 201

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

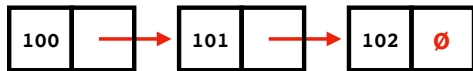
```
List<Double> ls = new SinglyLinkedList<>();  
// ... initialize ls ...  
double sum = 0.0;  
→ for (int i = 0; i < ls.size(); i++) {  
    sum += ls.get(i);  
}
```



↑  
i 2      sum 303

## 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);  
}
```



i 2      sum 303

"Iteration is terminated!"

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

The following code

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

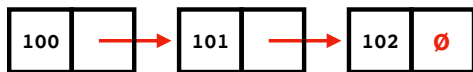
is the moral equivalent to

```
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;  
}
```

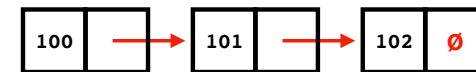


sum 0

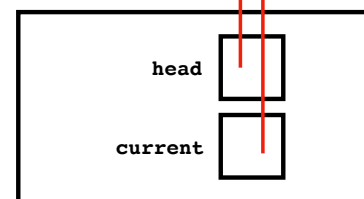
d 0

## Example.

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



Iterable<Double>

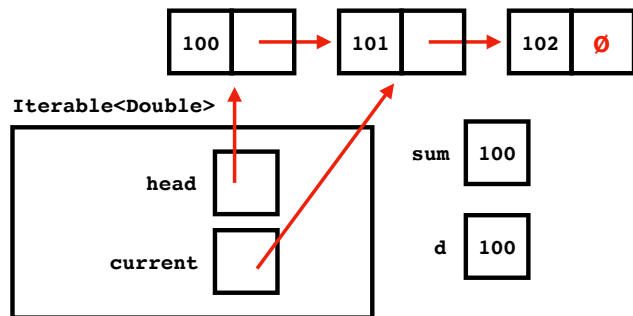


sum 0

d 0

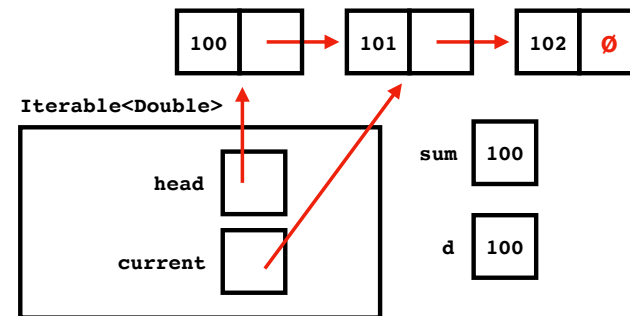
## Example.

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



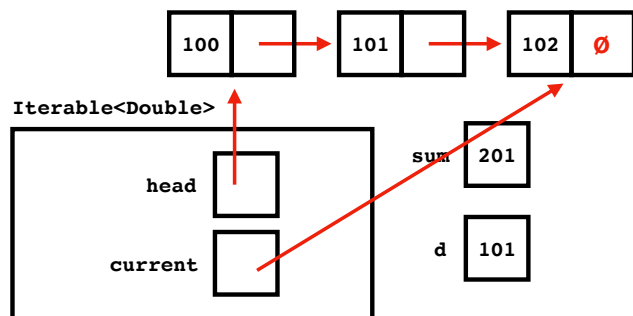
## Example.

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



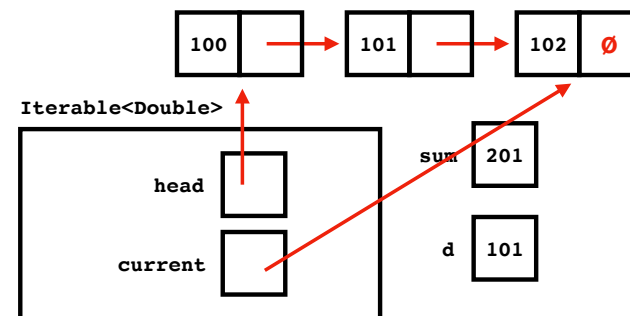
## Example.

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



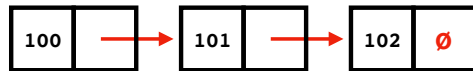
## Example.

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

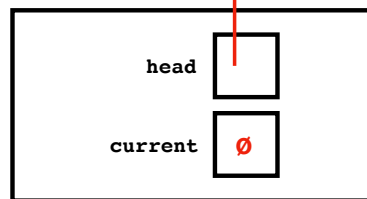


## Example.

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



Iterable<Double>

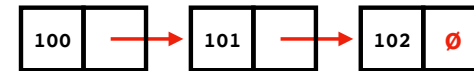


sum 303

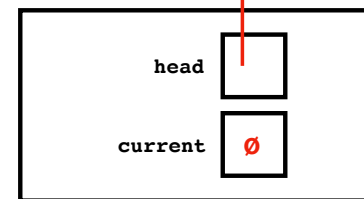
d 102

## Example.

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



Iterable<Double>



sum 303

d 102



"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

Looking for the value **322**.

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

↑ (red)                      ↑ (blue)                      ↑ (red)

## Binary search

Looking for the value **322**.

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

↑ (red)                      ↑ (blue)                      ↑ (red)

**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

↑ (red)                      ↑ (red)                      ↑ (blue)

## Binary search

Looking for the value **322**.

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

↑ (red)                      ↑ (blue)                      ↑ (red)

**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

↑ (blue arrow at index 1)  
↑ (red arrow at index 2)  
↑ (red arrow at index 2)

## Binary search

Looking for the value **322**.

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

↑ (red arrow at index 2)  
↑ (red arrow at index 2)  
↑ (blue arrow at index 2)

**322 = 322? yes**  
**return 2**

## Recap & Next Class

### Today:

Iteration

Binary search

### Next class:

Ordered structures