CSCI 136: Data Structures and Advanced Programming

Lecture 19
Search

Instructor: Dan Barowy

Your to-dos

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

Topics

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

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

  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?

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

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

```java
Stack<Double> s = new StackVector();
// _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.

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

Each program iterates

<table>
<thead>
<tr>
<th>i</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

Each program iterates

<table>
<thead>
<tr>
<th>i</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>
Each program iterates

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

```
100 101 102 103
0 1 2 3

i 1  sum  201
```

Each program iterates

```
100 101 102 103
0 1 2 3

i  2  sum  201
```

Each program iterates

```
100 101 102 103
0 1 2 3

i  2
```

Each program iterates

```
100 101 102 103
0 1 2 3

i  3
```

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

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

```
0 1 2 3
100 101 102 103
```

```
i 3 sum 406
```

```
Iteration is terminated!
```

Each program iterates

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

```
0 1 2 3
100 101 102 103
```

```
i 0 sum 0
```

```
100 101 102 Ø
```

```
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);
}
```

```
0 1 2 3
100 101 102 103
```

```
i 0 sum 100
```

```
100 101 102 Ø
```

```
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);
}

```
<table>
<thead>
<tr>
<th>i</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>201</td>
</tr>
<tr>
<td>3</td>
<td>Ø</td>
</tr>
</tbody>
</table>
```
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();
}

Each program iterates

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

sum 203

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();
}

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).

---

Iteration abstraction to the rescue.

```java
double[] a;
// _initialize a_
double sum = 0.0;
for (double d : a) {
    sum += d;
}
```

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

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

Brought to you by **Iterators**.

---

What if I told you that you could solve all of these problems with **abstraction**?

---

**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>`

(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>`.

Let’s look at `SinglyLinkedList<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.
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 0  sum 0
100 101 102 Ø

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 0  sum 100
100 101 102 Ø

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 1  sum 100
100 101 102 Ø
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 $O(n^2)$!

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

```
100 101 102 Ø
```

```
i 2 sum 303
```

```
“Iteration is terminated!”
```

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

The following code is the moral equivalent to

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

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

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

```
Example.
```

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

```
100 101 102 Ø
```

```
sum 0
d 0
```

```
“Iteration is terminated!”
```
Example.

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

Example.

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

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

```
head

100

101

102

\Ø

current

sum 201

d 101

```

```
Example.

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

```
head

100

101

102

\Ø

current

sum 303

d 102

```

```
Example.

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

```
head

100

101

102

\Ø

current

sum 303

d 102

```

```
Example.

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

```
head

100

101

102

\Ø

current

sum 303

d 102

```

Efficient searching: binary search
Binary search

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.

Important precondition: array must be sorted.

Looking for the value 322.
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.

322 = 101? no
322 < 101? no
322 > 101? yes

Binary search

Looking for the value 322.

322 = 322? yes
return 2

Resubmission procedure
Remember: the goal of this course is mastery.

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

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

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).

---

Recap & Next Class

Today:
- Iteration
- Binary search
- Resubmission procedure

Next class:
- Ordered structures

---

Resubmission procedure

Sample:

2. **Troubleshooting**
   
   My fix was slightly wrong. Right before calling `random_string()`, I added
   
   ```
   char * arrerr[i] = malloc(sizeof(char) * MAXLEN);
   ```
   
   when what I should have added is
   
   ```
   arrerr[i] = malloc(sizeof(char) * MAXLEN);
   ```
   
   *chck* (arrerr[i]);
   
   There is no need for "char *" because I am not declaring `arrerr`. I got my explanation and drawing wrong. In my drawing, I had `arrerr[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 `arrerr[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 `malloc()` or manipulating `arrerr[i]` in `random_string()`, we are likely to get memory errors. Below is what I should have drawn.

---

![Diagram](image)