CSCI 136: Data Structures and Advanced Programming

Lecture 17
Linear structures, part 2

Instructor: Dan Barowy
Williams

Topics

• Stack data structure
• Queue ADT
• Queue data structure
• Resubmission procedure

Your to-dos

1. Lab 6 (partner lab), due Tuesday 11/1 by 10pm.
   (two weeks!)
2. Review readings for midterm.

Announcements

• Colloquium: What I Did Last Summer (Research), 2:35pm in Wege Auditorium with cookies.
• Practice midterm posted on the course website.
• TA feedback.
Announcements

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

Applications **due Friday, October 28.**

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

Queue ADT

A **queue** is an **abstract data type** that stores a collection of **any type of element**. A queue **restricts which elements are accessible**: elements may only be added to the "end" of the collection and elements may only be removed from the "front" of a collection. The "enqueue" operation places an element at the end of a queue while a "dequeue" operation removes an element from the front.

Queue ADT

Also sometimes referred to as a **FIFO**: “**first in, first out.**”

(a stack would be an annoying way to process a line at Starbucks!)

Frequently used as a **buffer** to hold work to do later.

We also frequently include a "peek" operation that lets us look at an element on the top of a queue without removing it, and "size" and “**isEmpty**” operations that let us check how many elements are stored and whether a queue stores zero elements, respectively.
Queue implementations

QueueArray

A QueueArray is a queue implemented using an array for element storage.

Pros: enqueue and dequeue are O(1) operations.

Cons: data structure has a maximum capacity.

Queue implementations

QueueVector

A QueueVector is a queue implemented using a Vector for element storage.

Pros: enqueue and dequeue are amortized O(1) operations. There is no maximum capacity.

Cons: Most of the time, they take O(1) time, but occasionally--when the underlying array needs to grow--an O(n) cost is incurred. This may be fine for most applications, but if the application cannot tolerate wide variation in time, this is a bad choice. Also, unless the underlying array is completely full, Vectors waste some space.

Queue implementations

QueueList

A QueueList is a queue implemented using a List (usu. DLL or CL) for element storage.

Pros: enqueue and dequeue are O(1) operations. There is no maximum capacity. enqueue and dequeue costs are predictable (always the same), unlike QueueVector.

Cons: because of the way computer hardware is implemented, a QueueList's constant-time cost is likely to be much higher than a QueueVector's. So a QueueList's performance may be more predictable than a QueueVector, but it will likely be slower on average.

Other queue-like ADTs

One very useful and interesting variant of the Queue ADT is the Priority Queue ADT. We'll talk about priority queues after the midterm!
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 or put in mailbox).

---

**Sample from CS334:**

2. **Troubleshooting**

   My fix was slightly wrong. Right before calling `random_string()`, I added
   ```c
   char * arrarr[i] = malloc(sizeof(char) * MAXLEN);
   ```
   when what I should have added is
   ```c
   arrarr[i] = malloc(sizeof(char) * MAXLEN);
   ```

   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.
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 step in an iteration performs the given process once; the result of each step is the starting point of the next step.

Each program iterates

```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];
}
```
Each program iterates

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

<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];
}
```

<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>
</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];
}
```

<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>303</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];
}
```

<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>303</td>
</tr>
</tbody>
</table>
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);
}

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);
}
```

```
| i | 0 | sum | 100 |
```

```
| i | 1 | sum | 100 |
```

```
| i | 1 | sum | 201 |
```
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);
}
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>101</td>
<td>102</td>
<td>Ø</td>
</tr>
</tbody>
</table>
```

```
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);
}
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>101</td>
<td>102</td>
<td>Ø</td>
</tr>
</tbody>
</table>
```

```
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);
}
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>101</td>
<td>102</td>
<td>Ø</td>
</tr>
</tbody>
</table>
```

```
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);
}
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>101</td>
<td>102</td>
<td>Ø</td>
</tr>
</tbody>
</table>
```

```
i   2   sum  303
```
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);
}
```

Iteration is terminated!

Each program iterates

```java
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: 203

Iteration is terminated!
Essentially the same algorithm!

```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();
}
```

But the code looks different.

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

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**.
Iterators are a really good idea.

- Invented by Barbara Liskov in 1974.
- Incidentally, abstract data types were also invented by Barbara Liskov in 1974.
- (1974 was a good year!)
- Both debuted in an influential PL named CLU.
- Barbara won the Turing Award in 2008 for this work and more.

• Remember this from Python?

```java
for num in nums{
    // do something
}
```

Java has it too!

```java
for (int num : nums) {
    // do something
}
```

(admittedly, it is less pretty in Java)

This is called a “for-each loop”

Recap & Next Class

**Today:**

- Queue ADT
- Queue implementation
- Resubmissions

**Next class:**

- Iterators
- Search