CSCI 1103: Array-based Data Structures

Chris Kauffman

Last Updated:
Fri Nov 17 09:57:28 CST 2017
## Logistics

<table>
<thead>
<tr>
<th>Date</th>
<th>Lecture</th>
<th>Outside</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 11/13</td>
<td>Expandable Arrays</td>
<td>Lab 10 on Stacks</td>
</tr>
<tr>
<td>Wed 11/15</td>
<td>Stacks/Queues</td>
<td>P4 Due</td>
</tr>
<tr>
<td>Fri 11/17</td>
<td>Queues</td>
<td></td>
</tr>
<tr>
<td>Mon 11/20</td>
<td>Review</td>
<td>Lab 10 Due, Review</td>
</tr>
<tr>
<td>Wed 11/22</td>
<td>Exam 2</td>
<td></td>
</tr>
</tbody>
</table>

### Reading from Eck

- Ch 5 on Objects/Classes
- Ch 8.3.3 on Throwing exceptions
- Ch 7.4 on ArrayList
- Ch 9.3 on Stacks and Queues

### Lab10: Stack Data Structure

- Define a new class for Stacks of Strings
- Fixed and Expandable

### Project 4

- Due Wednesday
- Questions?
Exceptions for Errors

- Java’s mechanism for indicating errors is to throw exceptions
- There are a wide variety of exception kinds available
- Can also create your own: they are a class
- For simple situations, RuntimeException suffices
- Construct one with a String error message indicating problem
  
  ```java
  RuntimeException e = new RuntimeException("Ya done messed up.");
  ```
- Raise the exception with the keyword throw
  
  ```java
  throw e;
  ```
- Frequently do this in one-liners
  
  ```java
  throw new RuntimeException("Ya done messed up.");
  ```
Exceptions share return semantics

- Uncaught throw statements immediately exit a method, similar to return
- Control flows up and out, usually crashes program

// Divide num by denom and return the quotient.
// Raise a RuntimeException if denom is 0.
public static int divide(int num, int denom) {
    if (denom == 0) {
        throw new RuntimeException("Divide by 0");   // error: immediately
    }                                               // throw exception
    int quotient = num / denom;
    return quotient;                                 // immediately return
}                                                  // value
Additional Info on Exceptions

- We will work with `RuntimeExceptions` as they are simple sufficient
- Exceptions are a complex topic, include
  - `try/catch` blocks to recover from exceptions
  - method signatures with `throws`
  - inheritance of exception types
- We will revisit some of these topics later when discussing File Input/Output as many methods in I/O involve exception handling
Basic Data Structures

- Information frequently comes/goes in patterns
- To make life easier for programmers and utilize the machine more efficiently, data structures provide a way to organize data for easy use
- The purpose of a creating data structure is to make programming another task easier
- We will discuss some simple data structures
  - Expandable Arrays (today)
  - Stacks built on arrays (lab 10)
  - Queues built on arrays (later in week)
- Textbook discusses some alternatives
  - Linked lists
  - Stacks built from linked nodes
  - Queues built for linked nodes
- You will likely study these in later CS courses
Expandable Data Structures

Standard Array

- Recall Java’s standard arrays
  1. Length is fixed at creation
  2. Initially filled with zeroey elements (0 or null or similar)
  3. Random access based on index number using square braces: arr[i]
  4. Cannot grow
- Inability to grow is a drag as one frequently wants to add without knowing limit
- The goal of an expandable array or ArrayList is to making adding possible

Expandable List

- Independent class created by us (and others)
  1. Length is NOT fixed
  2. Initially empty: size 0
  3. Random access based on index number using methods: a.get(i) and a.set(i,x)
- No magic: a field of the expandable list will be a standard array
- When standard array fills up, make a bigger one, copy over elements
First pass: **FixedList doesn’t grow**

<table>
<thead>
<tr>
<th>Create/Initial Add</th>
<th>Further Adds/Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Welcome to DrJava.</td>
<td></td>
</tr>
<tr>
<td>&gt; FixedList f = new FixedList(3);</td>
<td>&gt; f.add(&quot;B&quot;)</td>
</tr>
<tr>
<td>&gt; f.toString()</td>
<td>&gt; f.toString()</td>
</tr>
<tr>
<td>[]</td>
<td>[A, B]</td>
</tr>
<tr>
<td>&gt; f.size()</td>
<td>&gt; f.size()</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>&gt; f.get(2)</td>
<td>&gt; f.add(&quot;C&quot;)</td>
</tr>
<tr>
<td>java.lang.RuntimeException:</td>
<td>&gt; f.toString()</td>
</tr>
<tr>
<td>out of bounds</td>
<td>[A, B, C]</td>
</tr>
<tr>
<td>at FixedList.get(FixedList.java:22)</td>
<td>&gt; f.size()</td>
</tr>
<tr>
<td>&gt; f.add(&quot;A&quot;)</td>
<td>3</td>
</tr>
<tr>
<td>&gt; f.size()</td>
<td>&gt; f.get(2)</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
</tr>
<tr>
<td>&gt; f.toString()</td>
<td>&gt; f.set(1,&quot;X&quot;)</td>
</tr>
<tr>
<td>[A]</td>
<td>&gt; f.toString()</td>
</tr>
<tr>
<td>&gt; f.get(0)</td>
<td>[A, X, C]</td>
</tr>
<tr>
<td>A</td>
<td>&gt; f.add(&quot;D&quot;)</td>
</tr>
<tr>
<td>&gt; f.get(1)</td>
<td>java.lang.RuntimeException:</td>
</tr>
<tr>
<td>java.lang.RuntimeException:</td>
<td>list array is full</td>
</tr>
<tr>
<td>out of bounds</td>
<td>at FixedList.add(FixedList.java:40)</td>
</tr>
</tbody>
</table>
Define size()

```java
public class FixedList{
    // number of elements
    // that have been added
    private int size;

    // contents of the array
    private String[] data;

    // Create the array backing
    // the fixed list
    public FixedList(int maxSize){
        this.size = 0;
        this.data = new String[maxSize];
    }

    // Return how many elements
    // are in the list
    public int size(){
        // YOUR CODE HERE
    }
}
```

Define set()

```java
public void set(int i, String x){
    // YOUR CODE HERE
}
```
Exercise: add() Method

Define add(x) method that allows new elements to be put in the list at the end increasing the size

```java
> f.toString()
[]
> f.add("A")
> f.add("B")
> f.toString()
[A, B]
> f.size()
2
```

```java
public class FixedList{
    // number of elements that have been added
    private int size;
    // contents of the array
    private String[] data;

    // Add the given string to the list at the end. If there is not
    // sufficient space for the addition, throw an exception
    public void add(String x){
        // YOUR CODE HERE to:
        // Check for space in array, throw exception if none
        // Put x in array
        // Increment size
    }
}
ExpandableList: Grow the Array

A modification to add(x) allows as many additions as memory supports: allocate larger arrays and copy when needed.

- **Draw pictures** to demonstrate how add(x) works
- **How much** does the array size increase during expansion?

// A class wrapper for a list of Strings. This version grows the underlying array when needed.

```java
public class ExpandableList{
    private int size; // number of elements that have been added
    private String[] data; // contents of the array

    // Add the given string to the list at the end. If there is not sufficient space for the addition, expand the underlying array to accommodate it.
    public void add(String x){
        if(this.size >= this.data.length){ // check for space
            String newData[] = new String[this.data.length*2]; // new larger array
            for(int i=0; i<this.data.length; i++){ // copy old elements
                newData[i] = this.data[i];
            }
            this.data = newData; // point at new array
        }
        this.data[this.size] = x; // add on element
        this.size++; // increase size
    }
}
```
Exercise: Removal in Lists

- Another common operation is removal: get rid of an element at a specific index
- List semantics dictate no gaps so much shift elements to account for this change
- Propose how one might write remove(i)
  - What fields must change and how?
  - What control structures are needed?

```plaintext
> l
[A, B, C, D, E] // 5 elements
> l.remove(2) // remove C
> l
[A, B, D, E]   // elements shifted
> l.size()
4           // size smaller
> l.add("F")
> l
[A, B, D, E, F] // 5 elements again
> l.remove(0) // remove A
> l
[B, D, E, F]  // elements shifted
> l.remove(2) // remove E
> l
[B, D, F]     // elements shifted
> l.size()
3     // down to 3 elements
```
Answer: Removal in Lists

Removal requires a loop to shift elements left in the array, decrease the size of the list.

```java
public void remove(int i){
    if(i < 0 || i >= this.size){
        throw new RuntimeException("out of bounds");
    }
    // shift elements to overwrite index i
    for(int j=i; j<this.size-1; j++){
        this.data[j] = this.data[j+1]; //
    }
    this.size--; // fewer elements
    this.data[size]=null; // nullify last element
}
```
Exercise: Stacks

Another major data structure, covered in Lab 10

Questions

▶ From lab work, what are the main operations of the stack?
▶ Where have we seen stacks used so far?
▶ How are stacks and expandable lists related?
▶ How are stacks and expandable lists different?
▶ What options exist when adding into a stack and the backing array is full (at capacity)?

Stacks are a LIFO: Last In First Out
Answers: Stacks

- **Stack Operations:**
  - `s.getTop()`: return whatever is on top
  - `s.push(x)`: put x on top
  - `s.pop()`: remove whatever is on top
  - `s.isEmpty()`: true when nothing is in it, false o/w

- Where have we seen stacks used so far?
  - Function call stack, contains data for running methods

- How are stacks and expandable arrays related?
  - Both backed by an array, `arr.add(x)` like `stack.push()`

- How are stacks and expandable arrays different?
  - Array allows get/set of any element, stack can only change top

- What options exist when pushing into a stack and the backing array is full (at capacity)?
  1. Throw an exception and ignore request
  2. Allocate a larger array, copy elements, proceed with push
Get in Line

 Queues are pervasive in computing and life
  ▶ Examples?
  ▶ Semantics?

Source: kittylittered
Queue Data Structure

Operations

- **enqueue(x)**: x enters at the back
- **dequeue()**: front leaves
- **getFront()**: return who’s in front
- **isEmpty()**: true when nothing is in it, false o/w

Implementation with arrays: seems easy...

- Enqueue elements at low indices like `list.add(x)`
- Dequeue elements by removing at index 0 like `list.remove(0)`
- Leads to a lot of shifting
- For efficiency, **never shift**
- Move front/back in a ring-like fashion
Efficient Array Queue in Pictures

1: Initial

2: q.dequeue();

3: q.enqueue("E")

4: q.dequeue()

5: q.enqueue("F")

6: q.enqueue("G")

7: q.enqueue("H"); // expand and re-order
Tricky to Implement

- Must wrap front/back around as they move off end of array
- On expansion must copy elements carefully and wrap around
- toString() must also account for wrap-around effect

```java
public class ArrayQueue{

    // Produce a string representation of the queue with the front
    // element leftmost followed by other elements to the right
    public String toString(){
        if(this.size==0){
            return "[]";
        }
        String str = "[" + this.data[this.front];
        for(int i=1; i<this.size; i++){
            int index = (this.front+i) % this.data.length;
            str += ", " + this.data[index];
        }
        str += "]";
        return str;
    }
}
```
Data Typing and Generics

- Notice our expandable list, stack, and queue all use String
- If you want a queue of integers, must recode: lots of redundancy
- In old Java (version 1.0-1.4) had bad set of choices for data structures and containers due to type problems
- Java 1.5 introduced generics, lifted from C++
- Allows containers to work with any type of item
- Used extensively in Java’s standard library

```java
ArrayList<String> als = new ArrayList<String>();
als.add("A"); // add a string to expandable array
als.add("B");
ArrayList<Integer> ali = new ArrayList<Integer>();
als.add(1); // add an integer to expandable array
als.add(2);
ArrayDeque<Double> ard = new ArrayDeque<Double>();
ard.addLast(1.23); // add double to expandable queue
ard.addLast(4.56);
```
Inheritance: Sharing Code between Classes

- Notice that the code for FixedList and ExpandableList is almost identical
- Created FixedList then copied all methods to ExpandableList, made a small change to the add() method to allow expansion
- This situation is well-suited for inheritance

```java
public class FixedList {
    ..
}

public class ExpandableList extends FixedList{
    @Override
    public void add(String x){
        // do this method a little differently
    }
}
```

- ExpandableList implicitly inherits all methods and fields of FixedList: don’t need to be copy them
- Method add() is overridden to have a different behavior than the version in the parent class