Lab 6: Amortized

November 21, 2016

Contents

1 Required Background Reading **1**

2 Flexible Arrays **1**

3 Expanding Flexible Arrays **3**

4 Shrinking Flexible Arrays **3**

5 Dynamic Arrays in Pyret **4**

6 Just for fun: Additional Readings **4**

Objectives

By the end of this lab, you will:

- understand a new array-based data structure
- understand amortized analysis
- have more practice with big-O analysis and amortized analysis

1 Required Background Reading

Before beginning this lab, please read *Programming and Programming Languages (2016): Chapter 17, Halloween Analysis.* **(click here to open on the web)**

Once you and your partner are done reading, please move to the next section. Feel free to reference the textbook at any point during the lab.

2 Flexible Arrays

Suppose we want an ordered, variable-size data structure that gives quick access to all elements. Specifically, we need to be able to:

- insert elements at one end.
- get the kth element.

In Pyret, lists support insertion (and deletion) at one end in constant time, but getting the k^{th} element takes $O([k \to k])$ time. This means lists alone won't be sufficient.

Arrays (known as vectors in some languages) support accessing elements in constant time, but are fixed in size. However, the fixed size of arrays is a major limitation. Once an array is full, we can no longer insert elements.

To solve this problem, we start out with a vector (another name for an array) of a certain size (assume a starting size of 1) and replace it with a bigger array when it is full. Here is a pseudocode description of this "flexible array":

```
makeFlexibleArray():
currentArray = new array[1]
nextUnusedIndex = size of currentArray

insertAtEnd(elem):
    if nextUnusedIndex > size of currentArray:
        newArray = new array[size of currentArray + 1]
        copy each element of currentArray to newArray
        nextUnusedIndex = size of currentArray + 1
        currentArray = newArray
        insertAtEnd(elem)
    else:
        currentArray[nextUnusedIndex] = elem
        nextUnusedIndex = nextUnusedIndex + 1

get(k):
    get kth element of currentArray
```

Task: What is the worst case (big-O) time complexity of `get`?

NOTE: For all tasks in this assignment, write all time complexities in the form $O([k \to k])$, define what each of your variables represent, and do not use n as a variable. Some examples of incorrect time complexities are: $[k \to k]$, $O[k \to k]$, or $O([k])$. Be ready to explain the reasoning behind your answer to a TA, so take notes if you need to.

NOTE: Also, be sure to actually write down your answers for this lab! We are expecting written answers for checkpoints!

Task: What is the worst case time complexity of `makeFlexibleArray`?

Task: What is the worst case time complexity of `insertAtEnd`?

Task: Is the amortized complexity of a sequence of k `insertAtEnd` operations any better? Assume that you start with the array of size 1.

⋆ **Before continuing, call over a TA to check that your answer is right.**

3 Expanding Flexible Arrays

Fortunately, there is a way to improve the amortized cost per operation of the flexible array. We change `insertAtEnd` so that the array doubles in size when it is full. Below is the pseudocode:

```
makeFlexibleArray():  -- unchanged
get(k): -- unchanged
insertAtEnd(elem):
    ONLY change is
        newArray = new array[size of currentArray + 1]
    is replaced by
        newArray = new array[size of currentArray * 2]
```

Task: Assume that you start with an empty array. What is the amortized cost per operation of k `insertAtEnd` operations?

⋆ **Before continuing, call over a TA to check that your answer is right.**

4 Shrinking Flexible Arrays

Suppose we want to extend our expanding flexible array to also support removing elements at the end. Specifically, we add the following operation:

```
deleteLast():
    delete last element of currentArray
```

To conserve memory, we may also want to shrink the array if it gets too empty. Here are two choices:

- When the array is half-full, shrink its size to half of what it was.
- When the array is one-fourth full, shrink its size to half of what it was.

Shrinking the array takes constant time. But you should assume that it notifies the operating system that it is taking less space, so the system is free to allocate the space

beyond the end of the shrunk region. That is, future allocation must again copy all elements.

Task: Which option is more efficient, and why? Give an example of a sequence of insertions and deletions with the corresponding amortized analysis to support your claim.

⋆ **Before continuing, call over a TA to check that your answer is right.**

5 Dynamic Arrays in Pyret

Now that you know a relatively efficient way of implementing flexible arrays, also known as dynamic arrays, you can implement them in Pyret. If you've programmed in Java, you may be familiar with `ArrayLists` – dynamic arrays are a primitive version of those.

Task: Implement dynamic arrays based on the pseudocode above, based on your decision in the last section. Be sure to include a proper set of tests – we will be checking for thorough testing!

NOTE: You should be using the Pyret array data structure. See the Pyret documentation for more info:
`https://www.pyret.org/docs/latest/arrays.html`

⋆ **Before continuing, call over a TA to check that your answer is right.** candy) and the amortized complexity?

6 Just for fun: Additional Readings

If you think amortized analysis is as *fun and exciting* as it seems, we recommend reading the following article titled "Amortized Analysis Explained":
`https://www.cs.princeton.edu/~fiebrink/423/AmortizedAnalysisExplained_Fiebrink.pdf`