Lecture 15: Dynamic Arrays: Analysis
10:00 AM, Feb 28, 2020

Contents

1 Picking up where we left off ... 1

2 Amortized Analysis 1

2.1 Aggregate Method ......................................................... 2

2.2 Accounting Method (Optional – for those who are interested) ............... 7

Motivating Question
What is the run-time performance of adding elements to a dynamic array? How close can we get to doing this in constant time?

Objectives
By the end of this lecture, you will know:

- what amortized analysis is
- the aggregate method for performing amortized analysis
- (optionally) the accounting (i.e., the banker’s) method for performing amortized analysis

By the end of this lecture, you will be able to:

- use the aggregate method to show that the run time of a sequence of additions to a dynamic array is amortized constant time

Credits: These notes, other than section 2, were largely written by Professor Amy Greenwald, with some edits by Kathi Fisler.

1 Picking up where we left off ...

These notes are the second half of the dynamic arrays lecture (which we gave a week ago). They continue where those notes left off.

We began this lecture by reviewing the material from the part 1 lecture, illustrating how the start and end markers move as we add items, how to handle empty lists, and what resizing the array looks like. Some of the material that we covered in class is in the part 1 lecture notes file. See the lecture capture for details.
2 Amortized Analysis

Recall from CS 17 that amortized analysis is a technique by which the run time of each of a sequence of operations is determined by averaging the aggregate run time over the sequence. There are three formal methods of amortized analysis.

The first, and most intuitive, is the aggregate method. Using this method, you simply sum up the individual run times of a sequence of \( n \) operations, and then divide by \( n \). While straightforward in principle, this method can be difficult to apply, because it is not always possible to express the summation in closed form. But have no fear: there are two other methods (which are actually almost one and the same, since they are so closely related to each other).

The accounting (or banker’s) method is a technique that relies on managing a bank account. Money is deposited into or withdrawn from the account with each operation, so that the bank balance represents savings accrued by a sequence of operations. When performing inexpensive operations extra money is deposited, thereby setting aside money to be withdrawn later to cover the cost of expensive operations. For this technique to work (i.e., for it to succeed at proving a bound), the bank balance can never be negative.

In this lecture, we will analyze adding elements to dynamic arrays using the aggregate method. There is an optional section on the accounting method, but this is not required (i.e., we won’t use it on labs, hwks, or the exams).

When performing an amortized analysis, the unit of analysis is a sequence of operations. In the case of dynamic arrays, what is one such worst-case sequence? Well, what is most expensive when it comes to dynamic arrays are growing and shrinking (i.e., resizing) the array. That is, adding and removing items, without resizing the array, take only constant time. (Getting an item always takes constant time; that is the whole point of dynamic arrays.)

So, the first problem we will tackle is when to grow the array; then, given that choice, we will consider what sequences of operations lead to poor performance.

For poor choices of when to resize the array, it is possible to construct sequences of operations that resize the array repeatedly, wreaking havoc on any analysis we might try to perform. So our goal in carrying out an amortized analysis of dynamic arrays will be to first make a reasonable choice about when to resize the array, and then figure out what a worst-case sequence of operations would look like given these choices.

In lecture, we will consider the problem of when to grow the array. For our choice, we will find that no interleaved sequence of add and remove operations can take longer to run than a homogeneous sequence of insertions, because interleaved sequences resize the array less often than homogeneous ones.

2.1 Aggregate Method

To begin our amortized analysis of inserting into a dynamic array, let’s start by asking ourselves how we should grow the array when it is full. Should we grow the array by 1 whenever it is full? That seems like a lot of excess copying. So should we instead grow our array by 10, or by 100, whenever it is full?

\[^1\text{or a bound on this aggregate}\]
\[^2\text{or a bound on those run times}\]
Here’s what it looks like if we initialize the capacity of the array to 1, and then insert 5 objects (letters), growing the array by 1, as necessary:

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert A</td>
<td>A</td>
</tr>
<tr>
<td>Grow Array, Insert B</td>
<td>A B</td>
</tr>
<tr>
<td>Grow Array, Insert C</td>
<td>A B C</td>
</tr>
<tr>
<td>Grow Array, Insert D</td>
<td>A B C D</td>
</tr>
<tr>
<td>Grow Array, Insert E</td>
<td>A B C D E</td>
</tr>
</tbody>
</table>

The array grows before B, is inserted. The array grows again before C is inserted. The array grows again before D is inserted. The array grows again before E is inserted.

Here’s what it looks like if we again initialize the capacity of the array to 1, and then insert 5 objects (letters), growing the array by 2, as necessary:

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert A</td>
<td>A</td>
</tr>
<tr>
<td>Grow Array, Insert B</td>
<td>A B</td>
</tr>
<tr>
<td>Insert C</td>
<td>A B C</td>
</tr>
<tr>
<td>Grow Array, Insert D</td>
<td>A B C D</td>
</tr>
<tr>
<td>Insert E</td>
<td>A B C D E</td>
</tr>
</tbody>
</table>

The array grows before B, is inserted. The array grows again before D is inserted.

To analyze dynamic arrays formally, we introduce the following notation:

- Let $\beta_i$ denote the capacity of the array before the $i$th insertion.
- Let $\alpha_i$ denote the capacity of the array after the $i$th insertion.
- Let $\gamma_i$ denote the cost of growing the array to accommodate the $i$th insertion. This cost is zero whenever the array is of already sufficient capacity. Otherwise, it is the cost of copying the contents of the array to a new, larger array. This cost is $O(i)$, since there are $O(i)$ items in the array.
- Let $c_i$ denote the cost of the $i$th insertion. We assume the cost of an insertion into an array of sufficient capacity is constant, specifically 1, so $c_i = 1 + \gamma_i$. 
Now, to apply the aggregate method, we record in a table costs over a sequence of operations. After the first 5 insertions into the array, costs accumulate like this:

<table>
<thead>
<tr>
<th>i</th>
<th>(\beta_i)</th>
<th>(\alpha_i)</th>
<th>(\gamma_i)</th>
<th>(c_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>


Now that we know the cost per insertion, the next step is aggregate: i.e., to total these costs.

Growing the array by 1, the total cost of \(n\) insertions is:

\[
\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} (1 + \gamma_i)
\]

\[
= n + \sum_{i=1}^{n} \gamma_i
\]

\[
= n + \sum_{i=0}^{n-1} i
\]

\[
= \sum_{i=0}^{n} i
\]

\[
= \sum_{i=1}^{n} i
\]

\[
= \frac{n(n+1)}{2}
\]

\[\in O(n^2)\]

Dividing this total evenly among \(n\) insertions yields an amortized cost of \(O(n)\).

But the cost of an insertion was already \(O(n)\) before we ever attempted an amortized analysis. So growing the array by 1 must not be the answer. What about growing the array by 2?

Growing the array by 2, the total cost of \(n\) insertions is:

\[
\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} (1 + \gamma_i)
\]

\[
= n + \sum_{i=1}^{n} \left\{ \begin{array}{ll}
  i - 1 & \text{if } i \text{ is even} \\
  0 & \text{otherwise}
\end{array} \right.
\]

\[
= n + \sum_{i=1}^{n} \left\{ \begin{array}{ll}
  i - 1 & \text{if } i - 1 \text{ is odd} \\
  0 & \text{otherwise}
\end{array} \right.
\]

Assume the number of insertions is even. (If the number of insertions is not even, then carry out
the analysis for the smallest even number greater than the number of insertions: i.e., \( n + 1 \). Now:

\[
\sum_{i=1}^{n} c_i = n + \left( 1 + 3 + 5 + \ldots + 2 \left( \frac{n}{2} \right) + 1 \right)
\]

the first \( \frac{n}{2} \) odd numbers

\[
= n + \sum_{i=0}^{\frac{n}{2}-1} (2i + 1)
\]

\[
\in n + O \left( \left( \frac{n}{2} \right)^2 \right)
\]

\[
\in O(n^2)
\]

Dividing this total evenly among \( n \) insertions again yields an amortized cost of \( O(n) \).

Let’s reflect on what we’ve discovered. Growing the array by 1 doesn’t help. Neither does growing the array by 2. What about growing the array by 10?

Growing the array by 10, the total cost of \( n \) insertions is:

\[
\sum_{i=1}^{n} c_i = n + \left( 1 + 11 + 21 + \ldots + 10 \left( \frac{n}{10} \right) + 1 \right)
\]

\[
\in n + O \left( \left( \frac{n}{10} \right)^2 \right)
\]

\[
\in O(n^2)
\]

Now’s about the time we give up on this strategy of growing the array (as necessary) by an additive constant and try something else instead. What else? Well, how about growing the array by a multiplicative factor? For example, what about doubling its capacity?

\[3\]If the number of insertions is not divisible by 10, then carry out the analysis for the smallest number divisible by 10 that is greater than the number of insertions.
The array grows before B, is inserted. The array grows again before C is inserted. The array grows again before E is inserted.

As above, we apply the aggregate method by recording in a table costs over a sequence of insertions. After the first 17 insertions into the array, costs accumulate like this:

<table>
<thead>
<tr>
<th>i</th>
<th>β_i</th>
<th>α_i</th>
<th>γ_i</th>
<th>c_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
<td>32</td>
<td>16</td>
<td>17</td>
</tr>
</tbody>
</table>

More generally, if we double the array, the total cost of \( n \) insertions is:

\[
\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} (1 + \gamma_i)
\]

\[
= n + \sum_{i=1}^{n} \left\{ \begin{array}{ll} i - 1 & \text{if } i - 1 \text{ is a power of 2} \\ 0 & \text{otherwise} \end{array} \right.
\]

\[
= n + \left( 1 + 2 + 4 + \ldots + 2^{\left\lfloor \log_2 n \right\rfloor - 1} \right) + 2^{\left\lfloor \log_2 n \right\rfloor}
\]

\[
= n + \sum_{j=0}^{\left\lfloor \log_2 n \right\rfloor} 2^j
\]

\[
= n + 2^{\left\lfloor \log_2 n \right\rfloor + 1} - 1
\]

\[
= n + (2)2^{\left\lfloor \log_2 n \right\rfloor} - 1
\]

\[
\leq n + 2n - 1
\]

\[
= 3n - 1
\]

\[
< 3n
\]

Amortizing this total cost over \( n \) insertions yields amortized constant cost. (At last!)
2.2 Accounting Method (Optional – for those who are interested)

The intuitive idea underlying the banker’s method is that the inexpensive operations before an expensive one “pay” the cost of the expensive one, thus averaging out costs over time.

To implement this idea, we associate with each operation a virtual cost $v_i$ in addition to its actual cost $c_i$. We then define the bank balance $b_n$ after $n$ operations as the difference between total virtual and total actual costs:

$$b_n = \sum_{i=1}^{n} v_i - \sum_{i=1}^{n} c_i$$

Given a virtual cost, which in conjunction with actual costs we can use to compute a running bank balance, there are two things to prove to establish an amortized constant run time:

1. For all $n$, $b_n \geq 0$: i.e., $\sum_{i=1}^{n} v_i \geq \sum_{i=1}^{n} c_i$.

2. There exists a constant $v \in \mathbb{N}_+$ that upper bounds all virtual costs: i.e., for all $i$, $v_i \leq v$.

If we can prove these two claims, then we will have established amortized constant time, because together they imply:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} v_i \leq \sum_{i=1}^{n} v = vn$$

Dividing through by $n$ yields the desired result:

$$\frac{1}{n} \sum_{i=1}^{n} c_i \leq v$$

In the case of dynamic arrays, we choose $v_i = \$$3, for all $i \geq 2$. First, we pay $\$$1 to cover the cost of inserting the $i$th item. Second, since we are inserting into an array that is already at least half full (by assumption $i \geq 2$, and the size of the array was initialized to 1), we set aside 2 extra dollars to pay for a future move, when the array becomes entirely full and its contents must be copied into a new array that is twice its size. One of the extra dollars is used to move item $i$, and the other is used to move item $i - s/2$, where $s$ is the current size of the array.

---

**Insert C and D**

Virtual Costs: $3$ $3$

Copy Costs: $2$ $2$

Initial Capacity: 4

New Capacity: 8

---

$^4$To establish amortized linear time, for example, you would prove that there exists a linear function that upper bounds all virtual costs. Likewise, for amortized quadratic time, etc.
When $i = 1$, and the array is empty, we choose $v_1 = 1$. In particular, one, rather than three, dollars suffice. We need only pay for the insert itself, as future items—specifically, those inserted after the array is already half full—will pay to copy any items inserted before the array is half full.

**Note:** If we initialize the array to a size other than 1, say $l$ (say $l = 2$) then we need only pay $\frac{1}{2}$ for the first $\lfloor \frac{l}{2} \rfloor$ inserts; none of these items need pay to copy an item already in the array since the array is initially empty, not half full, and no item need pay a copying cost until the array is half full.

It remains to establish the two necessary properties. Given our choices of $v_i$, the second property is trivial. Indeed, there exists a constant, namely 3, s.t. for all $i$, $v_i \leq v$ (since $1 \leq 3$ and $3 \leq 3$).

Finally, it remains to show that $b_n \geq 0$ for all $n$. This fact is apparent in the table below (where $l = 2$) for $n = 1, \ldots, 17$. To establish this claim for all $n$ requires a proof by induction.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\beta_i$</th>
<th>$\alpha_i$</th>
<th>$\gamma_i$</th>
<th>$c_i$</th>
<th>$v_i$</th>
<th>$b_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>9</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>13</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>14</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
<td>32</td>
<td>16</td>
<td>17</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>N/A</strong></td>
<td><strong>N/A</strong></td>
<td><strong>30</strong></td>
<td><strong>47</strong></td>
<td><strong>49</strong></td>
<td><strong>2</strong></td>
</tr>
</tbody>
</table>

**Theorem** If we initialize the array to be of size $l \geq 2$, and the bank balance $b_i = 0$, then after the $i$th insert in a sequence of consecutive insertions, $b_i \geq \max\{2(i - \lfloor \frac{\alpha_i}{2} \rfloor), 0\}$, for all $i \in \{1, 2, \ldots, \}$, where, as usual, $\alpha_i$ denotes the capacity of the array after the $i$th operation is performed.

**Corollary** For all $i \in \{1, 2, \ldots, \}$, after a sequence of $i$ insertions, the bank balance $b_i \geq 0$.

**Proof** We will prove this theorem in the special case where $l$ is a power of 2. In this special case, after the $i$th insert in a sequence of consecutive insertions, we can calculate the bank balance exactly: $b_i = \max\{2(i - \frac{\alpha_i}{2}), 0\}$, for all $i \in \{1, 2, \ldots, \}$,

The proof is by induction on $i$.

**Basis** In the base case, $0 < i \leq \frac{l}{2}$. With each insertion, nothing is deposited into the bank. But nothing is withdrawn either. So $b_i$ is 0 before and after the course of any such insertions. Moreover,
since $\alpha_i = l$, for all $0 < i \leq \frac{l}{2}$ (i.e., the array does not grow in the base case), it follows that $i \leq \frac{\alpha_l}{2}$. Therefore, $\max \left\{ 2 \left( i - \frac{\alpha_l}{2} \right), 0 \right\} = 0$.

**Step** Assume $i > \frac{l}{2}$, and further assume the induction hypothesis, namely that after the $i-1$st insertion, $b_{i-1} = 2 \left( (i-1) - \frac{\alpha_{i-1}}{2} \right)$. Since $i > \frac{l}{2}$, it follows that $2i \geq \alpha_i$ so that $i \geq \frac{\alpha_i}{2}$. Therefore, it suffices to show $b_i = 2 \left( i - \frac{\alpha_i}{2} \right)$ after the $i$th insertion.

As usual, let $\beta_i$ denote the capacity of the array *before* the $i$th operation is performed.

**Case 1:** The array is not yet full: i.e., $i \leq \beta_i$.

If the array is not yet full, then $\alpha_i = \alpha_{i-1}$. In addition, $v_i = 3$, while $c_i = 1$. The induction hypothesis, together with these observations, yields:

$$b_i = b_{i-1} + v_i - c_i$$
$$= b_{i-1} + 2$$
$$= 2 \left( (i-1) - \frac{\alpha_{i-1}}{2} \right) + 2$$
$$= 2 \left( (i-1) - \frac{\alpha_i}{2} \right) + 2$$
$$= 2 \left( i - \frac{\alpha_i}{2} \right)$$

**Case 2:** The array is full, so must grow before any further items can be inserted: i.e., $i = \beta_i + 1$.

As always, $b_i = b_{i-1} + v_i - c_i$. Our strategy will be to calculate $b_i$ in this way, then to evaluate $2 \left( i - \frac{\alpha_i}{2} \right)$ in this case, and finally to check that the former exceeds the latter.

For starters, since $i = \beta_i + 1$, it follows that $i-1$ (equivalently, $\beta_i$) is a multiple of $l$. Therefore, since we restricted our attention to a sequence of insertions only, $i-1 = \alpha_{i-1}$.

For starters, the array is full, and since we restricted our attention to a sequence of insertions only, $i-1 = \alpha_{i-1}$.

Now, by the induction hypothesis,

$$b_{i-1} = 2 \left( (i-1) - \frac{\alpha_{i-1}}{2} \right)$$
$$= 2 \left( \alpha_{i-1} - \frac{\alpha_{i-1}}{2} \right)$$
$$= 2 \alpha_{i-1} - \alpha_{i-1}$$
$$= \alpha_{i-1}$$
$$= i - 1$$

Next, as always $v_i = 3$.

Finally, in this case in particular, because the array is full, $c_i = i - 1 + 1$, namely the cost of copying all $i-1$ items into a newly grown array, plus the cost of inserting the $i$th item into this new array.

Therefore,

$$b_i = b_{i-1} + v_i - c_i$$
$$= i - 1 + 3 - ((i-1) + 1)$$
$$= 2$$
Next, as promised, we will evaluate the formula for $b_i$. The key step in this evaluation is the first one, which follows from the fact that the array grows in this case. In particular, $\alpha_i = 2\alpha_{i-1}$.

Therefore,

$$2 \left( i - \frac{\alpha_i}{2} \right) = 2 \left( i - \frac{2\alpha_{i-1}}{2} \right)$$

$$= 2 \left( i - \alpha_{i-1} \right)$$

$$= 2 \left( i - (i - 1) \right)$$

$$= 2(1)$$

$$= 2$$

Putting it all together, we note that $b_i = 2 = 2 \left( i - \frac{\alpha_i}{2} \right)$. Therefore, $b_i = 2 \left( i - \frac{\alpha_i}{2} \right)$. ✷

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [https://cs.brown.edu/courses/cs018/feedback]