A more efficient algorithm for appending data

James Ko

October 25, 2017

Contents

Abstract

This paper introduces **growth arrays**, array-like data structures that are designed for appending elements. When the number of items is not known beforehand but is expected to be large, they are more efficient than dynamic arrays by a constant factor. Growth arrays support all operations dynamic arrays do, such as random access, iteration, and insertion or deletion at an index. However, they perform no better, or slightly worse, than dynamic arrays for operations other than appending.

1 Introduction

In imperative languages, the dynamic array is the most common data structure used by programs. People often want to add multiple items to a collection and then iterate it, which dynamic arrays make simple and efficient. But can it be said they have the *most* efficient algorithm for this pattern?

In this paper, I introduce an alternative data structure to the dynamic array, called the **growth array**. It is more efficient than the dynamic array at appending large numbers of items. This is due to how it 'grows' once it cannot fit more items in its buffer.

When dynamic arrays run out of space, they allocate a new buffer, copy the contents of the old one into it, and throw the old one away. However, growth arrays are less wasteful. Instead of throwing away the filled buffer, they keep it a part of the data structure. The new buffer they allocate represents a continuation of the items from the old buffer. For example, if the old buffer contained items $0 - 31$, the new buffer would contain items 32 and beyond. Because of this, growth arrays allocate less memory to store the same number of items, and they do not need to copy items from the old buffer to the new one.

Growth arrays have caveats, however. They perform no better, or slightly worse, than dynamic arrays for operations other than appending. Random access, in particular, involves many more instructions than it does for dynamic arrays. Also, since growth arrays are not contiguous in memory, they may have poorer locality than dynamic arrays, and cannot be passed to external code that accepts contiguous buffers.

It is worth mentioning that if the size of the data is known in advance, both dynamic and growth arrays are completely unnecessary. A raw array could simply be allocated with the known size, and items could be appended to it just as quickly. Thus, growth arrays are only beneficial for cases where the amount of data to be appended is unknown, but is expected to be large.

2 Predefined Functions

I assume the following functions, which I will use in my algorithms, are already defined by the runtime.

```
▷ Copies $len$ items from $source$ to $dest$
$Array\ copy(source, dest, len)$

▷ Returns the length of $array$
$array.Len$

▷ Returns a new array with length $len$
$New\ array(len)$

▷ Returns a new, empty dynamic array
$New\ dynamic\ array()$
```

3 Fields and Properties

In subsequent sections, I will implement algorithms for both dynamic and growth arrays. In this section, I define fields and properties for these data structures which the algorithms will use. **Fields** are variables associated with an object that may be read from or written to. **Properties** are trivial, constant-time methods that do not change state.

If L is a dynamic array, then it is assumed to have the following fields:

- $L.Buf$ - The **buffer**, or raw array, that L stores its items in.
- $L.Size$ - The number of items in L.

As a dynamic array, L is also given the following properties. := denotes a definition, as opposed to = which checks equivalence. Functions that return boolean values are suffixed with ?.

```
▷ Returns the capacity of $L$
$L.Cap := L.Buf.Len$

▷ Returns whether $L$ is full
$L.Full? := L.Size = L.Cap$
```

When a dynamic array is instantiated, the following code should run:

```
**procedure** $Constructor(L)$
    $L.Buf \leftarrow New\ array(0)$
    $L.Size \leftarrow 0$
```

If L is a growth array, then it is assumed to have the following fields:

- $L.Head$ - The **head** of L. It returns the buffer we are currently adding items to.
- $L.Tail$ - The **tail** of L. It returns a dynamic array of references to buffers that have already been filled. This can be thought of as a two-dimensional array.
  Note: It may seem strange for growth arrays to use dynamic arrays internally. There is little harm from the extra copying and allocations these dynamic arrays perform, however, since only $O(\log n)$ references are appended to them (proved in [ref3]).
- $L.Size$ - The number of items in L.
- $L.Cap$ - The **capacity** of L. It returns the maximum number of items L can hold without resizing.

As a growth array, L is also given the following properties:

```
$\triangleright$ Returns whether $L$ is empty
$L.Empty? := L.Size = 0$

$\triangleright$ Returns whether $L$ is full
$L.Full? := L.Size = L.Cap$

$\triangleright$ Returns the capacity of $Head$
$L.Hcap := L.Head.Len$

$\triangleright$ Returns the size of $Head$
$\triangleright$ **Rationale:** $Cap - Hcap$ is the total capacity of the buffers in $Tail$.
$\triangleright$ Then, $Size - (Cap - Hcap)$ is the number of items that were added
$\triangleright$ after depleting the buffers in $Tail$.
$L.Hsize := L.Size - (L.Cap - L.Hcap)$
```

When a growth array is instantiated, the following code should run:

```
procedure $Constructor(L)$
    $L.Head \leftarrow New\ array(0)$
    $L.Tail \leftarrow New\ dynamic\ array()$
    $L.Size \leftarrow 0$
    $L.Cap \leftarrow 0$
```

4 Asymptotic Notation

4.1 Motivation

In order to highlight the benefit of growth arrays, I will use an alternative notation for time complexity. The reason for this is, for certain operations, growth arrays are only better than dynamic arrays by a constant factor. For example, dynamic arrays might allocate roughly $2n$ memory for appending n items, while growth arrays would allocate roughly n memory. Even though growth arrays are clearly better in this regard, the big-O space complexity for both data structures would be the same, $O(n)$.

My goal is to be able to compare the coefficients of the highest-order terms in both expressions. For example, I would like to take the ratio $\frac{2n}{n}$, see that it is 2, and conclude that dynamic arrays allocate roughly twice as much as growth arrays for large n. However, big-O notation does not support this.

4.2 Definition

To analyze time complexity, I will use the $\sim$ relation. It is defined as follows:

$$f \sim g \leftrightarrow \lim_{n\to\infty} \frac{g}{f} = 1$$

This is read as "f is asymptotic to g" or "f and g are asymptotic." **Note:** f and g are used as shorthand to denote $f(n)$ and $g(n)$, respectively.

Notice that while $O(2n) = O(n)$, $2n \nsim n$. Thus, $\sim$ makes it possible to distinguish between a function that uses n space and one that uses $2n$ space. **Note:** A consequence of this is that bases for logarithms cannot be omitted, like in big-O notation.

4.3 Properties

Here, I define various properties of the $\sim$ relation that will be used in my proofs. Proofs of these properties can be found in the appendix.

The following two theorems state that $\sim$ can "merge", or un-distribute, over arithmetic operations. This is a property shared with big-O.

Theorem 4.1. *P merges over addition and subtraction. That is, for functions f and g,*

$$(\hat{f} \sim f) \wedge (\hat{g} \sim g) \rightarrow \begin{cases} (\hat{f} + \hat{g}) \sim (f + g) \\ (\hat{f} - \hat{g}) \sim (f - g) \end{cases}$$

Theorem 4.2. *P merges over multiplication and division. That is, for functions f and g,*

$$(\hat{f} \sim f) \wedge (\hat{g} \sim g) \rightarrow \begin{cases} \hat{f}\hat{g} \sim fg \\ \hat{f}/\hat{g} \sim f/g \end{cases}$$

The following theorem states that lower-order terms may be removed. For example, $(n + \log_2 n) \sim n$.

Theorem 4.3. *If $\lim_{n\to\infty} g/f = 0$, then $f + O(g) \sim f$.*

The following theorem shows that $\sim$ is a transitive relation. This property is used implicitly by proofs that chain multiple $\sim$s in the form $f \sim g \sim h$, then conclude that $f \sim h$.

Theorem 4.4. *If $f \sim g$ and $g \sim h$, then $f \sim h$.*

The following theorem states that if two functions are asymptotic, then they belong to the same big-O class.

Theorem 4.5. *If $f \sim g$ and $g = O(h)$, then $f = O(h)$.*

5 Common Operations

In this section, I implement the most common operations for dynamic and growth arrays, then analyze their time complexity. If the operation allocates memory, I also analyze its space complexity.

5.1 Appending

Description

Appending is the most common operation done on dynamic arrays. Growth arrays improve the performance of appending in two ways: by allocating less memory, and by reducing the amount of copying.

Dynamic array implementation

I will implement appending for dynamic arrays first. Let L be a dynamic array. The following definitions are used in the code:

initial capacity Denoted by c_0. The capacity of a dynamic a array when one item is appended to it.
Note: The capacity of any dynamic array with size zero should be zero.

growth factor Denoted by g. The factor by which the current capacity is multiplied to get the new capacity when L grows.

```
procedure $Append(L, item)$
    if $L.Full?$ then
        $L.Grow()$
    $L.Head[L.Hsize] \leftarrow item$
    $L.Size \leftarrow L.Size + 1$

procedure $Grow(L)$
    if $L.Empty?$ then
        $L.Buf \leftarrow New\ array(c_0)$
    else
        $new\ buf \leftarrow New\ array(L.Size \times g)$
        $Array\ copy(L.Buf, new\ buf, L.Size)$
        $L.Buf \leftarrow new\ buf$
```

Time complexity

Before I analyze the time complexity of *Append*, I consider a different method for measuring its cost. Suppose I start with an empty collection and n elements are appended. How many times is an element stored in an array? I will term the answer to this question the **write cost** of n appends, and denote it $w(n)$.

In the code for *Append*, one array store is performed unconditionally, so it is apparent that $w(n) \geq n$ after n appends. However, *Grow* also does some writing, so in order to find a precise formula for $w(n)$, I need to analyze when *Grow* is called. To do this, I use the following lemma:

Lemma 5.1. *Let L by a dynamic array. Let its* ***capacity sequence****, C, be the capacities L takes on as n items are appended. Then*

$$C = 0,\ c_0,\ gc_0,\ g^2c_0,\ \ldots\ g^{\lceil \log_g n - \log_g c_0 \rceil} c_0$$

Proof. The capacity starts out at zero and becomes c_0 once one element is appended. From then on, it can only grow by a factor of g via *Grow*, so if $g^i c_0$ is the current capacity then $g^{i+1}c_0$ must be the next capacity. By induction, the rest of the sequence is

$$\left\{g^i c_0\right\}_{i=1}^{k}$$

for some k. Since the capacity is only as big as is needs to be, namely greater than or equal to n, k is the smallest integer for which $g^k c_0 \geq n$. Using this property, I now derive k.

$$\begin{aligned} g^k c_0 &\geq n \\ g^k &\geq \frac{n}{c_0} \\ k &\geq \log_g n - \log_g c_0 \\ k &\geq \log_g n - \log_g c_0 > k - 1 \\ k &= \left\lceil \log_g n - \log_g c_0 \right\rceil \end{aligned}$$

Then the final term in the sequence is $g^{\lceil \log_g n - \log_g c_0 \rceil} c_0$, completing the proof. □

[make lemma] Now, consider that to grow by g, *Grow* must be called. So $g^k c_0$ being present in S_C means *Grow* was called at size $g^{k-1}c_0$ for $k \geq 1$. Then *Grow* was called at the sizes

$$S_R = 0,\ c_0,\ gc_0,\ g^2c_0,\ \ldots\ g^{\lceil \log_g n - \log_g c_0 \rceil - 1} c_0$$

Each time *Grow* is called, i items are copied where i is the current size of the dynamic array. Then the total number of items copied for n appends is

$$0 + c_0 + gc_0 + \ldots + g^{\lceil \log_g n - \log_g c_0 \rceil - 1} c_0 = \left(\frac{g^{\lceil \log_g n - \log_g c_0 \rceil} - 1}{g - 1} \right) c_0$$

Finally, counting the writes made for every item in *Append*, I finally find an explicit formula for $w(n)$.

[ref1]

$$w(n) = n + \left(\frac{g^{\lceil \log_g n - \log_g c_0 \rceil} - 1}{g - 1} \right) c_0$$

[ref2] To make this expression easier to work with, I will write it in big-P notation to approximate it for large n. First I note that $\log_g k \leq \lceil \log_g k \rceil < \left(\log_g k\right) + 1$ for any $k > 0$, so $k \leq g^{\lceil \log_g k \rceil} < gk$. Then

$$\begin{aligned}
w(n) &= n + \left(\frac{g^{\left\lceil \log_g \left(\frac{n}{c_0} \right) \right\rceil} - 1}{g - 1} \right) c_0 \\
n + \left(\frac{\frac{n}{c_0} - 1}{g - 1} \right) c_0 \leq w(n) &< n + \left(\frac{g\frac{n}{c_0} - 1}{g - 1} \right) c_0 \\
n + \left(\frac{n - c_0}{g - 1} \right) \leq w(n) &< n + \left(\frac{gn - c_0}{g - 1} \right) \\
P\left(n + \left(\frac{n - c_0}{g - 1} \right) \right) \leq P\left(w(n)\right) &< P\left(n + \left(\frac{gn - c_0}{g - 1} \right) \right) \\
P\left(\left(\frac{g}{g - 1} \right) n \right) \leq P\left(w(n)\right) &< P\left(\left(\frac{2g - 1}{g - 1} \right) n \right)
\end{aligned}$$

Space complexity

I wish to find the space allocated while appending n items to a dynamic array. I denote this quantity $s(n)$, and define it as the total length of all arrays allocated during n *Append* calls. I now derive a formula for $s(n)$.

From 5.1 I know the exact capacities L takes on as n items are appended. Dynamic arrays also have the property that $i \in S_C$ iff an array of length i was allocated. Then the total length of arrays allocated is

$$\begin{aligned}
s(n) &= \sum_{i \in S_C} i \\
&= 0 + c_0 + gc_0 + g^2 c_0 + \ldots + g^{\lceil \log_g n - \log_g c_0 \rceil} c_0 \\
&= \left(\frac{g^{\lceil \log_g n - \log_g c_0 \rceil + 1} - 1}{g - 1} \right) c_0
\end{aligned}$$

Using the identity in [ref2] again, I bound its big-P complexity:

$$\left(\frac{g\frac{n}{c_0}-1}{g-1}\right)c_0 \le s(n) < \left(\frac{g^2\frac{n}{c_0}-1}{g-1}\right)c_0$$
$$P\left(\left(\frac{g\frac{n}{c_0}-1}{g-1}\right)c_0\right) \le P\left(s(n)\right) < P\left(\left(\frac{g^2\frac{n}{c_0}-1}{g-1}\right)c_0\right)$$
$$P\left(\left(\frac{g}{g-1}\right)n\right) \le P\left(s(n)\right) < P\left(\left(\frac{g^2}{g-1}\right)n\right)$$

Growth array implementation

Time complexity

I again start off by finding the write cost when n are appended. I give the write cost function a slightly different name, $\hat{w}(n)$, so I can compare it to $w(n)$ later. I also use $\hat{g}$ and $\hat{c_0}$ instead of g and c_0 for the growth factor and initial capacity, respectively.

5.1 still holds, since all of the assumptions made there are also true for growth arrays. In particular, although growth arrays grow differently than dynamic arrays, the following claim is still true:

Lemma 5.2. *The capacity of a growth array grows by a constant factor $\hat{g}$ after the first time it is called.*

Proof. I prove that the algorithm ensures this using induction.

For the base case, consider the second time *Grow* is called: the current capacity is $\hat{c_0}$, and the next (total) capacity should be $\hat{g}\hat{c_0}$. Then a buffer of size $(\hat{g}-1)\hat{c_0}$ should be appended, and that is what the algorithm does.

For $i \ge 2$, given the algorithm behaves correctly the ith time it is called, I wish to show it behaves correctly the $(i+1)$th time. From the inductive hypothesis, when the algorithm is called the $(i+1)$th time, the current capacity is $\hat{g}^i\hat{c_0}$ and the head capacity is $(\hat{g}^i-\hat{g}^{i-1})\hat{c_0}$. The next (total) capacity should be $\hat{g}^{i+1}\hat{c_0}$, and the next head capacity should be $(\hat{g}^{i+1}-\hat{g}^i)\hat{c_0}$, which is $\hat{g}$ times the current head capacity. The algorithm determines the capacity of the next buffer in this manner, proving the inductive step and completing the induction. $\square$

Now I take a look at *Grow*. Unlike dynamic arrays, the number of writes performed does not equal i if the current size is i. No items are being copied; the only source of writes is appending a buffer to the tail. The goal is to find the number of writes that causes.

Let $w_{Grow}(n)$ denote the total number of writes made by *Grow*. From [make lemma], I know that *Grow* is called at the sizes

$$S_R = 0,\ \hat{c_0},\ \hat{g}\hat{c_0},\ \hat{g}^2\hat{c_0},\ \ldots\ \hat{g}^{\lceil \log_{\hat{g}} n - \log_{\hat{g}} \hat{c_0} \rceil - 1}\hat{c_0}$$

Let k be the number of times the tail is appended to. Since the tail is a dynamic array, the number of writes it makes is $w(k)$. Since *Grow* is called $|S_R|$ times and it appends to the tail each time except for the first, $k = |S_R| - 1 = \lceil \log_{\hat{g}} n - \log_{\hat{g}} \hat{c_0} \rceil$. Then the formula for $w_{Grow}(n)$ is

$$w_{Grow}(n) = k + \left(\frac{g^{\lceil \log_g k - \log_g c_0 \rceil} - 1}{g - 1} \right) c_0$$

where c_0 g are the capacity of the *tail* (not the growth array). I finally conclude that

$$\hat{w}(n) = n + k + \left(\frac{g^{\lceil \log_g k - \log_g c_0 \rceil} - 1}{g - 1} \right) c_0$$

I now approximate this expression using $\sim$. First, I approximate k using big-O:

$$\begin{aligned} k &= \lceil \log_{\hat{g}} n - \log_{\hat{g}} \hat{c_0} \rceil \\ O(k) &= O(\log n) \end{aligned}$$

Now, the k-term disappears when using $\sim$ to approximate $\hat{w}(n)$, leaving just n:

$$\begin{aligned} \hat{w}(n) &\sim n + w(k) \\ &= n + O(\log n) \\ &\sim n \end{aligned}$$

Space complexity

The space function for n *Append* calls on a growth array is denoted $\hat{s}(n)$. Since growth arrays never throw away item buffers, if the current capacity is C then the total length of item buffers is also C. $\hat{s}(n)$ is usually greater than C, however; this is because growth arrays not only allocate item buffers, but stores references to them in the tail. Thus the space the tail allocates must also be determined.

I determined in [...] that if n are appended then the tail is appended to k times, where $k = \lceil \log_{\hat{g}} n - \log_{\hat{g}} \hat{c_0} \rceil$. Since the tail is a dynamic array, it allocates $s(k)$ space, where s is the space function of the tail. Through a similar method to [...] it can be shown $O(s(k)) = O(k) = O(\log n)$. Then

$$\hat{s}(n) \sim 2^{\lceil \log_2 n \rceil} = O(n)$$

Time complexity comparison

Space complexity comparison

5.2 Indexing

Description

Indexing is another very common operation on a list. I will call methods that get or set an item at a specified index **get** and **set indexers**, respectively.

5.3 Iterating

Description

Iteration of a list is the process of performing some action on each of its elements.

5.4 Copying to an array

Description

Users often want to take list structures, such as dynamic arrays, and convert them into plain arrays. There are multiple reasons why someone would want to do this after they are done appending to the list:

- Plain arrays hold on to exactly the amount of memory needed to hold their elements. However, dynamic and growth arrays allocate more space than necessary to optimize appending new items.

- The user wants to call a function in third-party code that takes a plain array as an argument.
-
- Plain arrays are contiguous, while growth arrays are fragmented and have worse locality.
- The indexer of growth arrays is several times slower than that of plain arrays, whether the $O(1)$ or $O(\log n)$ implementation is chosen.

6 Other Operations

7 Implementations

8 Benchmarks

9 Closing Remarks

Appendices

A Proofs of $\sim$ Properties

A.1 Merging over Arithmetic Operations

Proof (Theorem 4.1). □

Proof (Theorem 4.2). This can be easily shown by multiplying the limits corresponding to $\hat{f}$ and $\hat{g}$:

$$\begin{aligned}
\lim_{n\to\infty} \frac{\hat{f}}{f} &= 1 \\
\lim_{n\to\infty} \frac{\hat{g}}{g} &= 1 \\
\left(\lim_{n\to\infty} \frac{\hat{f}}{f}\right)\left(\lim_{n\to\infty} \frac{\hat{g}}{g}\right) &= 1 \cdot 1 \\
\lim_{n\to\infty} \frac{\hat{f}\hat{g}}{fg} &= 1
\end{aligned}$$

it follows that $\hat{f}\hat{g} \sim fg$. The statement about division can be proved by flipping the fraction for $\hat{g}$ before multiplying, resulting in

$$\lim_{n\to\infty} \frac{\hat{f}/\hat{g}}{f/g} = 1$$

This shows $\hat{f}/\hat{g} \sim f/g$. □

A.2 Removal of Lower-Order Terms

Proof (Theorem 4.3). Since

$$\begin{aligned}\lim_{n\to\infty}\left(\frac{f+g}{f}\right) &= \lim_{n\to\infty}\frac{f}{f} + \lim_{n\to\infty}\frac{g}{f} \\ &= 1+0 \\ &= 1\end{aligned}$$

it follows that $f+g \sim f$. □

A.3 Transitivity

Proof (Theorem 4.4). By definition, $\lim_{n\to\infty} g/f = 1$ and $\lim_{n\to\infty} h/g = 1$. Multiplying the two equations, $\lim_{n\to\infty} h/f = 1$ which implies $f \sim h$. □