A more efficient algorithm for appending data

James Ko

November 10, 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. **Note:** For pseudocode, I will use := to denote a definition, and = to denote an equivalence check. 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 $Constructor(L)$
  $L.Buf \leftarrow New\_array(c_0)$
  $L.Size \leftarrow 0$

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. **Important note: The tail is a dynamic array.** It returns a *dynamic array* of references to buffers that are already filled with items. The tail can be thought of as a two-dimensional array.
  Note: It may seem strange for a growth array to use the very data structure it is replacing. As shown in Lemma 5.4, however, only $O(\log n)$ many references are appended to the tail. Thus, the extra copying and allocations the tail performs is minuscule compared to other work done by the growth array.
- $L.Size$ - The number of items in L.
- $L.Cap$ - The **capacity** of L. It returns the maximum number of items L can hold before it must grow.

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

▷ Returns whether L is empty
$L.Empty? := L.Size = 0$

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

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

▷ Returns the size of $Head$
▷ **Rationale:** $Cap - Hcap$ is the total capacity of the buffers in $Tail$.
▷ Then, $Size - (Cap - Hcap)$ is the number of items that were added
▷ 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 $Constructor(L)$
    $L.Head \leftarrow New\_array(c_0)$
    $L.Tail \leftarrow New\_dynamic\_array()$
    $L.Size \leftarrow 0$
    $L.Cap \leftarrow c_0$

4 Asymptotic Notation

4.1 Introduction

Typically, big-O notation is used to analyze the time or space complexity of a function. In order to highlight the benefit of growth arrays, however, I will use the $\sim$ relation to analyze complexity. This is because for certain operations, growth arrays are only better than dynamic arrays by a constant factor. For example, dynamic arrays might use roughly $2n$ space for appending n items, while growth arrays would use roughly n space. 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 $2n/n$, 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 Definitions

I will mainly use the $\sim$ relation to analyze complexity. It is defined as follows:

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

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 $O(2n) = O(n)$, $2n \not\sim n$. 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.

I will also introduce the cousins of $\sim$, $\prec$ and $\preceq$. $f \preceq g$ if and only if $f \prec g$ or $f \sim g$. $\prec$ is defined as follows:

$$f \prec g \iff \lim_{n\to\infty} \frac{f}{g} < 1$$

4.3 Properties

Here, I define various properties of the $\sim$ relation that will be used in my proofs. These properties themselves are proved in the appendix.

The following theorem states that $\sim$ can "merge," or un-distribute, over addition, multiplication, and division. This is a property shared with big-O.

Theorem 4.1. *Suppose f, f_0, g, and g_0 are functions. If $f \sim f_0$ and $g \sim g_0$, then*

$$\begin{aligned} f + g &\sim f_0 + g_0 \\ fg &\sim f_0 g_0 \\ \frac{f}{g} &\sim \frac{f_0}{g_0} \end{aligned}$$

The following theorem states that lower-order terms may be removed: for example, $(n + \log_2 n) \sim n$. This is also a property shared with big-O.

Theorem 4.2. *If $\lim_{n\to\infty} \dfrac{g}{f} = 0$, then $f + O(g) \sim f$.*

The following theorem states 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 4.3. *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 4.4. *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

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 an empty dynamic array.
Assumptions: c_0 is an integer, $c_0 > 0$

growth factor Denoted by g. The factor by which the current capacity is multiplied to get the new capacity when L is non-empty and grows.
Assumptions: $gc_0 \geq c_0 + 1$

```
procedure Append(L, item)
    if L.Full? then
        L.Grow()
    L.Buf[L.Size] ← item
    L.Size ← L.Size + 1

procedure Grow(L)
    new_buf ← New_array(g × L.Size)
    Array_copy(L.Buf, new_buf, L.Size)
    L.Buf ← 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 be a dynamic array. Let its* ***capacity sequence****, κ, be the range of values for $L.Cap$ as n items are appended. For $n = 0$, trivially $\kappa = (c_0)$. For $n > 0$,*

$$\kappa = c_0,\ gc_0,\ g^2c_0,\ \ldots\ g^{\max(\lceil \log_g n - \log_g c_0 \rceil, 0)}c_0$$

Proof. I use the following properties of dynamic arrays:

1. The capacity of an empty dynamic array is c_0.

2. The capacity of a dynamic array can only grow by g.

3. The capacity is as small as possible. Put formally, if κ_i is the capacity for n items, then $\kappa_i \geq n$ but $n > \kappa_{i-1}$. (By convention, $\kappa_{-1} = 0$.)

Assumption (1) immediately shows $\kappa_0 = c_0$. Assumption (2) shows that if $g^i c_0$ is the current capacity, then $g^{i+1} c_0$ must be the next capacity. By induction, $\kappa = \left(g^i c_0\right)_{i=0}^{\lambda}$ for some whole number λ.

The final value of the sequence, κ_λ, is the capacity needed for n items. By assumption (3), $\kappa_\lambda \geq n > \kappa_{\lambda-1}$. Consider the case when $n > c_0$: it must be true that $\kappa_\lambda > c_0$, so $\lambda \geq 1$. Since $\lambda - 1 \neq 0$, $\kappa_\lambda = g^\lambda c_0$ and $\kappa_{\lambda-1} = g^{\lambda-1} c_0$. Then

$$\begin{aligned} g^\lambda c_0 &\geq n > g^{\lambda-1} c_0 \\ g^\lambda &\geq \frac{n}{c_0} > g^{\lambda-1} \\ \lambda &\geq \log_g n - \log_g c_0 > \lambda - 1 \end{aligned}$$

Since λ is an integer,

$$\lambda = \left\lceil \log_g n - \log_g c_0 \right\rceil$$

Now consider the case when $n \leq c_0$. By assumption (3), $n > \kappa_{\lambda-1}$. $\lambda - 1$ must then equal -1, since any other value would imply $n > \kappa_{\lambda-1} \geq c_0$. Thus $\lambda = 0$.

It was shown $\lambda \geq 1 \geq 0$ for the first case, and it can be shown $\left\lceil \log_g n - \log_g c_0 \right\rceil \leq 0$ for the second case. Then, a general formula for λ is as follows:

$$\lambda = \max(\left\lceil \log_g n - \log_g c_0 \right\rceil, 0)$$

The final term in the sequence is $g^\lambda c_0 = g^{\max(\lceil \log_g n - \log_g c_0 \rceil, 0)} c_0$, completing the proof. $\square$

Corollary 5.1.1. *Let the* ***growth sequence****, γ, of L be the sizes for which Grow is called when n items are appended. Then $\gamma = \kappa \setminus \{\kappa_\lambda\}$.*

Proof. If κ_i exists and $i \geq 1$, then clearly *Grow* must have been called when the size was κ_{i-1}, so $\kappa_{i-1} \in \gamma$. Then γ contains every term in κ except for the last, κ_λ, as the corollary states. $\square$

When *Grow* is called and the current size is γ_i, the algorithm copies γ_i items.

Then the total number of items copied when n items are appended is:

$$\sum_i \gamma_i = c_0 + gc_0 + \ldots + g^{\lambda-1}c_0$$
$$= \left(\frac{g^\lambda - 1}{g-1}\right) c_0$$

Counting the writes made for each item by *Append*, an explicit formula for $w(n)$ is as follows:

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

Now, my goal is to approximate $w(n)$ with $\sim$. To make is easier to do so, I will asymptotically bound g^λ which depends on n.

Lemma 5.2. *For* $n > c_0$, $\dfrac{n}{c_0} \leq g^\lambda < \dfrac{gn}{c_0}$.

Proof. It was shown in Lemma 5.1 that if $n > c_0$, $\lambda = \lceil \log_g n - \log_g c_0 \rceil \geq 1$. Now note that λ may also be written as $\left\lceil \log_g \frac{n}{c_0} \right\rceil$. Then

$$\log_g \frac{n}{c_0} \leq \lambda < \log_g \frac{n}{c_0} + 1$$
$$\frac{n}{c_0} \leq \lambda < \frac{gn}{c_0}$$

as desired. $\square$

Now, I proceed to asymptotically bound $w(n)$.

$$w(n) = n + \left(\frac{g^\lambda - 1}{g-1}\right) c_0$$
$$n + \left(\frac{n/c_0 - 1}{g-1}\right) c_0 \preceq w(n) \prec n + \left(\frac{gn/c_0 - 1}{g-1}\right) c_0$$
$$\left(\frac{g}{g-1}\right) n - \left(\frac{c_0}{g-1}\right) \preceq w(n) \prec \left(\frac{2g-1}{g-1}\right) n - \left(\frac{c_0}{g-1}\right)$$
$$\left(\frac{g}{g-1}\right) n \preceq w(n) \prec \left(\frac{2g-1}{g-1}\right) n$$

Space complexity

I wish to find the space allocated when n items are appended to a dynamic array. I call this quantity the **space cost**, denote it $s(n)$, and define it as the total length of buffers allocated by n *Append* calls. Now, I derive a formula for $s(n)$.

First, from the definition of $L.Cap$, note that a dynamic array's capacity is the length of the buffer it stores its items in. Then a buffer of length c is allocated at some point if and only if $c \in \kappa$. Then the total length of those buffers is

$$\begin{aligned} s(n) &= \sum_i \kappa_i \\ &= c_0 + gc_0 + g^2 c_0 + \ldots + g^\lambda c_0 \\ &= \left(\frac{g^{\lambda+1} - 1}{g - 1}\right) c_0 \end{aligned}$$

Using Lemma 5.2 again, I asymptotically bound $s(n)$:

$$\begin{gathered} \left(\frac{gn/c_0 - 1}{g-1}\right) c_0 \preceq s(n) \prec \left(\frac{g^2 n/c_0 - 1}{g-1}\right) c_0 \\ \left(\frac{g}{g-1}\right) n - \left(\frac{c_0}{g-1}\right) \preceq s(n) \prec \left(\frac{g^2}{g-1}\right) n - \left(\frac{c_0}{g-1}\right) \\ \left(\frac{g}{g-1}\right) n \preceq s(n) \prec \left(\frac{g^2}{g-1}\right) n \end{gathered}$$

Growth array implementation

In this section, L is a growth array. The following algorithm implements appending for growth arrays.

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

procedure Grow(L)
    L.Tail.Append(L.Head)
    if L.Hcap = c_0 then
        new_hcap ← (g − 1) × L.Hcap
    else
        new_hcap ← g × L.Hcap
    L.Head ← New_array(new_hcap)
    L.Cap ← L.Cap + new_hcap
```

Time complexity

I start off again by finding the write cost for n items. Lemma 5.1 still holds, since growth arrays satisfy the properties used by that proof. In particular, although growth arrays use a different growth algorithm than dynamic arrays, the following claim is still true:

Lemma 5.3. *The capacity of a growth array grows by the constant factor g.*

Proof. I prove that the *Grow* algorithm enforces this using induction. I induct on the number of times *Grow* is called, k, showing that for all natural numbers k, *Grow* behaves correctly when called the kth time. I will let c_i and c_f denote the initial/final capacities and h_i and h_f denote the initial/final head capacities for the kth call, respectively.

For $k = 1$, $c_i = c_0$. I wish to show that $c_f = gc_0$. This happens if and only if the next buffer has size $\Delta c = (g - 1)c_0$, which the algorithm ensures.

For $k > 1$, by induction $c_i =$ previous $c_f = g^{k-2}c_0$, and $h_i =$ previous $h_f = (g^{k-2} - g^{k-3})c_0$. I wish to show $c_f = g^{k-1}c_0$ and $h_f = (g^{k-1} - g^{k-2})c_0$. Because $k > 1$, the algorithm will calculate h_f as g times h_i. Then

$$h_f = gh_i = g(g^{k-2} - g^{k-3})c_0 = (g^{k-1} - g^{k-2})c_0$$

and

$$c_f = c_i + h_f = g^{k-2}c_0 + (g^{k-1} - g^{k-2})c_0 = g^{k-1}c_0$$

as desired. □

Since Lemma 5.3 has been proven, Lemma 5.1 and all results based on it must also hold true for growth arrays. Now, I am ready to find the write cost of *Grow*. Unlike dynamic arrays, *Grow* does not make γ_i writes when the current size is γ_i. In fact, *Grow* does not copy *any* items supplied by the user. Writes are only made when a buffer is appended to the tail, since the tail is a dynamic array.

Let $w_\gamma(n)$ denote the total number of writes made by *Grow*, and let n_τ be the size of the tail. Since Corollary 5.1.1 also holds true for growth arrays, *Grow* is called $|\gamma|$ times. A buffer is appended to the tail each time *Grow* is called. Thus, the tail's size is

$$n_\tau = |\gamma| = |\kappa| - 1 = \lambda$$

Then the formula for $w_\gamma(n)$ is simply $w_\tau(\lambda)$, where w_τ denotes the tail's write cost function, that is, the write cost function for dynamic arrays. Finally, adding the writes made by *Append*, the formula for $w(n)$ is

$$w(n) = n + w_\tau(\lambda)$$

Now, I approximate $w(n)$ using $\sim$. To do this, I will derive the big-O complexity of λ.

Lemma 5.4. $\lambda = O(\log n)$.

Proof. From 5.1, $\lambda = \max(\lceil \log_g n - \log_g c_0 \rceil, 0)$. As mentioned in Lemma 5.2, for sufficiently large n, $\lambda = \lceil \log_g n - \log_g c_0 \rceil$. Then

$$\lambda \sim \lceil \log_g n - \log_g c_0 \rceil \sim \left(\log_g n - \log_g c_0 \right) \sim \log_g n$$

By [lemma], $O(\lambda) = O(\log_g n) = O(\log n)$ as desired. $\square$

Now when $w(n)$ is approximated with $\sim$, the $w_\tau(\lambda)$ term disappears:

$$w(n) = n + w_\tau(\lambda) = n + O(\lambda) = n + O(\log n) \sim n$$

The last step is true because of Theorem 4.2.

Space complexity

Unlike dynamic arrays, growth arrays never throw away buffers. This means that if the current capacity is c, then the total length of buffers allocated to store items is also c. Typically, however, $s(n) > c$. This is because growth arrays not only store items in buffers, they also store **references** (or **pointers**) to the buffers holding the items, in the tail. Thus, the space the tail allocates must also be considered.

We established in [lemma] that $n_\tau = \lambda$. Then since κ_λ is the space needed to hold items, and $s_\tau(\lambda)$ is the space the tail needs to hold references, the formula for $s(n)$ is

$$s(n) = \kappa_\lambda + s_\tau(\lambda)$$

I now wish to approximate this using $\sim$. First, note that since $\kappa_\lambda = g^\lambda c_0$, it follows from Lemma 5.2 that $n \preceq \kappa_\lambda \prec gn$. Since n and gn are both $O(n)$, [the squeeze theorem] implies that $\kappa_\lambda = O(n)$.

Now, consider the ratio $\lim_{n\to\infty} \dfrac{s_\tau(\lambda)}{\kappa_\lambda} = \dfrac{O(\log n)}{O(n)} = O\left(\dfrac{\log n}{n}\right) = 0$. Because it is 0, from Theorem 4.2 one can conclude

$$s(n) = \kappa_\lambda + s_\tau(\lambda) \sim \kappa_\lambda$$

Substituting $s(n)$ for κ_λ in the previous inequality,

$$n \preceq s(n) \prec gn$$

Time complexity comparison

Let $w_D(n)$ and $w_G(n)$ be the write costs for dynamic and growth arrays, respectively. I derived earlier that

$$\left(\frac{g}{g-1}\right) n \preceq w_D(n) \prec \left(\frac{2g-1}{g-1}\right) n$$

Dividing all sides of this inequality by n, I receive

$$\frac{g}{g-1} \preceq \frac{w_D(n)}{n} \prec \frac{2g-1}{g-1}$$

Since $w_G(n) \sim n$, $\dfrac{w_D(n)}{n} \sim \dfrac{w_D(n)}{w_G(n)}$, thus

$$\frac{g}{g-1} \preceq \frac{w_D(n)}{w_G(n)} \prec \frac{2g-1}{g-1}$$

Space complexity comparison

Let $s_D(n)$ and $s_G(n)$ be the space costs for dynamic and growth arrays, respectively. Recall that

$$s_D(n) = \left(\frac{g^{\lambda+1}-1}{g-1}\right) c_0$$

and $s_G(n) \sim \kappa_\lambda = g^\lambda c_0$. Taking the ratio of $s_D(n)$ to $s_G(n)$, I receive

$$\begin{aligned}
\frac{s_D(n)}{s_G(n)} &\sim \left(\frac{1}{g^\lambda}\right)\left(\frac{g^{\lambda+1}-1}{g-1}\right) \\
&= \frac{g - 1/g^\lambda}{g-1} \\
&\sim \frac{g}{g-1}
\end{aligned}$$

The last step comes from the fact that $\lambda = \max(\lceil \log_g n - \log_g c_0 \rceil, 0) \to \infty$ as $n \to \infty$.

5.2 Indexing

Indexing, or accessing the ith element of a collection, is another common dynamic array operation. Indexing a growth array takes more instructions than indexing a dynamic array. Depending on how it is implemented, indexing for growth arrays runs in either constant or logarithmic time.

Dynamic array implementation

The following algorithm implements *Get_item* for dynamic arrays. A *Set_item* function can be implemented in a similar fashion.

Note: I assume all arguments passed to my algorithms are valid. Thus, I do not include argument validation nor error handling in my pseudocode.

```
function Get_item(L, index)
    return L.Buf[index]
```

Clearly, this function runs in constant time. It also has the benefit of using very few instructions: on architectures like x86, indexing the buffer may take as little as one instruction.

Growth array implementation

Indexing a growth array is slower and more complex than indexing a dynamic array. However, this does not mean that accessing data of a growth array has to be slower than accessing that of a dynamic array. Refer to Sections [Iterating] and [Copying to an array] to learn how to quickly access growth arrays' data.

Despite growth arrays' relatively poor indexing performance, I include this section for two reasons. 1) Dynamic arrays are indexed frequently. In order for growth arrays to be a viable replacement for them, it should be possible to index growth arrays too. 2) I wish to show that it is possible to index growth arrays in constant time.

$O(\log n)$ **implementation**

The first algorithm I will demonstrate is a naïve implementation that runs in logarithmic time:

```
function Get_item(L, index)
    i ← index
    for buf in L.Tail do
        if i < buf.Len then
            return buf[i]
        i ← i − buf.Len
    return L.Head[i]
```

I know that this algorithm runs in $O(\log n)$ time since it was shown earlier that the size of the tail, n_τ, equals λ, and (from Lemma 5.4) that $\lambda = O(\log n)$. Aside from iteration over the tail, all statements run in constant time.

I present this algorithm alongside the constant time one because it is easier to understand, and the latter is often slower if special circumstances are not met.

$O(1)$ implementation

In this section, I will present a constant time algorithm for indexing a growth array. Before I do so, however, I must establish a mathematical justification for it.

Suppose I want to get the ith element of a growth array, where i is zero-based. I assume that i is a valid index, or that $0 \le i < L.Size$. In order to locate the desired element, I must find two things: the buffer that holds the item, and the index of the item within that buffer.

Now, consider that every buffer except the head is stored inside the tail. Thus, such buffers can be uniquely identified by their index in the tail. I will refer to this quantity as the **buffer index** and denote it i_B. I wish to assign the head a buffer index as well, so that each buffer has a unique ID. Since the head succeeds the tail's last buffer, which has $i_B = n_\tau - 1$, I will let the head's i_B be n_τ.

I define a helper function, Get_buf, that gets the buffer associated with a given i_B. i_B is assumed to be valid; that is, $0 \le i_B \le n_\tau$.

---

function $Get\_buf(L,\ i_B)$
    if $i_B < L.Tail.Size$ **then**
        return $L.Tail[i_B]$
    else
        return $L.Head$

I will call the index of the desired element within the buffer the **element index**, and denote it i_E.

If I define a helper function $Decompose$ that returns the i_B and i_E associated with i in an ordered pair, then Get_item may be written as follows:

---

function $Get\_item(L,\ index)$
    $(i_B,\ i_E) \leftarrow Decompose(index)$
    return $L.Get\_buf(i_B)[i_E]$

The task is now to find formulae for i_B and i_E in terms of i, in order to implement $Decompose$.

Lemma 5.5. *The formulae for i_B and i_E are*

$$i_B = \lambda|_{n=i+1}$$
$$i_E = i - \gamma_{i_B - 1}$$

Proof. Appending new items does not change the index of an item that is already in the list. Thus, this problem can be reduced to finding the last element when $n = i + 1$.

The last element always resides in the head, so $i_B = n_\tau|_{n=i+1}$. As shown earlier, $n_\tau = \lambda$, so $i_B = \lambda|_{n=i+1}$.

To find i_E, consider the equation:

$$n = \#\text{ of items in tail} + \#\text{ of items in head}$$

From the premise, $n = i + 1$. The number of items in the tail is the last size at which *Grow* was called, or $\gamma_{\lambda-1} = \gamma_{i_B-1} =$ the last term of γ. Finally, since the desired item is the last item in the growth array, it is also the last item in the head. Thus i_E is the head's last valid index, so the size of the head is $i_E + 1$. Substituting these values into the above equation, I receive

$$\begin{aligned} i + 1 &= \gamma_{i_B-1} + i_E + 1 \\ i_E &= i - \gamma_{i_B-1} \end{aligned}$$

completing the proof. □

Now, the *Decompose* function can be implemented as follows:

---

function *Decompose*(*index*)
  $i_B \leftarrow \max(\lceil \log_g(index + 1) - \log_g c_0 \rceil, 0)$
  $i_E \leftarrow index - g^{i_B-1} \times c_0$
  return $(i_B,\ i_E)$

Clearly, *Decompose* runs in constant time. Despite that, it still appears to be quite expensive: normally, logarithms and exponentiation utilize costly floating-point instructions. However, in the special case where $g = 2$ and $c_0 = 2^\varepsilon$ for some constant whole number ε, I claim that i_B and i_E can be computed without use of floating-point instructions.

To see this, first note that i_B becomes $\max(\lceil \log_2(i + 1)\rceil - \varepsilon, 0)$. There is a constant time algorithm for computing $\lceil \log_2 k\rceil$ for any $k \in \mathbb{N}$, which uses bitwise operations instead of floating-point instructions. (I will not discuss it in this paper due to the length its mathematical justification would require, but the algorithm may be found in [Implementations section].) Thus $\lceil \log_2(i + 1)\rceil$ can be computed without floating-point instructions. It is easy to see that the rest of the expression for i_B can be computed without them, too, proving the claim for i_B.

The expression for i_E becomes $i - 2^{i_B+\varepsilon-1}$. Since i_B has already been found, it is trivial to compute $i_B + \varepsilon - 1$. 2 raised to this power can be computed cheaply with a bit shift. Finally, subtracting from i does not use floating-point instructions, proving the claim for i_E.

5.3 Iterating

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

5.4 Copying to an array

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 Proving $\sim$ Properties

A.1 $\sim$ Merges over $+$, $\times$, and $\div$

Proof (Theorem 4.1). The statement about multiplication can be shown easily by multiplying the limits corresponding to f and g:

$$\begin{aligned}
\lim_{n\to\infty}\frac{f}{f_0} &= 1\\
\lim_{n\to\infty}\frac{g}{g_0} &= 1\\
\left(\lim_{n\to\infty}\frac{f}{f_0}\right)\left(\lim_{n\to\infty}\frac{g}{g_0}\right) &= 1\cdot 1\\
\lim_{n\to\infty}\frac{fg}{f_0g_0} &= 1
\end{aligned}$$

it follows that $fg \sim f_0g_0$. The statement about division can be proved by flipping the limit for g before multiplying, resulting in

$$\lim_{n\to\infty}\frac{f/g}{f_0/g_0} = 1$$

This shows $\dfrac{f}{g} \sim \dfrac{f_0}{g_0}$. □

A.2 $\sim$ Removes Lower-Order Terms

Proof (Theorem 4.2). Since

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

it follows that $f + g \sim f$. □

A.3 $\sim$ is Transitive

Proof (Theorem 4.3). By definition, $\lim_{n\to\infty} \frac{f}{g} = 1$ and $\lim_{n\to\infty} \frac{g}{h} = 1$. Multiplying the two equations, $\lim_{n\to\infty} \frac{f}{h} = 1$ which implies $f \sim h$. $\square$