Abstract

1 Introduction

2 Fields and Properties

I define some **fields** and **properties** (constant time methods that do not change state) for both types of arrays, so I may use them later to implement non-trivial methods.

3 Common Operations

In this section, I implement and analyze common operations for dynamic and funnel arrays. I compare both implementations' time complexities, and space complexities if the operation allocates memory.

The following mathematical definitions will be used while analyzing time and space complexity:

$P(f(n))$ This is an alternative to big-O notation that I will name "big-P notation". It is similar to big-O, but it preserves the coefficient of the fastest-growing term in $f(n)$. For example, $O(2n) = O(n)$, but $P(2n) \neq P(n)$. This makes it possible to compare two time complexities, if their ratio tends to a constant for large n.
More formally, $P(f(n)) = P(g(n))$ iff

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

.

3.1 Adding

Description

Adding is the most common operation done on dynamic arrays[1]. Funnel arrays improve the performance of adding in two ways: by allocating less mem-

[1]

ory, and reducing the amount of copying. We begin with the implementation for dynamic arrays.

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

initial capacity The capacity of a dynamic a array when one item is added to it. I denote this C_f.
(The capacity of any dynamic array with size zero should be zero.)

growth factor The factor by which the current capacity is multiplied to get the new capacity when L is resized. I denote this with G.

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

In the above code, one write is performed after the conditional, so it is apparent that $W(n) \geq n$ after n adds. However, [resize] also does some writing, so in order to find a precise formula for $W(n)$ I need to analyze when [resize] is called. Consider the following lemma:

[lemma] Let L by a dynamic array. As n are added, the sequence of capacities L takes on is

$$S_C = 0,\ C_f,\ GC_f,\ G^2C_f,\ \ldots\ G^{\lceil \log_G n - \log_G C_f \rceil} C_f$$

.

[proof] The capacity starts out at zero and becomes C_f once one element is added. From then on, it can only grow by a factor of G via [resize], so if G^iC_f is the current capacity then $G^{i+1}C_f$ must be the next capacity. By induction, the rest of the sequence is

$$\left\{G^iC_f\right\}_{i=1}^{k}$$

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 $G^kC_f \geq n$. Using this property, we now derive k.

$$
\begin{aligned}
G^k C_f &\geq n \\
G^k &\geq \frac{n}{C_f} \\
k &\geq \log_G n - \log_G C_f \\
k &\geq \log_G n - \log_G C_f > k - 1 \\
k &= \left\lceil \log_G n - \log_G C_f \right\rceil
\end{aligned}
$$

Then the final term in the sequence is $G^{\lceil \log_G n - \log_G C_f \rceil} C_f$, completing the proof.

Now, consider that to grow by G we have to call [resize]. So $G^k C_f$ being present in S_C means we once called [resize] at size $G^{k-1} C_f$ for $k \geq 1$. Then we must have called [resize] at the sizes

$$S_R = 0,\ C_f,\ GC_f,\ G^2 C_f,\ \ldots\ G^{\lceil \log_G n - \log_G C_f \rceil - 1} C_f$$

Each time [resize] is called, we copy i items where i is the current size of the dynamic array. Then the total number of items copied for n adds is

$$0 + C_f + GC_f + \ldots + G^{\lceil \log_G n - \log_G C_f \rceil - 1} C_f = \left(\frac{G^{\lceil \log_G n - \log_G C_f \rceil} - 1}{G - 1} \right) C_f$$

Finally, adding the writes made for every item in [add],

$$W(n) = n + \left(\frac{G^{\lceil \log_G n - \log_G C_f \rceil} - 1}{G - 1} \right) C_f$$

$$W(n) = P(2n)$$

Space complexity

$$S(n) = P(2^{\lceil \log_2 n \rceil + 1}) = O(n)$$

Funnel array implementation

Time complexity

$$W'(n) = P(n)$$

Space complexity

$$S'(n) = P(2^{\lceil \log_2 n \rceil}) = O(n)$$

Time complexity comparison

Space complexity comparison

$$r_S = \frac{P(2^{\lceil \log_2 n \rceil})}{P(2^{\lceil \log_2 n \rceil + 1})} = \frac{1}{2}$$

3.2 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 indexer** and *set indexers*, respectively.

Dynamic array implementation

Time complexity

$T(n) = O(1)$

Time complexity

$T'(n) = O(1)$

Time complexity comparison

3.3 Iterating

Description

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

Dynamic array implementation

Time complexity

Ignoring the arbitrary code run after getting each element, the time complexity of this method is $O(n)$.

Funnel array implementation

Time complexity

$T'(n) = O(n)$

Time complexity comparison

3.4 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 adding to the list:

- Plain arrays hold on to exactly the amount of memory needed to hold their elements. However, dynamic and funnel arrays allocate more space than necessary to optimize adding 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 funnel arrays are fragmented and have worse locality.
- The indexer of funnel arrays is several times slower than that of plain arrays, whether the $O(1)$ or $O(\log n)$ implementation is chosen.

Dynamic array implementation

Time complexity

Funnel array implementation

Time complexity

4 Other Operations

4.1 Inserting

Description

Inserting an element places it at a specified index within the list, and increments the list's count. If the index equals the size of the list, the effect is the same as adding the element. Otherwise, the elements at indices greater than or equal to the specified index are moved to the next index, then the element is placed at the specified index.

4.2 Deleting

Description

Deleting the element at a specified index moves the elements at indices greater than the specified index to the previous index. The count of the list is decremented.

4.3 Searching

Description

Searching for an element returns the first or last index within the list where the item can be found. If the user knows the items of the lists are sorted, **binary search** can be used.

4.4 In-place sorting

Description

Given a strict ordering $<$, we say a list L is **sorted** by $<$ iff $(a, b \in L \wedge a < b) \leftrightarrow$

$I_M(a) < I_m(b)$. $I_M(a)$ is the last (maximum) index of a in L, and $I_m(b)$ is the first (minimum) index of b in L. Note that the use of $<$ on the right-hand side compares integers and not elements of L, so this is not a recursive definition.

5 Implementations

6 Benchmarks

7 Closing Remarks