Abstract

1 Introduction

2 Definitions

decrement, increment Decreasing or increasing (respectively) the value of an integer variable by one.

field A variable associated with an object that may be read from or written to. It is assumed that fields can be read from and written to in constant time.

get indexer, set indexer A method that gets or sets (respectively) an item at a specified index in a list.

indexer Refers to a get indexer or set indexer.

property A function of an object that returns a value, runs in constant time, and does not mutate any state (i.e. by assigning new values to fields).

trivial member A field or property of an object.

3 Runtime-Defined Functions

4 Trivial Members of Dynamic and Funnel Arrays

In this section, I define **fields** and **properties** on dynamic and funnel arrays. These will be used later to implement operations on them in pseudocode.

5 Common Funnel Array Operations

In this section, I implement common operations for dynamic and funnel arrays in pseudocode. Then, I analyze and compare the time complexities of the implementations. If the operation allocates memory, I also compare their space complexities.

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

$P(f(n))$ This is an alternative to big-O notation for time complexity; I will call it **big-P notation**.

The main purpose of big-P notation is to preserve the constant coefficient of the fastest-growing term in $f(n)$, while still maintaining many properties of big-O. For example, $O(2n) = O(n)$, but $P(2n) \neq P(n)$. This makes it possible to compare time complexities whose ratio converges to a constant as n becomes larger.

More formally, $P(f(n)) = P(g(n))$ iff

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

.

5.1 Adding

Description

Adding is the most common operation done on a dynamic array[1]. Funnel arrays improve the performance of adding in two ways: by allocating less memory, and reducing the amount of copying.

Dynamic array implementation

Time complexity

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

Space complexity

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

Funnel array implementation

Time complexity

[1]

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

Space complexity

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

Time complexity comparison

Space complexity comparison

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

5.2 Indexing

Description

Indexing is another very common operation on a list. An **indexer** gets or sets an item at a specified index in a list.

Dynamic array implementation

Time complexity

$T(n) = O(1)$

Time complexity

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

Time complexity comparison

5.3 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

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 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

6 Other Funnel Array Operations

6.1 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.

6.2 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.

6.3 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.

6.4 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.