Abstract

This paper introduces the *fragmented list* data structure, which implements the list abstract data type. A fragmented list consists of multiple arrays, or *buffers*, that store its elements. Fragmented lists do not make copies or throw away old buffers while resizing. As a result, they perform significantly better than dynamic arrays when a large number of elements are added to them. However, this is provided the final size of the list (or a close upper bound on it) is not known beforehand. If it is, a buffer of that size can be pre-allocated and resizing can be avoided, making fragmented lists redundant.

1 Introduction

Often, node-based structures receive the most attention in a course on data structures. Although it is simpler to implement everything with arrays[1], node-based structures are favored because they have better time complexity for operations such as insertion and deletion.

However, arrays have their own performance advantages over such structures, including:

- Better locality when iterating: elements of an array live next to each other in memory.
- Less memory overhead: memory does not need to be allocated for nodes.
- Fast random access: since arrays are just pointers, indexing them takes a few instructions and runs in $O(1)$ time.

When an abstract data type (which I will refer to as *data type*) can be implemented easily with arrays, they are often used. One such data type is the *list*[2], which is usually implemented with a dynamic array.

Dynamic arrays perform reasonably well with respect to their most common operations. Adding an item runs in amortized $O(1)$ time, indexing them runs in $O(1)$ time, and they can be iterated in $O(n)$ time.

However, when a dynamic array is *full* (that is, its size equals its capacity) and an item is added, it must resize its buffer to fit the new item. Its behavior here is less than optimal: it allocates a new buffer larger than the current one, copies over the items, and discards the current buffer.

Instead of throwing away buffers once they're filled up, *fragmented lists* store references to them in a two-dimensional array. During a resize, a new buffer is

[1]

[2]

still allocated; however, for a sufficiently large number of items, it is always half the size of the buffer that a dynamic array would allocate.

In addition, no items are copied to the front of the new buffer. This is because no buffer represents the entire content of the list; each buffer holds just a *fragment* of the items. To get back the items that were added to the fragmented list, one needs to traverse each of the buffers and copy their share of the items.

2 Definitions

2.1 Terminology

buffer An array that holds a portion of the items in a fragmented list.
capacity The maximum amount of items a list can hold without resizing.
full, filled A list is full iff its size equals its capacity, meaning it must resize if another item is added.
growth factor The factor by which the capacity grows when the list is resized, if the current capacity is nonzero.
list The list data type. This term does not refer to a particular implementation of the data type, only the "interface."
resize To increase a list's capacity once it is full.
threshold sizes The set of sizes for which the list is full.

2.2 Runtime-Provided Functions

I assume the following functions are provided by the runtime environment, so that I may use them without defining them beforehand.

▷ Adds *item* to dynamic array d
$Add\ dyn_d(item)$

▷ Allocates and returns an array of length *len*
$Array(len)$

▷ Allocates and returns an empty dynamic array
$Dynamic()$

▷ Returns the length of array a
Len_a

2.3 Mathematical Notation

$T_m(n)$ Time it takes to allocate memory for an array of size n

3 Fragmented List Structure

Each fragmented list, denoted by L, is given four *fields*: $Head_L$, $Tail_L$, $Size_L$, and Cap_L . These fields may be read from and assigned to.

- $Head_L$ is the *head* of the fragmented list. It returns the buffer we are currently adding items to.
- $Tail_L$ is the *tail* of the list. It returns a list of buffers that have already been filled.
- $Size_L$ is the list's *size*. It returns the number of items in L.
- Cap_L is the list's *capacity*. It returns the maximum number of items L can hold without resizing.

Below, I also define some auxiliary functions on L (which may not be assigned to). := denotes a definition, as opposed to = which checks equivalence. Functions that return boolean values are suffixed with ?.

▷ Returns whether L is empty
$Empty?_L := Size_L = 0$

▷ Returns whether L is full
$Full?_L := Size_L = Cap_L$

▷ Returns the capacity of $Head_L$
$Hcap_L := Len_{Head_L}$

▷ Returns the size of $Head_L$
$Hsize_L := Size_L - (Cap_L - Hcap_L)$

Before we call any functions with L, we need to make sure that it is properly initialized. (In an object-oriented language, this code would go in the constructor.)

$Head_L \leftarrow Array(0)$
$Tail_L \leftarrow Dynamic()$

---

4 Fragmented List Operations

4.1 Adding an Element

I begin with the implementation for *Add* since it is the most common list operation[3]. Its algorithm is very similar to that of a dynamic array: it resizes the list if it's full, then stores the item and increments the size. The head's size is also incremented since an item is added to it.

procedure $Add_L(item)$
  if $Full?_L$ **then**
    $Resize_L()$
  $Head_L[Hsize_L] \leftarrow item$
  $Size_L \leftarrow Size_L + 1$
  $Hsize_L \leftarrow Hsize_L + 1$

A fragmented list is resized quite a bit differently from a dynamic array, however.

procedure $Resize_L$
Require:
  $Full?_L$
  if $Empty?_L$ **then**
    $Head_L \leftarrow Array(4)$
    $Cap_L \leftarrow 4$
    return
  $Add\ dyn_{Tail_L}(Head_L)$
  $new\ cap \leftarrow G(Hcap_L)$
  $Head_L \leftarrow Array(new\ cap)$
  $Hsize_L \leftarrow 0$
  $Cap_L \leftarrow Cap_L + new\ cap$

▷ G is the *growth function*.
▷ It decides the capacity of the next buffer based on the current buffer's capacity.
function $G(cap)$
Require:
  $cap \geq 4 \wedge \log_2 cap \in \mathbb{N}$
  return $\begin{cases} 4 & cap = 4 \\ 2 * cap & cap \neq 4 \end{cases}$

[3]

4.1.1 Time Analysis

In each "Time Analysis" section, I derive not just the time complexity of the operation, but a complete formula that approximates the actual (amortized) amount of time it takes. In order to do this in a clean fashion, I use the following notation:

$T\big[F\big]$ The amortized amount of time it takes for function F to run.
T_A The amortized amount of time it takes for the code at line to run.
$T_{A,B}$ The amortized amount of time it takes for code between lines A and B to run, inclusive.
$T(k)$ The amount of time it takes for T to run, where T is any time expression, when $Size_L = k$.

Since the first three definitions represent *amortized* time, which is the average amount of time it takes for an operation to run, I can expand them to

$$T\big[F\big] = \frac{\sum_{i=0}^{n-1} F(i)}{n}$$
$$T_A = \frac{\sum_{i=0}^{n-1} T_A(i)}{n}$$
$$T_{A,B} = \frac{\sum_{i=0}^{n-1} T_{A,B}(i)}{n}$$

These definitions will be used frequently when a piece of code takes longer to run for some sizes than others.

Now, I begin analyzing the amortized amount of time for Add_L . The expression we want to find is $T\big[Add_L\big]$. In order to do this, I will need a few lemmas:

Lemma. Let $Caps_L$ be the set of all values assigned to Cap_L as $Size_L$ increases indefinitely. Let $Caps_L(n)$ be the subset of $Caps_L$ less than n. I show that $capacities(n)$ is the set

$$Caps_L(n) = \{0\} \cup \{2^i | 1 < i \leq lgc(n)\}$$

where $lgc(n) = \lfloor \log_2 (n-1) \rfloor$, the base 2 *logarithm of the greatest capacity* less than n.

We define $\overline{Caps_L(n)}$ as $\{0, 1, ...n-1\} \backslash Caps_L(n)$.

$$
\begin{aligned}
T\left[Add_L\right] &= T_{2,6} = T_2 + T_3 + T_{4,6} \\
&= T_3 + C \\
&= \frac{\sum_{i=0}^{n} T_3(i)}{n} \\
&= \frac{\sum_{i \in Caps_L(n)} T_3(i) + \sum_{i \in \overline{Caps_L(n)}} T_3(i)}{n}
\end{aligned}
$$

Since $|Caps_L(n)| = lgc(n) - 1$ and $|\overline{Caps_L(n)}| = n - |Caps_L(n)| = n + 1 - lgc(n)$, this resolves to

$$content...$$

When L is not full:

$$T_{2,3} + T_{4,6} = T\left[full?\right] + T_{4,6} = C$$

When L is full:

$$T_{2,3} + T_{4,6} = T\left[full?\right] + T_[resize] + T_{4,6} = T[resize] + C$$

Now we must analyze $resize$.

When L is empty:

$$T[resize] = T_{2,5} = T[empty?] + T_alloc(4) + T_5 = T_alloc(4) + C$$

When L is non-empty:

$$T[resize] = T[empty?] +$$

4.1.2 Space Analysis

4.2 Indexing

A function to set an item at a particular index can be implemented in a similar fashion.

function $item\ at\ index(L, index)$
Require:
$0 \leq index < Size_L$
$i \leftarrow index$
for all $buf \in Tail_L$ **do**
if $i < Len_{buf}$ **then**
return $buf[i]$
$i \leftarrow i - Len_{buf}$
return $Head_L[i]$

4.3 Inserting an Element

4.4 Deleting the Last Element

4.5 Deleting an Element

4.6 Copying to an Array

4.7 Iterating

4.8 Searching

4.9 In-Place Sorting

5 Implementations

6 Benchmarks

7 Closing Remarks