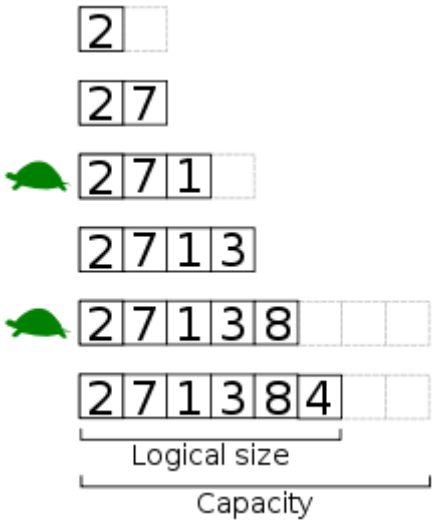
Dynamic array

In computer science, a **dynamic array**, **growable array**, **resizable array**, **dynamic table**, **mutable array**, or **array list** is a random access, variable-size list data structure that allows elements to be added or removed. It is supplied with standard libraries in many modern mainstream programming languages. Dynamic arrays overcome a limit of static arrays, which have a fixed capacity that needs to be specified at allocation.

A dynamic array is not the same thing as a dynamically allocated array, which is an array whose size is fixed when the array is allocated, although a dynamic array may use such a fixed-size array as a back end.[1]



Several values are inserted at the end of a dynamic array using geometric expansion. Grey cells indicate space reserved for expansion. Most insertions are fast (constant time), while some are slow due to the need for reallocation ($\Theta(n)$ time, labelled with turtles). The *logical size* and *capacity* of the final array are shown.

Contents

Bounded-size dynamic arrays and capacity

A simple dynamic array can be constructed by allocating an array of fixed-size, typically larger than the number of elements immediately required. The elements of the dynamic array are stored contiguously at the start of the underlying array, and the remaining positions towards the end of the underlying array are reserved, or unused. Elements can be added at the end of a dynamic array in constant time by using the reserved space, until this space is completely consumed. When all space is consumed, and an additional element is to be added, then the underlying fixed-sized array needs to be increased in size. Typically resizing is expensive because it involves allocating a new underlying array and copying each element from the original array. Elements can be removed from the end of a dynamic array in constant time, as no resizing is required. The number of elements used by the dynamic array contents is its *logical size* or *size*, while the size of the underlying array is called the dynamic array's *capacity* or *physical size*, which is the maximum possible size without relocating data.[2]

A fixed-size array will suffice in applications where the maximum logical size is fixed (e.g. by specification), or can be calculated before the array is allocated. A dynamic array might be preferred if:

- the maximum logical size is unknown, or difficult to calculate, before the array is allocated

- it is considered that a maximum logical size given by a specification is likely to change
- the amortized cost of resizing a dynamic array does not significantly affect performance or responsiveness

Geometric expansion and amortized cost

To avoid incurring the cost of resizing many times, dynamic arrays resize by a large amount, such as doubling in size, and use the reserved space for future expansion. The operation of adding an element to the end might work as follows:

```
function insertEnd(dynarray a, element e)
    if (a.size == a.capacity)
        // resize a to twice its current capacity:
        a.capacity ← a.capacity * 2
        // (copy the contents to the new memory location here)
    a[a.size] ← e
    a.size ← a.size + 1
```

As *n* elements are inserted, the capacities form a geometric progression. Expanding the array by any constant proportion *a* ensures that inserting *n* elements takes $O(n)$ time overall, meaning that each insertion takes amortized constant time. Many dynamic arrays also deallocate some of the underlying storage if its size drops below a certain threshold, such as 30% of the capacity. This threshold must be strictly smaller than $1/a$ in order to provide hysteresis (provide a stable band to avoid repeatedly growing and shrinking) and support mixed sequences of insertions and removals with amortized constant cost.

Dynamic arrays are a common example when teaching amortized analysis.[3][4]

Growth factor

The growth factor for the dynamic array depends on several factors including a space-time trade-off and algorithms used in the memory allocator itself. For growth factor *a*, the average time per insertion operation is about $a/(a-1)$, while the number of wasted cells is bounded above by $(a-1)n$. If memory allocator uses a first-fit allocation algorithm, then growth factor values such as $a=2$ can cause dynamic array expansion to run out of memory even though a significant amount of memory may still be available.[5] There have been various discussions on ideal growth factor values, including proposals for the golden ratio as well as the value 1.5.[6] Many textbooks, however, use $a = 2$ for simplicity and analysis purposes.[3][4]

Below are growth factors used by several popular implementations:

Implementation	Growth factor (*a*)
Java ArrayList[1]	1.5 (3/2)
Python PyListObject[7]	~1.125 (n + n >> 3)
Microsoft Visual C++ 2013[8]	1.5 (3/2)
G++ 5.2.0[5]	2
Clang 3.6[5]	2
Facebook folly/FBVector[9]	1.5 (3/2)
Rust Vec[10]	2

Performance

Comparison of list data structures

	Linked list	Array	Dynamic array	Balanced tree	Random access list	Hashed array tree
Indexing	Θ(*n*)	Θ(1)	Θ(1)	Θ(log n)	Θ(log n)[11]	Θ(1)
Insert/delete at beginning	Θ(1)	N/A	Θ(*n*)	Θ(log n)	Θ(1)	Θ(*n*)
Insert/delete at end	Θ(1) when last element is known; Θ(*n*) when last element is unknown	N/A	Θ(1) amortized	Θ(log *n*)	N/A [11]	Θ(1) amortized
Insert/delete in middle	search time + Θ(1)[12][13]	N/A	Θ(*n*)	Θ(log *n*)	N/A [11]	Θ(*n*)
Wasted space (average)	Θ(*n*)	0	Θ(*n*)[14]	Θ(*n*)	Θ(*n*)	$\Theta(\sqrt{n})$

The dynamic array has performance similar to an array, with the addition of new operations to add and remove elements:

- Getting or setting the value at a particular index (constant time)
- Iterating over the elements in order (linear time, good cache performance)
- Inserting or deleting an element in the middle of the array (linear time)
- Inserting or deleting an element at the end of the array (constant amortized time)

Dynamic arrays benefit from many of the advantages of arrays, including good locality of reference and data cache utilization, compactness (low memory use), and random access. They usually have only a small fixed additional overhead for storing information about the size and capacity. This makes dynamic arrays an attractive tool for building cache-friendly data structures. However, in languages like Python or Java that enforce reference semantics, the dynamic array generally will not store the actual data, but rather it will store references to the data that resides in other areas of memory. In this case, accessing items in the array sequentially will actually involve accessing multiple non-contiguous areas of memory, so the many advantages of the cache-friendliness of this data structure are lost.

Compared to linked lists, dynamic arrays have faster indexing (constant time versus linear time) and typically faster iteration due to improved locality of reference; however, dynamic arrays require linear time to insert or delete at an arbitrary location, since all following elements must be moved, while linked lists can do this in constant time. This disadvantage is mitigated by the gap buffer and *tiered vector* variants discussed under *Variants* below. Also, in a highly fragmented memory region, it may be expensive or impossible to find contiguous space for a large dynamic array, whereas linked lists do not require the whole data structure to be stored contiguously.

A balanced tree can store a list while providing all operations of both dynamic arrays and linked lists reasonably efficiently, but both insertion at the end and iteration over the list are slower than for a dynamic array, in theory and in practice, due to non-contiguous storage and tree traversal/manipulation overhead.

Variants

Gap buffers are similar to dynamic arrays but allow efficient insertion and deletion operations clustered near the same arbitrary location. Some deque implementations use array deques, which allow amortized constant time insertion/removal at both ends, instead of just one end.

Goodrich[15] presented a dynamic array algorithm called *tiered vectors* that provides $O(n^{1/k})$ performance for insertions and deletions from anywhere in the array, and O(k) get and set, where $k \geq 2$ is a constant parameter.

Hashed array tree (HAT) is a dynamic array algorithm published by Sitarski in 1996.[16] Hashed array tree wastes order $n^{1/2}$ amount of storage space, where n is the number of elements in the array. The algorithm has O(1) amortized performance when appending a series of objects to the end of a hashed array tree.

In a 1999 paper,[17] Brodnik et al. describe a tiered dynamic array data structure, which wastes only $n^{1/2}$ space for *n* elements at any point in time, and they prove a lower bound showing that any dynamic array must waste this much space if the operations are to remain amortized constant time. Additionally, they present a variant where growing and shrinking the buffer has not only amortized but worst-case constant time.

Bagwell (2002)[18] presented the VList algorithm, which can be adapted to implement a dynamic array.

Language support

C++'s `std::vector` and Rust's `std::vec::Vec` are implementations of dynamic arrays, as are the `ArrayList`[19] classes supplied with the Java API and the .NET Framework.[20]

The generic `List<>` class supplied with version 2.0 of the .NET Framework is also implemented with dynamic arrays. Smalltalk's `OrderedCollection` is a dynamic array with dynamic start and end-index, making the removal of the first element also O(1).

Python's `list` datatype implementation is a dynamic array.

Delphi and D implement dynamic arrays at the language's core.

Ada's `Ada.Containers.Vectors` generic package provides dynamic array implementation for a given subtype.

Many scripting languages such as Perl and Ruby offer dynamic arrays as a built-in primitive data type.

Several cross-platform frameworks provide dynamic array implementations for C, including `CFArray` and `CFMutableArray` in Core Foundation, and `GArray` and `GPtrArray` in GLib.

Common Lisp provides a rudimentary support for resizable vectors by allowing to configure the built-in `array` type as *adjustable* and the location of insertion by the *fill-pointer*.

References

1. See, for example, the source code of java.util.ArrayList class from OpenJDK 6 (http://hg.openjdk.java.net/jdk6/jdk6/jdk/file/e0e25ac28560/src/share/classes/java/util/ArrayList.java).
2. Lambert, Kenneth Alfred (2009), "Physical size and logical size" (https://books.google.com/books?id=VtfM3YGW5jYC&q=%22logical+size%22+%22dynamic+array%22&pg=PA518), *Fundamentals of Python: From First Programs Through Data Structures*, Cengage Learning, p. 510, ISBN 978-1423902188

3. Goodrich, Michael T.; Tamassia, Roberto (2002), "1.5.2 Analyzing an Extendable Array Implementation", *Algorithm Design: Foundations, Analysis and Internet Examples*, Wiley, pp. 39–41.
4. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001) [1990]. "17.4 Dynamic tables". *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. pp. 416–424. ISBN 0-262-03293-7.
5. "C++ STL vector: definition, growth factor, member functions" (https://web.archive.org/web/20150806162750/http://www.gahcep.com/cpp-internals-stl-vector-part-1/). Archived from the original (http://www.gahcep.com/cpp-internals-stl-vector-part-1/) on 2015-08-06. Retrieved 2015-08-05.
6. "vector growth factor of 1.5" (https://groups.google.com/forum/#!topic/comp.lang.c++.moderated/asH_VojWKJw%5B1-25%5D). *comp.lang.c++.moderated*. Google Groups.
7. List object implementation (https://github.com/python/cpython/blob/bace59d8b8e38f5c779ff6296ebdc0527f6db14a/Objects/listobject.c#L58) from github.com/python/cpython/, retrieved 2020-03-23.
8. Brais, Hadi. "Dissecting the C++ STL Vector: Part 3 - Capacity & Size" (https://hadibrais.wordpress.com/2013/11/15/dissecting-the-c-stl-vector-part-3-capacity/). *Micromysteries*. Retrieved 2015-08-05.
9. "facebook/folly" (https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md). *GitHub*. Retrieved 2015-08-05.
10. "rust-lang/rust" (https://github.com/rust-lang/rust/blob/fd4b177aabb9749dfb562c48e47379cea81dc277/src/liballoc/raw_vec.rs#L443). *GitHub*. Retrieved 2020-06-09.
11. Chris Okasaki (1995). "Purely Functional Random-Access Lists". *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*: 86–95. doi:10.1145/224164.224187 (https://doi.org/10.1145%2F224164.224187).
12. *Day 1 Keynote - Bjarne Stroustrup: C++11 Style* (http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style) at *GoingNative 2012* on *channel9.msdn.com* from minute 45 or foil 44
13. *Number crunching: Why you should never, ever, EVER use linked-list in your code again* (http://kjellkod.wordpress.com/2012/02/25/why-you-should-never-ever-ever-use-linked-list-in-your-code-again/) at *kjellkod.wordpress.com*
14. Brodnik, Andrej; Carlsson, Svante; Sedgewick, Robert; Munro, JI; Demaine, ED (1999), *Resizable Arrays in Optimal Time and Space (Technical Report CS-99-09)* (http://www.cs.uwaterloo.ca/research/tr/1999/09/CS-99-09.pdf) (PDF), Department of Computer Science, University of Waterloo
15. Goodrich, Michael T.; Kloss II, John G. (1999), "Tiered Vectors: Efficient Dynamic Arrays for Rank-Based Sequences" (https://archive.org/details/algorithmsdatast0000wads/page/205), *Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science, **1663**: 205–216 (https://archive.org/details/algorithmsdatast0000wads/page/205), doi:10.1007/3-540-48447-7_21 (https://doi.org/10.1007%2F3-540-48447-7_21), ISBN 978-3-540-66279-2
16. Sitarski, Edward (September 1996), "HATs: Hashed array trees" (http://www.ddj.com/architect/184409965?pgno=5), Algorithm Alley, *Dr. Dobb's Journal*, **21** (11)
17. Brodnik, Andrej; Carlsson, Svante; Sedgewick, Robert; Munro, JI; Demaine, ED (1999), *Resizable Arrays in Optimal Time and Space* (http://www.cs.uwaterloo.ca/research/tr/1999/09/CS-99-09.pdf) (PDF) (Technical Report CS-99-09), Department of Computer Science, University of Waterloo
18. Bagwell, Phil (2002), *Fast Functional Lists, Hash-Lists, Deques and Variable Length Arrays* (http://citeseer.ist.psu.edu/bagwell02fast.html), EPFL
19. Javadoc on `ArrayList (https://docs.oracle.com/javase/10/docs/api/java/util/ArrayList.html)`
20. ArrayList Class (https://msdn.microsoft.com/en-us/library/system.collections.arraylist)

External links

- NIST Dictionary of Algorithms and Data Structures: Dynamic array (https://xlinux.nist.gov/dads/HTML/dynamicarray.html)
- VPOOL (http://www.bsdua.org/libbsdua.html#vpool) - C language implementation of dynamic array.
- CollectionSpy (https://web.archive.org/web/20090704095801/http://www.collectionspy.com/) — A Java profiler with explicit support for debugging ArrayList- and Vector-related issues.
- Open Data Structures - Chapter 2 - Array-Based Lists (http://opendatastructures.org/versions/edition-0.1e/ods-java/2_Array_Based_Lists.html), Pat Morin