Associative array

In computer science, an **associative array**, **map**, **symbol table**, or **dictionary** is an abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection.

Operations associated with this data type allow:[1][2]

- the addition of a pair to the collection
- the removal of a pair from the collection
- the modification of an existing pair
- the lookup of a value associated with a particular key

The **dictionary problem** is a classic computer science problem: the task of designing a data structure that maintains a set of data during 'search', 'delete', and 'insert' operations.[3] The two major solutions to the dictionary problem are a hash table or a search tree.[1][2][4][5] In some cases it is also possible to solve the problem using directly addressed arrays, binary search trees, or other more specialized structures.

Many programming languages include associative arrays as primitive data types, and they are available in software libraries for many others. Content-addressable memory is a form of direct hardware-level support for associative arrays.

Associative arrays have many applications including such fundamental programming patterns as memoization and the decorator pattern.[6]

Contents

Operations

In an associative array, the association between a key and a value is often known as a "binding", and the same word "binding" may also be used to refer to the process of creating a new association.

The operations that are usually defined for an associative array are:[1][2]

- **Add** or **insert**: add a new $(key, value)$ pair to the collection, binding the new key to its new value. The arguments to this operation are the key and the value.
- **Reassign**: replace the value in one of the $(key, value)$ pairs that are already in the collection, binding an old key to a new value. As with an insertion, the arguments to this operation are the key and the value.
- **Remove** or **delete**: remove a $(key, value)$ pair from the collection, unbinding a given key from its value. The argument to this operation is the key.
- **Lookup**: find the value (if any) that is bound to a given key. The argument to this operation is the key, and the value is returned from the operation. If no value is found, some associative array implementations raise an exception.

Often then instead of add or reassign there is a single **set** operation that adds a new $(key, value)$ pair if one does not already exist, and otherwise reassigns it.

In addition, associative arrays may also include other operations such as determining the number of bindings or constructing an iterator to loop over all the bindings. Usually, for such an operation, the order in which the bindings are returned may be arbitrary.

A multimap generalizes an associative array by allowing multiple values to be associated with a single key.[7] A bidirectional map is a related abstract data type in which the bindings operate in both directions: each value must be associated with a unique key, and a second lookup operation takes a value as argument and looks up the key associated with that value.

Example

Suppose that the set of loans made by a library is represented in a data structure. Each book in a library may be checked out only by a single library patron at a time. However, a single patron may be able to check out multiple books. Therefore, the information about which books are checked out to which patrons may be represented by an associative array, in which the books are the keys and the patrons are the values. Using notation from Python or JSON, the data structure would be:

```
{
    "Pride and Prejudice": "Alice",
    "Wuthering Heights": "Alice",
    "Great Expectations": "John"
}
```

A lookup operation on the key "Great Expectations" would return "John". If John returns his book, that would cause a deletion operation, and if Pat checks out a book, that would cause an insertion operation, leading to a different state:

```
{
    "Pride and Prejudice": "Alice",
    "The Brothers Karamazov": "Pat",
    "Wuthering Heights": "Alice"
}
```

Implementation

For dictionaries with very small numbers of bindings, it may make sense to implement the dictionary using an association list, a linked list of bindings. With this implementation, the time to perform the basic dictionary operations is linear in the total number of bindings; however, it is easy to implement and the constant factors in its running time are small.[1][8]

Another very simple implementation technique, usable when the keys are restricted to a narrow range of integers, is direct addressing into an array: the value for a given key k is stored at the array cell $A[k]$, or if there is no binding for k then the cell stores a special sentinel value that indicates the absence of a binding. As well as being simple, this technique is fast: each dictionary operation takes constant time. However, the space requirement for this structure is the size of the entire keyspace, making it impractical unless the keyspace is small.[4]

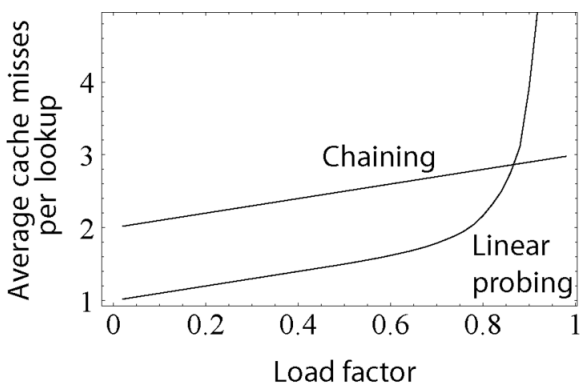
The two major approaches to implementing dictionaries are a hash table or a search tree.[1][2][4][5]

Hash table implementations

The most frequently used general purpose implementation of an associative array is with a hash table: an array combined with a hash function that separates each key into a separate "bucket" of the array. The basic idea behind a hash table is that accessing an element of an array via its index is a simple, constant-time operation. Therefore, the average overhead of an operation for a hash table is only the computation of the key's hash, combined with accessing the corresponding bucket within the array. As such, hash tables usually perform in O(1) time, and outperform alternatives in most situations.

Hash tables need to be able to handle collisions: when the hash function maps two different keys to the same bucket of the array. The two most widespread approaches to this problem are separate chaining and open addressing.[1][2][4][9] In separate chaining, the array does not store the value itself but stores a pointer to another container, usually an association list, that stores all of the values matching the hash. On the other hand, in open addressing, if a hash collision is found, then the table seeks an empty spot in an array to store the value in a deterministic manner, usually by looking at the next immediate position in the array.

Open addressing has a lower cache miss ratio than separate chaining when the table is mostly empty. However, as the table becomes filled with more elements, open addressing's performance degrades exponentially. Additionally, separate chaining uses less memory in most cases, unless the entries are very small (less than four times the size of a pointer).



This graph compares the average number of cache misses required to look up elements in tables with separate chaining and open addressing.

Tree implementations

Self-balancing binary search trees

Another common approach is to implement an associative array with a self-balancing binary search tree, such as an AVL tree or a red-black tree.[10]

Compared to hash tables, these structures have both advantages and weaknesses. The worst-case performance of self-balancing binary search trees is significantly better than that of a hash table, with a time complexity in big O notation of O(log n). This is in contrast to hash tables, whose worst-case performance involves all elements sharing a single bucket, resulting in O(n) time complexity.

In addition, and like all binary search trees, self-balancing binary search trees keep their elements in order. Thus, traversing its elements follows a least-to-greatest pattern, whereas traversing a hash table can result in elements being in seemingly random order. However, hash tables have a much better average-case time complexity than self-balancing binary search trees of O(1), and their worst-case performance is highly unlikely when a good hash function is used.

It is worth noting that a self-balancing binary search tree can be used to implement the buckets for a hash table that uses separate chaining. This allows for average-case constant lookup, but assures a worst-case performance of O(log *n*). However, this introduces extra complexity into the implementation, and may cause even worse performance for smaller hash tables, where the time spent inserting into and balancing the tree is greater than the time needed to perform a linear search on all of the elements of a linked list or similar data structure.[11][12]

Other trees

Associative arrays may also be stored in unbalanced binary search trees or in data structures specialized to a particular type of keys such as radix trees, tries, Judy arrays, or van Emde Boas trees, but these implementation methods are less efficient than hash tables as well as placing greater restrictions on the types of data that they can handle. The advantages of these alternative structures come from their ability to handle operations beyond the basic ones of an associative array, such as finding the binding whose key is the closest to a queried key, when the query is not itself present in the set of bindings.

Comparison

Underlying data structure	Lookup		Insertion		Deletion		Ordered
	average	worst case	average	worst case	average	worst case	
Hash table	O(1)	O(*n*)	O(1)	O(*n*)	O(1)	O(*n*)	No
Self-balancing binary search tree	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)	Yes
unbalanced binary search tree	O(log *n*)	O(*n*)	O(log *n*)	O(*n*)	O(log *n*)	O(*n*)	Yes
Sequential container of key-value pairs (e.g. association list)	O(*n*)	O(*n*)	O(1)	O(1)	O(*n*)	O(*n*)	No

Language support

Associative arrays can be implemented in any programming language as a package and many language systems provide them as part of their standard library. In some languages, they are not only built into the standard system, but have special syntax, often using array-like subscripting.

Built-in syntactic support for associative arrays was introduced by SNOBOL4, under the name "table". MUMPS made multi-dimensional associative arrays, optionally persistent, its key data structure. SETL supported them as one possible implementation of sets and maps. Most modern scripting languages, starting with AWK and including Rexx, Perl, Tcl, JavaScript, Wolfram Language, Python, Ruby, Go, and Lua, support associative arrays as a primary container type. In many more languages, they are available as library functions without special syntax.

In Smalltalk, Objective-C, .NET,[13] Python, REALbasic, Swift, VBA and Delphi[14] they are called *dictionaries*; in Perl, Ruby and Seed7 they are called *hashes*; in C++, Java, Go, Clojure, Scala, OCaml, Haskell they are called *maps* (see map (C++), unordered_map (C++), and `Map (https://docs.oracle.com/javase/9/docs/api/java/util/Map.html)`); in Common Lisp and Windows PowerShell, they are called *hash tables* (since both typically use this implementation). In PHP, all arrays can be associative, except that the keys are limited to integers and strings. In JavaScript (see also JSON), all objects behave as associative arrays with string-valued keys, while the Map and WeakMap types take arbitrary objects as keys. In Lua, they are called *tables*, and are used as the primitive building block for all data structures. In Visual FoxPro, they are called *Collections*. The D language also has support for associative arrays.[15]

Permanent storage

Most programs using associative arrays will at some point need to store that data in a more permanent form, like in a computer file. A common solution to this problem is a generalized concept known as *archiving* or *serialization*, which produces a text or binary representation of the original objects that can be written directly to a file. This is most commonly implemented in the underlying object model, like .Net or Cocoa, which include standard functions that convert the internal data into text form. The program can create a complete text representation of any group of objects by calling these methods, which are almost always already implemented in the base associative array class.[16]

For programs that use very large data sets, this sort of individual file storage is not appropriate, and a database management system (DB) is required. Some DB systems natively store associative arrays by serializing the data and then storing that serialized data and the key. Individual arrays can then be loaded or saved from the database using the key to refer to them. These key-value stores have been used for many years and have a history as long as that as the more common relational database (RDBs), but a lack of standardization, among other reasons, limited their use to certain niche roles. RDBs were used for these roles in most cases, although saving objects to a RDB can be complicated, a problem known as object-relational impedance mismatch.

After c. 2010, the need for high performance databases suitable for cloud computing and more closely matching the internal structure of the programs using them led to a renaissance in the key-value store market. These systems can store and retrieve associative arrays in a native fashion, which can greatly improve performance in common web-related workflows.

See also

- Key-value database
- Tuple
- Function (mathematics)
- JSON

References

1. Goodrich, Michael T.; Tamassia, Roberto (2006), "9.1 The Map Abstract Data Type", *Data Structures & Algorithms in Java* (4th ed.), Wiley, pp. 368–371
2. Mehlhorn, Kurt; Sanders, Peter (2008), "4 Hash Tables and Associative Arrays", *Algorithms and Data Structures: The Basic Toolbox* (http://people.mpi-inf.mpg.de/~mehlhorn/ftp/Toolbox/HashTables.pdf) (PDF), Springer, pp. 81–98
3. Anderson, Arne (1989). "Optimal Bounds on the Dictionary Problem". *Proc. Symposium on Optimal Algorithms*. Springer Verlag: 106–114.
4. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), "11 Hash Tables", *Introduction to Algorithms* (2nd ed.), MIT Press and McGraw-Hill, pp. 221–252, ISBN 0-262-03293-7.
5. Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Meyer auf der Heide, F., Rohnert, H., and Tarjan, R. E. 1994. "Dynamic Perfect Hashing: Upper and Lower Bounds" (http://www.arl.wustl.edu/~sailesh/download_files/Limited_Edition/hash/Dynamic%20Perfect%20Hashing-%20Upper%20and%20Lower%20Bounds.pdf). SIAM J. Comput. 23, 4 (Aug. 1994), 738-761. http://portal.acm.org/citation.cfm?id=182370 doi:10.1137/S0097539791194094 (https://doi.org/10.1137%2FS0097539791194094)
6. Goodrich & Tamassia (2006), pp. 597–599.
7. Goodrich & Tamassia (2006), pp. 389–397.
8. "When should I use a hash table instead of an association list?" (http://www.faqs.org/faqs/lisp-faq/part2/section-2.html). lisp-faq/part2. 1996-02-20.
9. Klammer, F.; Mazzolini, L. (2006), "Pathfinders for associative maps", *Ext. Abstracts GIS-I 2006*, GIS-I, pp. 71–74.
10. Joel Adams and Larry Nyhoff. "Trees in STL" (http://cs.calvin.edu/books/c++/intro/3e/WebItems/Ch15-Web/STL-Trees.pdf). Quote: "The Standard Template library ... some of its containers -- the set<T>, map<T1, T2>, multiset<T>, and multimap<T1, T2> templates -- are generally built using a special kind of *self-balancing binary search tree* called a *red-black tree*."
11. Knuth, Donald (1998). *'The Art of Computer Programming'*. 3: *Sorting and Searching* (2nd ed.). Addison-Wesley. pp. 513–558. ISBN 0-201-89685-0.
12. Probst, Mark (2010-04-30). "Linear vs Binary Search" (https://schani.wordpress.com/2010/04/30/linear-vs-binary-search/). Retrieved 2016-11-20.
13. "Dictionary<TKey, TValue> Class" (http://msdn.microsoft.com/en-us/library/xfhwa508.aspx). MSDN.
14. "System.Generics.Collections.TDictionary - RAD Studio API Documentation" (http://docwiki.embarcadero.com/Libraries/Tokyo/en/System.Generics.Collections.TDictionary). *docwiki.embarcadero.com*. Retrieved 2017-04-18.
15. "Associative Arrays, the D programming language" (http://dlang.org/hash-map.html). Digital Mars.
16. "Archives and Serializations Programming Guide" (https://developer.apple.com/library/prerelease/ios/documentation/Cocoa/Conceptual/Archiving/Archiving.html#//apple_ref/doc/uid/10000047i), Apple Inc., 2012

External links

- NIST's Dictionary of Algorithms and Data Structures: Associative Array (https://xlinux.nist.gov/dads/HTML/assocarray.html)