* [Skip navigation](#content" \o "skip navigation)
* [Skip to search form](#q)

# **December Column: Searching Algorithms**

By [George T. Heineman](http://www.oreillynet.com/pub/au/3414)  
December 4, 2008 ([1ed](http://broadcast.oreilly.com/2008/12/december-column-searching-algo.html)) | Revised January 3 2016 (2ed)

**Overview**

This is the second of a series of monthly columns in the blog associated with the [**Algorithms in a Nutshell**](http://oreilly.com/catalog/9780596516246/)**, 1ed** book, published October 2008 by [**O'Reilly Media, Inc.**](http://www.oreilly.com/)

[**In our last column**](http://broadcast.oreilly.com/2008/10/welcome-to-algorithms-in-a-nut.html) we described:

* how to download and install the **Algorithm Development Kit (ADK)**, the code repository associated with the book.
* how to use some of the **Sorting** algorithms implemented in C.
* how to download the November code samples

In **Chapter 5: Searching** of [**Algorithms in a Nutshell**](http://oreilly.com/catalog/9780596516246/), we present the three common forms of a search over a collection *C* of elements:

1. Existence: Does *C* contain a target element *t*?
2. Retrieval: Return the element in *C* that matches the target element *t*
3. Associative Lookup: Return information associated in collection *C* with the target key *k*.

In this column we focus only on the first search type and the material in this month's column extends what you will find in the book. The examples are programmed in Java and at the [**end of this column**](#Eclipse), we show how to import the ADK code into an Eclipse workspace.

**Retrieving and building December code samples**

All Blog code samples are found on the github repository (<https://github.com/heineman/algorithms-nutshell-2ed.git>) in the Blogs/ project. You must have built the ADK before attempting any of these Blog examples.

**API Interface for searching routines**

The following snippet presents an API (defined for this blog) that determines whether an element exists within a collection of elements.

public interface ICollectionSearch<E> {

/\*\*

\* Determine if the given target element exists in

\* a collection.

\* @param target non-null element to be searched for.

\* @return true if the target element exists within

\* collection; false otherwise.

\*/

boolean exists(E target);

}

This interface takes advantage of Java generics to enable the searchable collections to be parameterized by the type of base element they contain. The implementations of the search algorithms presented in this column implement the ICollectionSearch interface so we can directly compare the approaches. In this column we cover:

* Hash-based Search (including *LinearProbing*, *Quadratic Probing*, *Collision Lists*, and *Perfect Hashing*)
* Balanced Binary Tree Search

We use the 213,557 word dictionary provided with the ADK release (the actual file can be found in $ADKHOME/Blogs/artifacts/searching/words.english.txt). From this dictionary, we construct two special lists of words:

* **keys2** contains 641 words
* **keys3** contains 6,027 words

To construct these files, execute the following within the Blog/ directory:

% java algs.blog.searching.main.ConstructTwo artifacts/searching/words.english.txt

This will output the keys\_2.txt file that you see in $ADKHOME/Blogs/artifacts/searching/keys\_2.txt. A similar execution produces the other file:

% java algs.blog.searching.main.ConstructThree artifacts/searching/words.english.txt

At the end of this column, we will explain why these lists are special. In each of the following sections, we evaluate: (a) the cost of constructing the initial collection for each of these two lists; (b) the cost of determining whether each of the 213,557 words exists in the collection. We simply do not have time for a full treatise on hashing! Rather, we aim to explain further some of the benefits of using hashing, as well as some of the weaknesses.

To keep our terminology clear, we use *n* to refer to the size of the collection over which a search is processed. For each of these *n* elements, one can compute its *key* value. For hash-based collections, an array of *b* bins (also called slots) is constructed.

**Sequential Search: The Strawman Algorithm**

The desire to investigate hashing is to overcome the weakness of Sequential Search. If a collection is unsorted, then searching for an element demands O(*n*) worst-case performance. Even when the collection is sorted, however, for uniform distribution of elements, one still must inspect about half of the elements, which still results in O(*n*) performance.

**Hash-based Search: Linear Probing**

In essence, *linear probing* disperses the elements of the collection throughout an array such that only a small number of probes is required to determine whether an element exists in the array. The first innovation is to recognize that *b* (the number of bins) can be larger than *n* (the number of elements). To compute the bin in which to store an element, a hash function takes an element's key value and normalizes it to fit within one of the *b* bins, typically using the modulo ('%') operator. The resulting bin number is called *h*. The second innovation is to resolve collisions when the computed bin for two (or more) elements is the same. The *linear probing* technique uses *open addressing* to resolve conflicts, which means that the insert(e) method chooses another bin in which to place the item being inserted; subsequently, the exists(e) method may have to inspect multiple bins to determine if the target element exists within the collection.

The following method (taken from the *HashBasedSearch* class which is available in this month's code attachment) shows the logic for inserting an element into the array representing the hash table. This method returns the number of probes used to find an empty bin or throws a *RuntimeException* should the method be unable to locate an empty bin. If *b* is a prime number, then any delta in the range [*1*, *b-1*] will be suitable; otherwise, just be sure that the greatest common divisor of *b* and *delta* is one (i.e., they are relatively prime).

/\*\*

\* Insert element into collection represented in storage[].

\*

\* Should the count of attempted slots reach the array

\* size, declare that the element cannot be added (either

\* because of a poor hash function or because the array

\* is full).

\*

\* If element already exists within collection, then the

\* collection is unchanged and we return number of probes

\* to locate the element.

\*

\* @param e Element to insert

\* @return # of probes to find location.

\* @throws RuntimeException should no empty location

\* be found, given the delta step

\*/

public int insert(E e) throws RuntimeException {

// compute initial bin location

int hash = (e.hashCode() & 0x7FFFFFFF) % storage.length;

int h = hash;

for (int p = 1; p <= storage.length; p++) {

if (storage[h] == null) {

storage[h] = e;

num++;

return p;

} else if (storage[h].equals(e)) {

return p;

}

// advance to next bin

h = probe.next(hash, p);

}

throw new RuntimeException ("Unable to insert element: " +

num + " slots taken out of " + storage.length);

}

A LinearProbe probe object computes the appropriate bin to inspect with each probe attempt. If a computed bin number (*h*) is already occupied, then *linear probing* resolves the collision by choosing bin *(h + p\*delta) % tableSize* where *delta* is a preassigned constant and *p* is the probe number. For example, if delta is "3", then inserting an element into the array would place the element in the first open bin of the series *h*, *h+3*, *h+6*, .... All of these bin numbers are computed modulo *b* thus treating the array as a circular structure. We use the term *chain* to describe the number of probes that are required to properly place an element into the array.

Execute the following Java program to generate data for the following table:

% java algs.blog.searching.main.Main

The following table shows the results of inserting the 641 strings for **keys2** into hashtables of different sizes while keeping a *delta* value of 1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **array size *b*** | **empty bins** | **average  chain size** | **max chain size** | **average search time (ms.)** |
| 9,615 | 8,974 | 1.03 | 2 | 13.39 |
| 5,128 | 4,487 | 1.07 | 3 | 14.54 |
| 4,413 | 3,772 | 1.10 | 5 | 16.18 |
| 2,884 | 2,243 | 1.14 | 6 | 17.29 |
| 1,762 | 1,121 | 1.26 | 12 | 24.57 |
| 1,201 | 560 | 1.49 | 10 | 34.57 |
| 921 | 280 | 2.41 | 53 | 98.21 |
| 781 | 140 | 2.98 | 46 | 149.04 |
| 711 | 70 | 4.18 | 64 | 313.61 |
| 676 | 35 | 5.29 | 127 | 565.89 |
| 658 | 17 | 13.82 | 446 | 2966.54 |

As the number of bins, *b* approaches *n*, the number of collisions increases. The striking figure is the last column, which computes the average time needed to query all 213,557 words. While the average number of probes is still quite small, there are some elements that require O(n) number of probes. Clearly, the user has some control of the situation, at the expense of choosing a sufficiently large array to use.

The choice of *delta* is not critical. Consider the following table which shows results for different *delta* values for four specific array sizes.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Table Size: 641 | | Table Size: 1283 | | Table Size: 2557 | | Table Size: 5113 | |
| **Delta** | **Average** | **Max** | **Average** | **Max** | **Average** | **Max** | **Average** | **Max** |
| 1 | 21.11 | 585 | 1.43 | 7 | 1.17 | 7 | 1.07 | 4 |
| 2 | 21.05 | 615 | 1.50 | 10 | 1.15 | 6 | 1.06 | 3 |
| 3 | 17.37 | 585 | 1.42 | 7 | 1.15 | 6 | 1.07 | 3 |
| 4 | 23.53 | 593 | 1.49 | 15 | 1.14 | 5 | 1.07 | 4 |
| 5 | 15.02 | 570 | 1.44 | 8 | 1.15 | 6 | 1.07 | 4 |
| 6 | 21.68 | 490 | 1.53 | 17 | 1.17 | 6 | 1.06 | 3 |
| 7 | 18.02 | 596 | 1.51 | 13 | 1.13 | 4 | 1.08 | 4 |
| 8 | 12.26 | 299 | 1.50 | 11 | 1.15 | 6 | 1.07 | 3 |
| 9 | 12.12 | 564 | 1.50 | 16 | 1.14 | 5 | 1.06 | 4 |

My advice is to choose an array size that is about 4-8 times as large as expected, since this should not only reduce the total number of probes but the variability in the number of probes used to place each item. Once the hash table array is constructed, the maximum chain within the array determines the time to determine whether an element exists. Here one trades off extra (unused) space to achieve reduced number of probes.

**Hash-based Search: Quadratic Probing**

*Quadratic probing* attempts to avoid the clustering that occurs when using a linear probe. A quadratic probe computes the pth bin using a formula of the form: *((int)(hash + c1\*p + c2\*p\*p)) % tableSize* where *c1* and *c2* are floating point coefficients. In the examples for this column, we choose two commonly used quadratic functions: (a) c1 = c2 = ½;(b) c2 = c2 = 1. As with *linear probing*, the computed result is normalized it to fit within one of the *b* bins. If we re-execute the trials for **keys2** we have the following result:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **average chain size** | | **max chain size** | | **average search time (ms.)** | |
| **array size *b*** | **empty bins** | **q1=½ q2 =½** | **q1=1 q2=1** | **q1=½ q2 =½** | **q1=1 q2=1** | **q1=½ q2 =½** | **q1=1 q2=1** |
| 9,615 | 8,974 | 1.03 | 1.03 | 2 | 3 | 13.93 | 13.43 |
| 5,128 | 4,487 | 1.08 | 1.07 | 4 | 4 | 15.07 | 15.07 |
| 4,413 | 3,772 | 1.09 | 1.08 | 4 | 3 | 15.65 | 15.61 |
| 2,884 | 2,243 | 1.14 | 1.12 | 4 | 3 | 17.89 | 17.29 |
| 1,762 | 1,121 | 1.26 | 1.26 | 8 | 8 | 24.00 | 24.00 |
| 1,201 | 560 | 1.45 | 1.44 | 8 | 9 | 35.18 | 34.61 |
| 921 | 280 | 1.89 | 1.86 | 13 | 16 | 57.46 | 59.71 |
| 781 | 140 | 2.16 | 2.26 | 19 | 24 | 90.93 | 89.29 |
| 711 | 70 | 2.63 | 2.89 | 30 | 44 | 169.61 | 163.5 |
| 676 | 35 | 3.24 | 3.33 | 51 | 74 | 277.93 | 498.32 |
| 658 | 17 | 4.07 | \*\* | 91 | \*\* | 574.21 | \*\* |

*Quadratic probing* initially is just a bit less efficient than linear probing, but after a crossover point (when the array is twice as large as the number of elements it contains) its growth rate is much better. In the final case, using *quadratic probing* performs almost five times as fast as *linear probing*. One must still be cautious, however, in ensuring that the selected coefficients produce an efficient distribution of elements to the bins. For example, when c1 = c2 = 1, the resulting behavior is more similar to linear probing, and for *b* = 658, this *quadratic probing* function only allows 621 of the slots to be filled before it fails to locate a suitable empty slot when attempting to insert the 622nd word.

**Hash-based Search: Collision Lists**

Hash-based Search can choose to resolve collisions instead by constructing lists to store all elements whose computed hash function returns the identical value. This is the approach implemented by the java.util.Hashtable class provided with the Java Development Kit (JDK).

If the number of elements in the hash table exceeds a specific "load factor" threshold, then the hash table is expanded (typically, the size *b* is changed to *2\*b+1*) and the elements stored in the hashtable are "rehashed" to their new location. The amortized savings over the life of the hashtable enables these costly rehash methods to be absorbed such that there are clear costs savings (as we document in **Table 5-7** of the book). To get a sense of how the JDK computes hash tables, we executed three experiments to populate a hashtable from the desired key elements using an initial hashtable whose capacity was {11, 4\**n*=2564, and 4413} where *n*=641. In each case, the time to construct the hashtable is negligible. Note that when starting with an initial size of 11, the resulting hashtable (after inserting 641 elements) had space for 1,535 due to rehashing. The average search time needed to query all 213,557 words is also shown in the table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Linear** | **Quadratic** | **JDK Hashtable experiment 1**  ***b*=1,535** | **JDK Hashtable experiment 2**  ***b*=2,564** | **JDK Hashtable experiment 3**  ***b*=4,413** |
| **Average Chain Size** | 1.10 | 1.09 | 1.21 | 1.13 | 1.08 |
| **Average Search Time (ms.)** | 16.18 | 15.65 | 30.14 | 27.92 | 29.57 |

These execution results are about twice as slow as the *linear probing* and *quadratic probing*. Why does this happen? If you inspect the source code for the java.util.Hashtable code, you will see that it resolves collisions by using linked lists of elements that all hash to the same bin. This is a practical solution to hashing since it allows elements to be deleted from a hashtable. One cannot simply delete elements from the arrays used for *linear probing* or *quadratic probing*; instead, one must mark a bin as deleted to enable the chains to still be processed. But what is the average length of these linked lists within the java.util.Hashtable? The designers of this class have made it nearly impossible to extract this useful bit of information. I wrote a small class, *HashtableReport* that hacks the serializable representation of a hashtable to get access to this information. You might find this solution nifty! In any event, the above table shows that the average chain length is initially higher (about 30%); even when this is smaller (as it is for experiment 3), however, the performance is simply not as efficient, due to a few extra computations needed to traverse these linked lists.

**Hash-based Search: Perfect Hashing**

A *perfect hash* is a hash function that allows one to determine in a single probe of an array representing a collection whether a target word exists in that collection. There are several freely available software packages for perfect hashing. For C, the GNU **GPERF** package generates C code to perform perfect hashing. The most commonly cited is [**Doug Schmidt's Perfect Hash Generator**](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.8511) and the code can be retrieved from [**GNU**](http://www.gnu.org/software/gperf/).

To create a perfect hash, the entire collection of elements to be searched must be known in advance and a suitable hash function is designed that dictates the array position in which a target word can be found should it exist within the collection. The first weakness to perfect hashing is that the size of the array used to represent the collection becomes quite large and sparse. For example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Keys list** | **Number of words** | **Size of Array** | **Percentage used** | **Average Search Time (ms.)** |
| **keys2** | 641 | 4,413 | 14.5% | 29.57 |
| **keys3** | 6,027 | 264,240 | 2.3% | 39.04 |

Clearly, such an approach is impractical for very large word sets! The second weakness to perfect hashing is the upfront cost of constructing the array representing the collection and searching for the hash function. For example, on a dual-AMD Opteron Processor with a 2 gigahertz chip, **GPERF** took nearly 2½ minutes to compute the array and hash function for **keys3**.

For this column, I used **GPERF** to generate C code for **keys2** and **keys3** and then manually converted this code to Java. Because of some limitations of constant pooling enforced by the Java compiler, the *GPerfThree* code must load up the large array from disk. In the above table, you can see that the performance of **GPERF** for **keys3** slows down. This is due to the added complexity of its hash function.

**Hashing with Special Cases**

If the collection has specific properties, then it is possible to hand-craft an efficient hash function to be used for perfect hashing. Indeed, in a small number of circumstances, you may be able to construct a hash function that requires very little extra storage in the array used to store the collection. If you review the hash function within *GPerfTwo* and *GPerfThree* you can see that they are based on inspecting just a fixed number of characters of the string. We took this approach to an extreme when we selected the words that make up the **keys2** and **keys3** sets. Specifically:

* **keys2** is a set of words whose *first* and *second-to-last* character are unique within the set. Out of a total possible of 26 \* 26 = 676 words, this set contains 641 (a density of 94.8%)
* **keys3** is a set of words whose *first*, *middle*, and *second-to-last* characters are unique within the set. Because the last six letters of the alphabet are infrequently used, we only selected words that have letters from [a-t] within these three designated locations. Out of a total possible of 20 \* 20 \* 20 = 8,000 words, this set contains 6,021 words (a density of 75%)

Using this approach, the exists method defined by the *ICollectionSearch* API is implemented as follows:

public boolean exists(String t) {

int first= (int)(t.charAt(0)-'a');

int second= (int)(t.charAt(t.length()-2)-'a');

return target.equals(storage[first\*26+second]);

}

For this special set, we have designed an efficient hash function to differentiate each of its 641 member elements. A similiar method is written for **keys3**. These hash methods outperform all other generic methods described in this column, once again reaffirming our observation that you must take advantage of every special characteristic of your input set when designing efficient algorithms. The following table shows the performance of these special hash functions:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Keys list** | **Number of words** | **Size of Array** | **Percentage used** | **Average Search Time (ms.)** |
| **keys2** | 641 | 676 | 94.8% | 9.5 |
| **keys3** | 6,027 | 8,000 | 75% | 16.21 |

**Using Balanced Binary Trees to store collection**

It is worthwhile to compare the hashing approaches of this column against simply using a balanced binary tree. The code is found in *BalancedTreeSearch*. When you simply don't have the space to burn when using sparse arrays as hashtables, then balanced trees are a reasonable solution since the total number of "probes" can be limited to the height of the tree, or O(*log n*) for balanced binary trees. Here are the performance results:

|  |  |  |  |
| --- | --- | --- | --- |
| **Keys list** | **Average Node  Height** | **Height of Tree** | **Average Search Time (ms.)** |
| **keys2** | 8.63 | 16 | 70.36 |
| **keys3** | 11.90 | 22 | 117.21 |

Note that the depth of the tree is effectively the maximum number of "probes" one can use to locate an element in the tree; we have also computed the average number of probes needed to locate each individual element within the tree. Using *linear probing*, the only case where the average chain size was as large as 11.90 for the **keys2** set was for the array size *b=658* and the search time was 2,966.54 which is more than 40 times slower. Clearly the structure of the balanced tree enables more efficient searching while the size is nearly the same.

**Useful hashcode shortcut**

Because the hashCode method may indeed return a negative number, there is a useful trick you can use to replace the following code, which we used in **Example 5.8** of the book:

int h = v.hashCode();

if (h < 0) { h = 0 - h; }

return h % tableSize;

The above statements can be replaced with the single statement:

return (v.hashCode() & 0x7ffffffff) % tableSize;

I learned this shortcut by reading the java.util.Hashtable source code for the JDK.

**Next Column**

In next Month's January column, we will further investigate algorithms from **Chapter 6: Graph Algorithms**. Until next time, we hope you take the opportunity to investigate the numerous algorithms in the Algorithms in a Nutshell book as well as to explore the examples provided in the ADK.

[**Algorithms in a Nutshell**](http://oreilly.com/catalog/9780596516246/)  
George T. Heineman, Gary Pollice, Stanley Selkow