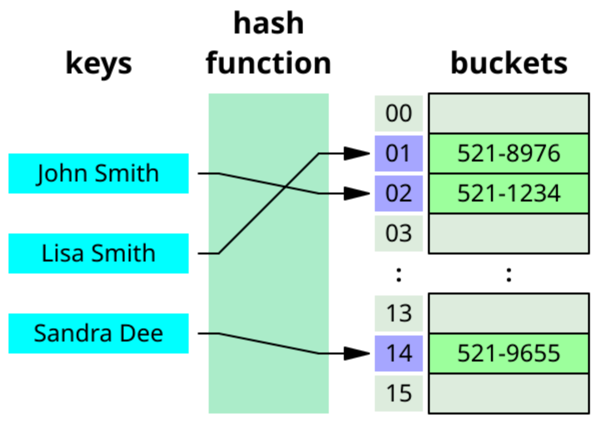
Hash table

In computer science, a **hash table** is a data structure that implements an associative array, also called a **dictionary** or simply **map**; an associative array is an abstract data type that maps keys to values.[3] A hash table uses a hash function to compute an *index*, also called a *hash code*, into an array of *buckets* or *slots*, from which the desired value can be found. During lookup, the key is hashed and the resulting hash indicates where the corresponding value is stored. A map implemented by a hash table is called a **hash map**.



A small phone book as a hash table

Hash table		
Type	Unordered associative array	
Invented	1953	
Time complexity in big O notation		
Operation	**Average**	**Worst case**
Search	$\Theta(1)$	$O((\log n)^2)$[1]
Insert	$\Theta(1)$	$O((\log n)^2)$[1]
Delete	$\Theta(1)$	$O(n)$
Space complexity		
Space	$\Theta(n)$[2]	$O(n)$

Most hash table designs employ an imperfect hash function. Hash collisions, where the hash function generates the same index for more than one key, therefore typically must be accommodated in some way.

In a well-dimensioned hash table, the average time complexity for each lookup is independent of the number of elements stored in the table. Many hash table designs also allow arbitrary insertions and deletions of key–value pairs, at amortized constant average cost per operation.[4][5][6]

Hashing is an example of a space-time tradeoff. If memory is infinite, the entire key can be used directly as an index to locate its value with a single memory access. On the other hand, if infinite time is available, values can be stored without regard for their keys, and a binary search or linear search can be used to retrieve the element.[7]:458

In many situations, hash tables turn out to be on average more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets.

History

The idea of hashing arose independently in different places. In January 1953, Hans Peter Luhn wrote an internal IBM memorandum that used hashing with chaining. The first example of open addressing was proposed by A. D. Linh, building on Luhn's memorandum.[5]:547 Around the same time, Gene Amdahl, Elaine M. McGraw, Nathaniel Rochester, and Arthur Samuel of IBM Research implemented hashing for the IBM 701 assembler.[8]:124 Open addressing with linear probing is credited to Amdahl, although Andrey Ershov independently had the same idea.[8]:124–125 The term "open addressing" was coined by W. Wesley Peterson in his article which discusses the problem of search in large files.[9]:15

The first published work on hashing with chaining is credited to Arnold Dumey, who discussed the idea of using remainder modulo a prime as a hash function.[9]:15 The word "hashing" was first published in an article by Robert Morris.[8]:126 A theoretical analysis of linear probing was submitted originally by Konheim and Weiss.[9]:15

Overview

An associative array stores a set of (key, value) pairs and allows insertion, deletion, and lookup (search), with the constraint of unique keys. In the hash table implementation of associative arrays, an array A of length m is partially filled with n elements, where $m \geq n$. A key x is hashed using a hash function h to compute an index location $A[h(x)]$ in the hash table, where $h(x) < m$. At this index, both the key and its associated value are stored. Storing the key alongside the value ensures that lookups can verify the key at the index to retrieve the correct value, even in the presence of collisions. Under reasonable assumptions, hash tables have better time complexity bounds on search, delete, and insert operations in comparison to self-balancing binary search trees.[9]:1

Hash tables are also commonly used to implement sets, by omitting the stored value for each key and merely tracking whether the key is present.[9]:1

Load factor

A *load factor* α is a critical statistic of a hash table, and is defined as follows:[2]

$$\text{load factor } (\alpha) = \frac{n}{m},$$

where

- n is the number of entries occupied in the hash table.
- m is the number of buckets.

The performance of the hash table deteriorates in relation to the load factor α.[9]:2

The software typically ensures that the load factor α remains below a certain constant, $\alpha_{\max}$. This helps maintain good performance. Therefore, a common approach is to resize or "rehash" the hash table whenever the load factor α reaches $\alpha_{\max}$. Similarly the table may also be resized if the load factor drops below $\alpha_{\max}/4$.[10]

Load factor for separate chaining

With separate chaining hash tables, each slot of the bucket array stores a pointer to a list or array of data.[11]

Separate chaining hash tables suffer gradually declining performance as the load factor grows, and no fixed point beyond which resizing is absolutely needed.[10]

With separate chaining, the value of $\alpha_{\max}$ that gives best performance is typically between 1 and 3.[10]

Load factor for open addressing

With open addressing, each slot of the bucket array holds exactly one item. Therefore an open-addressed hash table cannot have a load factor greater than 1.[11]

The performance of open addressing becomes very bad when the load factor approaches 1.[10]

Therefore a hash table that uses open addressing *must* be resized or *rehashed* if the load factor α approaches 1.[10]

With open addressing, acceptable figures of max load factor $\alpha_{\max}$ should range around 0.6 to 0.75.[12][13]:110

Hash function

A hash function $h : U \rightarrow \{0, \ldots, m-1\}$ maps the universe U of keys to indices or slots within the table, that is, $h(x) \in \{0, \ldots, m-1\}$ for $x \in U$. The conventional implementations of hash functions are based on the *integer universe assumption* that all elements of the table stem from the universe $U = \{0, \ldots, u-1\}$, where the bit length of u is confined within the word size of a computer architecture.[9]:2

A hash function h is said to be perfect for a given set S if it is injective on S, that is, if each element $x \in S$ maps to a different value in $0, \ldots, m-1$.[14][15] A perfect hash function can be created if all the keys are known ahead of time.[14]

Integer universe assumption

The schemes of hashing used in *integer universe assumption* include hashing by division, hashing by multiplication, universal hashing, dynamic perfect hashing, and static perfect hashing.[9]:2 However, hashing by division is the commonly used scheme.[16]:264 [13]:110

Hashing by division

The scheme in hashing by division is as follows:[9]:2

$$h(x) \ = \ x \bmod m,$$

where $h(x)$ is the hash value of $x \in S$ and m is the size of the table.

Hashing by multiplication

The scheme in hashing by multiplication is as follows:[9]:2–3

$$h(x) = \lfloor m\bigl((xA) \bmod 1\bigr)\rfloor$$

Where A is a non-integer real-valued constant and m is the size of the table. An advantage of the hashing by multiplication is that the m is not critical.[9]:2–3 Although any value A produces a hash function, Donald Knuth suggests using the golden ratio.[9]:3

Choosing a hash function

Uniform distribution of the hash values is a fundamental requirement of a hash function. A non-uniform distribution increases the number of collisions and the cost of resolving them. Uniformity is sometimes difficult to ensure by design, but may be evaluated empirically using statistical tests, e.g., a Pearson's chi-squared test for discrete uniform distributions.[17][18]

The distribution needs to be uniform only for table sizes that occur in the application. In particular, if one uses dynamic resizing with exact doubling and halving of the table size, then the hash function needs to be uniform only when the size is a power of two. Here the index can be computed as some range of bits of the hash function. On the other hand, some hashing algorithms prefer to have the size be a prime number.[19]

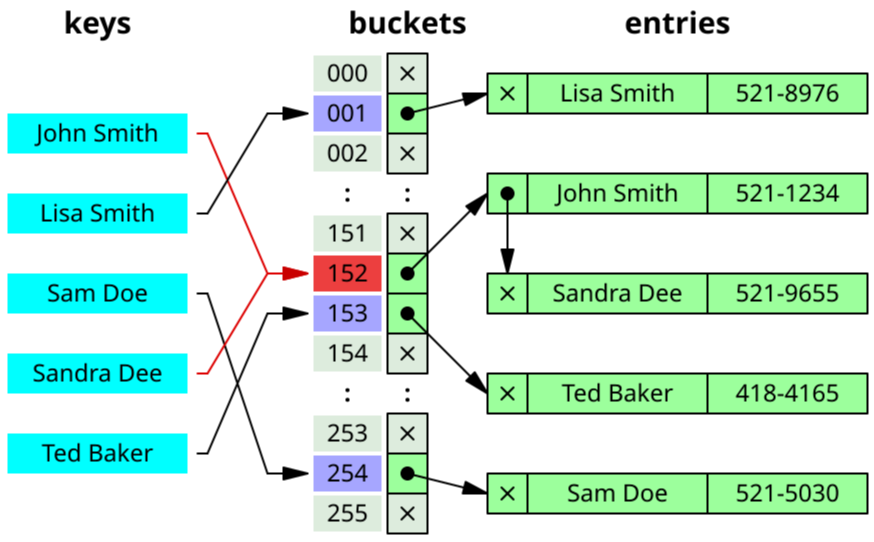
For open addressing schemes, the hash function should also avoid *clustering*, the mapping of two or more keys to consecutive slots. Such clustering may cause the lookup cost to skyrocket, even if the load factor is low and collisions are infrequent. The popular multiplicative hash is claimed to have particularly poor clustering behavior.[19][5]

K-independent hashing offers a way to prove a certain hash function does not have bad keysets for a given type of hashtable. A number of K-independence results are known for collision resolution schemes such as linear probing and cuckoo hashing. Since K-independence can prove a hash function works, one can then focus on finding the fastest possible such hash function.[20]

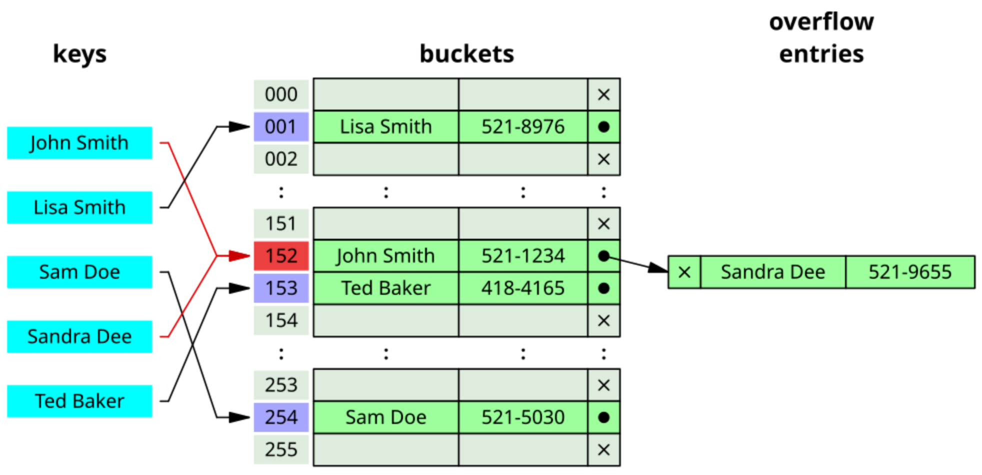
Collision resolution

A search algorithm that uses hashing consists of two parts. The first part is computing a hash function which transforms the search key into an array index. The ideal case is such that no two search keys hash to the same array index. However, this is not always the case and impossible to guarantee for unseen given data.[21]:515 Hence the second part of the algorithm is collision resolution. The two common methods for collision resolution are separate chaining and open addressing.[7]:458

Separate chaining



Hash collision resolved by separate chaining



Hash collision by separate chaining with head records in the bucket array.

In separate chaining, the process involves building a linked list with key–value pair for each search array index. The collided items are chained together through a single linked list, which can be traversed to access the item with a unique search key.[7]:464 Collision resolution through chaining with linked list is a common method of implementation of hash tables. Let T and x be the hash table and the node respectively, the operation involves as follows:[16]:258

```
Chained-Hash-Insert(T, k)
  insert x at the head of linked list T[h(k)]

Chained-Hash-Search(T, k)
  search for an element with key k in linked list T[h(k)]

Chained-Hash-Delete(T, k)
  delete x from the linked list T[h(k)]
```

If the element is comparable either numerically or lexically, and inserted into the list by maintaining the total order, it results in faster termination of the unsuccessful searches.[21]:520–521

Other data structures for separate chaining

If the keys are ordered, it could be efficient to use "self-organizing" concepts such as using a self-balancing binary search tree, through which the theoretical worst case could be brought down to $O(\log n)$, although it introduces additional complexities.[21]:521

In dynamic perfect hashing, two-level hash tables are used to reduce the look-up complexity to be a guaranteed $O(1)$ in the worst case. In this technique, the buckets of k entries are organized as perfect hash tables with k^2 slots providing constant worst-case lookup time, and low amortized time for insertion.[22] A study shows array-based separate chaining to be 97% more performant when compared to the standard linked list method under heavy load.[23]:99

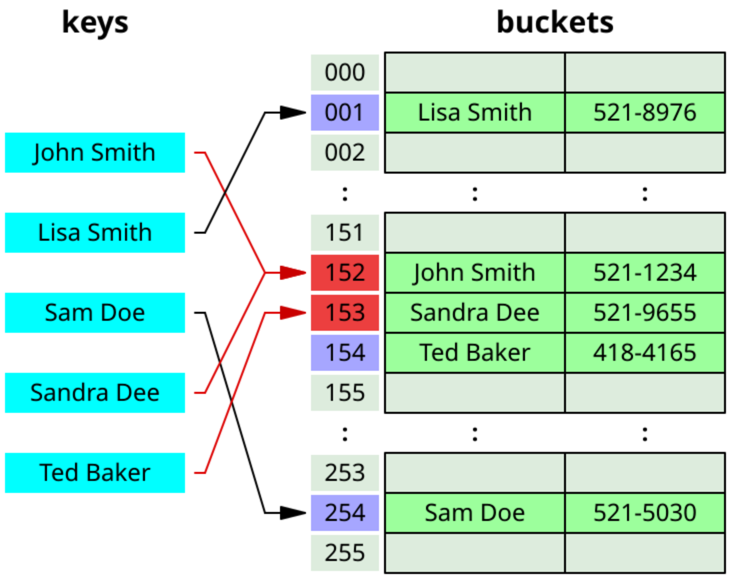
Techniques such as using fusion tree for each buckets also result in constant time for all operations with high probability.[24]

Caching and locality of reference

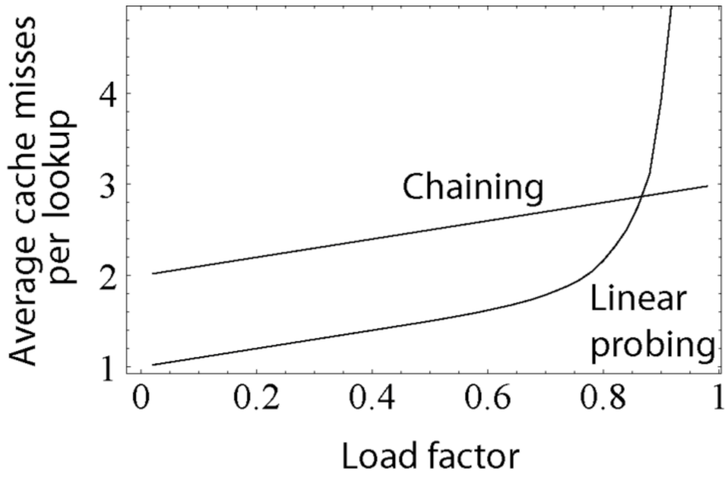
The linked list of separate chaining implementation may not be cache-conscious due to spatial locality—locality of reference—when the nodes of the linked list are scattered across memory, thus the list traversal during insert and search may entail CPU cache inefficiencies.[23]:91

In cache-conscious variants of collision resolution through separate chaining, a dynamic array found to be more cache-friendly is used in the place where a linked list or self-balancing binary search trees is usually deployed, since the contiguous allocation pattern of the array could be exploited by hardware-cache prefetchers—such as translation lookaside buffer—resulting in reduced access time and memory consumption.[25][26][27]

Open addressing



Hash collision resolved by open addressing with linear probing (interval=1). Note that "Ted Baker" has a unique hash, but nevertheless collided with "Sandra Dee", that had previously collided with "John Smith".



This graph compares the average number of CPU cache misses required to look up elements in large hash tables (far exceeding size of the cache) with chaining and linear probing. Linear probing performs better due to better locality of reference, though as the table gets full, its performance degrades drastically.

Open addressing is another collision resolution technique in which every entry record is stored in the bucket array itself, and the hash resolution is performed through **probing**. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some *probe sequence*, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates an unsuccessful search.[28]

Well-known probe sequences include:

- Linear probing, in which the interval between probes is fixed (usually 1).[29]
- Quadratic probing, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the value given by the original hash computation.[30]:272
- Double hashing, in which the interval between probes is computed by a secondary hash function.[30]:272–273

The performance of open addressing may be slower compared to separate chaining since the probe sequence increases when the load factor α approaches 1.[10][23]:93 The probing results in an infinite loop if the load factor reaches 1, in the case of a completely filled table.[7]:471 The average cost of linear probing depends on the hash function's ability to distribute the elements uniformly throughout the table to avoid clustering, since formation of clusters would result in increased search time.[7]:472

Caching and locality of reference

Since the slots are located in successive locations, linear probing could lead to better utilization of CPU cache due to locality of references resulting in reduced memory latency.[29]

Other collision resolution techniques based on open addressing

Coalesced hashing

Coalesced hashing is a hybrid of both separate chaining and open addressing in which the buckets or nodes link within the table.[31]:6–8 The algorithm is ideally suited for fixed memory allocation.[31]:4 The collision in coalesced hashing is resolved by identifying the largest-indexed empty slot on the hash table, then the colliding value is inserted into that slot. The bucket is also linked to the inserted node's slot which contains its colliding hash address.[31]:8

Cuckoo hashing

Cuckoo hashing is a form of open addressing collision resolution technique which guarantees $O(1)$ worst-case lookup complexity and constant amortized time for insertions. The collision is resolved through maintaining two hash tables, each having its own hashing function, and collided slot gets replaced with the given item, and the preoccupied element of the slot gets displaced into the other hash table. The process continues until every key has its own spot in the empty buckets of the tables; if the procedure enters into infinite loop—which is identified through maintaining a threshold loop counter—both hash tables get rehashed with newer hash functions and the procedure continues.[32]:124–125

Hopscotch hashing

Hopscotch hashing is an open addressing based algorithm which combines the elements of cuckoo hashing, linear probing and chaining through the notion of a *neighbourhood* of buckets—the subsequent buckets around any given occupied bucket, also called a "virtual" bucket.[33]:351–352 The algorithm is designed to deliver better performance when the load factor of the hash table grows beyond 90%; it also provides high throughput in concurrent settings, thus well suited for implementing resizable concurrent hash table.[33]:350 The neighbourhood characteristic of hopscotch hashing guarantees a property that, the cost of finding the desired item from any given buckets within the neighbourhood is very close to the cost of finding it in the bucket itself; the algorithm attempts to be an item into its neighbourhood—with a possible cost involved in displacing other items.[33]:352

Each bucket within the hash table includes an additional "hop-information"—an H-bit bit array for indicating the relative distance of the item which was originally hashed into the current virtual bucket within *H* – 1 entries.[33]:352 Let k and Bk be the key to be inserted and bucket to which the key is hashed into respectively; several cases are involved in the insertion procedure such that the neighbourhood property of the algorithm is vowed:[33]:352–353 if Bk is empty, the element is inserted, and the leftmost bit of bitmap is set to 1; if not empty, linear probing is used for finding an empty slot in the table, the bitmap of the bucket gets updated followed by the insertion; if the empty slot is not within the range of the *neighbourhood,* i.e. *H* – 1, subsequent swap and hop-info bit array manipulation of each bucket is performed in accordance with its neighbourhood invariant properties.[33]:353

Robin Hood hashing

Robin Hood hashing is an open addressing based collision resolution algorithm; the collisions are resolved through favouring the displacement of the element that is farthest—or longest *probe sequence length* (PSL)—from its "home location" i.e. the bucket to which the item was hashed into.[34]:12 Although Robin Hood hashing does not change the theoretical search cost, it significantly affects the variance of the distribution of the items on the buckets,[35]:2 i.e. dealing with cluster formation in the hash table.[36] Each node within the hash table that uses Robin Hood hashing should be augmented to store an extra PSL value.[37] Let x be the key to be inserted, $x.\mathrm{psl}$ be the (incremental) PSL length of x, T be the hash table and j be the index, the insertion procedure is as follows:[34]:12–13[38]:5

- If $x.\mathrm{psl}\ \le\ T[j].\mathrm{psl}$: the iteration goes into the next bucket without attempting an external probe.
- If $x.\mathrm{psl}\ >\ T[j].\mathrm{psl}$: insert the item x into the bucket j; swap x with $T[j]$—let it be x'; continue the probe from the $(j+1)$th bucket to insert x'; repeat the procedure until every element is inserted.

Dynamic resizing

Repeated insertions cause the number of entries in a hash table to grow, which consequently increases the load factor; to maintain the amortized $O(1)$ performance of the lookup and insertion operations, a hash table is dynamically resized and the items of the tables are *rehashed* into the buckets of the new hash table,[10] since the items cannot be copied over as varying table sizes results in different hash value due to modulo operation.[39] If a hash table becomes "too empty" after deleting some elements, resizing may be performed to avoid excessive memory usage.[40]

Resizing by moving all entries

Generally, a new hash table with a size double that of the original hash table gets allocated privately and every item in the original hash table gets moved to the newly allocated one by computing the hash values of the items followed by the insertion operation. Rehashing is simple, but computationally expensive.[41]:478–479

Alternatives to all-at-once rehashing

Some hash table implementations, notably in real-time systems, cannot pay the price of enlarging the hash table all at once, because it may interrupt time-critical operations. If one cannot avoid dynamic resizing, a solution is to perform the resizing gradually to avoid storage blip—typically at 50% of new table's size—during rehashing and to avoid memory fragmentation that triggers heap compaction due to deallocation of large memory blocks caused by the old hash table.[42]:2–3 In such case, the rehashing operation is done incrementally through extending prior memory block allocated for the old hash table such that the buckets of the hash table remain unaltered. A common approach for amortized rehashing involves maintaining two hash functions h_{old} and h_{new}. The process of rehashing a bucket's items in accordance with the new hash function is termed as *cleaning*, which is implemented through command pattern by encapsulating the operations such as $\mathrm{Add}(\mathrm{key})$, $\mathrm{Get}(\mathrm{key})$ and $\mathrm{Delete}(\mathrm{key})$ through a $\mathrm{Lookup}(\mathrm{key}, \mathrm{command})$ wrapper such that each element in the bucket gets rehashed and its procedure involve as follows:[42]:3

- Clean $\mathrm{Table}[h_{\text{old}}(\mathrm{key})]$ bucket.
- Clean $\mathrm{Table}[h_{\text{new}}(\mathrm{key})]$ bucket.
- The *command* gets executed.

Linear hashing

Linear hashing is an implementation of the hash table which enables dynamic growths or shrinks of the table one bucket at a time.[43]

Performance

The performance of a hash table is dependent on the hash function's ability in generating quasi-random numbers (σ) for entries in the hash table where K, n and $h(x)$ denotes the key, number of buckets and the hash function such that $\sigma = h(K)\ \%\ n$. If the hash function generates the same σ for distinct keys ($K_1 \neq K_2,\ h(K_1)\ =\ h(K_2)$), this results in *collision*, which is dealt with in a variety of ways. The constant time complexity ($O(1)$) of the operation in a hash table is presupposed on the condition that the hash function doesn't generate colliding indices; thus, the performance of the hash table is directly proportional to the chosen hash function's ability to disperse the indices.[44]:1 However, construction of such a hash function is practically infeasible, that being so, implementations depend on case-specific collision resolution techniques in achieving higher performance.[44]:2

Applications

Associative arrays

Hash tables are commonly used to implement many types of in-memory tables. They are used to implement associative arrays.[30]

Database indexing

Hash tables may also be used as disk-based data structures and database indices (such as in dbm) although B-trees are more popular in these applications.[45]

Caches

Hash tables can be used to implement caches, auxiliary data tables that are used to speed up the access to data that is primarily stored in slower media. In this application, hash collisions can be handled by discarding one of the two colliding entries—usually erasing the old item that is currently stored in the table and overwriting it with the new item, so every item in the table has a unique hash value.[46][47]

Sets

Hash tables can be used in the implementation of set data structure, which can store unique values without any particular order; set is typically used in testing the membership of a value in the collection, rather than element retrieval.[48]

Transposition table

A transposition table to a complex Hash Table which stores information about each section that has been searched.[49]

Implementations

Many programming languages provide hash table functionality, either as built-in associative arrays or as standard library modules.

- In JavaScript, an "object" is a mutable collection of key-value pairs (called "properties"), where each key is either a string or a guaranteed-unique "symbol"; any other value, when used as a key, is first coerced to a string. Aside from the seven "primitive" data types, every value in JavaScript is an object.[50] ECMAScript 2015 also added the `Map` data structure, which accepts arbitrary values as keys.[51]
- C++11 includes `unordered_map` in its standard library for storing keys and values of arbitrary types.[52]
- Go's built-in `map` implements a hash table in the form of a type.[53]
- Java programming language includes the `HashSet`, `HashMap`, `LinkedHashSet`, and `LinkedHashMap` generic collections.[54]
- Python's built-in `dict` implements a hash table in the form of a type.[55]
- Ruby's built-in `Hash` uses the open addressing model from Ruby 2.4 onwards.[56]
- Rust programming language includes `HashMap`, `HashSet` as part of the Rust Standard Library.[57]
- The .NET standard library includes `HashSet` and `Dictionary`,[58][59] so it can be used from languages such as C# and VB.NET.[60]

See also

- Bloom filter
- Consistent hashing
- Distributed hash table
- Extendible hashing

- Hash array mapped trie
- Lazy deletion
- Pearson hashing
- PhotoDNA
- Rabin–Karp string search algorithm
- Search data structure
- Stable hashing
- Succinct hash table

References

1. Martin Farach-Colton; Andrew Krapivin; William Kuszmaul. *Optimal Bounds for Open Addressing Without Reordering*. 2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS). arXiv:2501.02305 (https://arxiv.org/abs/2501.02305) . doi:10.1109/FOCS61266.2024.00045 (https://doi.org/10.1109%2FFOCS61266.2024.00045) .
2. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009). *Introduction to Algorithms* (3rd ed.). Massachusetts Institute of Technology. pp. 253–280. ISBN 978-0-262-03384-8.
3. Mehlhorn, Kurt; Sanders, Peter (2008). "Hash Tables and Associative Arrays" (https://people.mpi-inf.mpg.de/~mehlhorn/ftp/Toolbox/HashTables.pdf) (PDF). *Algorithms and Data Structures*. Springer. pp. 81–98. doi:10.1007/978-3-540-77978-0_4 (https://doi.org/10.1007%2F978-3-540-77978-0_4) . ISBN 978-3-540-77977-3.
4. Leiserson, Charles E. (Fall 2005). "Lecture 13: Amortized Algorithms, Table Doubling, Potential Method" (http://videolectures.net/mit6046jf05_leiserson_lec13/) . *course MIT 6.046J/18.410J Introduction to Algorithms*. Archived (https://web.archive.org/web/20090807022046/http://videolectures.net/mit6046jf05_leiserson_lec13/) from the original on August 7, 2009.
5. Knuth, Donald (1998). *The Art of Computer Programming*. Vol. 3: *Sorting and Searching* (2nd ed.). Addison-Wesley. pp. 513–558. ISBN 978-0-201-89685-5.

6. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Chapter 11: Hash Tables". *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. pp. 221 (https://archive.org/details/introductiontoal00corm_691/page/n243) –252. ISBN 978-0-262-53196-2.

7. Sedgewick, Robert; Wayne, Kevin (2011). *Algorithms* (https://algs4.cs.princeton.edu/) . Vol. 1 (4 ed.). Addison-Wesley Professional – via Princeton University, Department of Computer Science.

8. Konheim, Alan G. (2010). *Hashing in Computer Science*. doi:10.1002/9780470630617 (https://doi.org/10.1002%2F9780470630617) . ISBN 978-0-470-34473-6.

9. Mehta, Dinesh P.; Mehta, Dinesh P.; Sahni, Sartaj, eds. (2004). *Handbook of Data Structures and Applications*. doi:10.1201/9781420035179 (https://doi.org/10.1201%2F9781420035179) . ISBN 978-0-429-14701-2.

10. Mayers, Andrew (2008). "CS 312: Hash tables and amortized analysis" (https://www.cs.cornell.edu/courses/cs312/2008sp/lectures/lec20.html) . Cornell University, Department of Computer Science. Archived (https://web.archive.org/web/20210426052033/http://www.cs.cornell.edu/courses/cs312/2008sp/lectures/lec20.html) from the original on April 26, 2021. Retrieved October 26, 2021 – via cs.cornell.edu.

11. James S. Plank and Brad Vander Zanden. "CS140 Lecture notes -- Hashing" (http://web.eecs.utk.edu/~bvanderz/teaching/cs140Sp15/Notes/Hashing/) .

12. Maurer, W. D.; Lewis, T. G. (March 1975). "Hash Table Methods". *ACM Computing Surveys*. **7** (1): 5–19. doi:10.1145/356643.356645 (https://doi.org/10.1145%2F356643.356645) . S2CID 17874775 (https://api.semanticscholar.org/CorpusID:17874775) .

13. Owolabi, Olumide (February 2003). "Empirical studies of some hashing functions". *Information and Software Technology*. **45** (2): 109–112. doi:10.1016/S0950-5849(02)00174-X (https://doi.org/10.1016%2FS0950-5849%2802%2900174-X) .

14. Lu, Yi; Prabhakar, Balaji; Bonomi, Flavio (2006). *Perfect Hashing for Network Applications*. 2006 IEEE International Symposium on Information Theory. pp. 2774–2778. doi:10.1109/ISIT.2006.261567 (https://doi.org/10.1109%2FISIT.2006.261567) . ISBN 1-4244-0505-X. S2CID 1494710 (https://api.semanticscholar.org/CorpusID:1494710) .

15. Belazzougui, Djamal; Botelho, Fabiano C.; Dietzfelbinger, Martin (2009). "Hash, displace, and compress" (http://cmph.sourceforge.net/papers/esa09.pdf) (PDF). *Algorithms—ESA 2009: 17th Annual European Symposium, Copenhagen, Denmark, September 7–9, 2009, Proceedings*. Lecture Notes in Computer Science. Vol. 5757. Berlin: Springer. pp. 682–693. CiteSeerX 10.1.1.568.130 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.568.130) . doi:10.1007/978-3-642-04128-0_61 (https://doi.org/10.1007%2F978-3-642-04128-0_61) . MR 2557794 (https://mathscinet.ams.org/mathscinet-getitem?mr=2557794) .

16. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Chapter 11: Hash Tables". *Introduction to Algorithms* (2nd ed.). Massachusetts Institute of Technology. ISBN 978-0-262-53196-2.

17. Pearson, Karl (1900). "On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling" (https://zenodo.org/record/1430618) . *Philosophical Magazine*. Series 5. **50** (302): 157–175. doi:10.1080/14786440009463897 (https://doi.org/10.1080%2F14786440009463897) .

18. Plackett, Robin (1983). "Karl Pearson and the Chi-Squared Test". *International Statistical Review*. **51** (1): 59–72. doi:10.2307/1402731 (https://doi.org/10.2307%2F1402731) . JSTOR 1402731 (https://www.jstor.org/stable/1402731) .

19. Wang, Thomas (March 1997). "Prime Double Hash Table" (https://web.archive.org/web/19990903133921/http://www.concentric.net/~Ttwang/tech/primehash.htm) . Archived from the original (https://www.concentric.net/~Ttwang/tech/primehash.htm) on September 3, 1999. Retrieved May 10, 2015.

20. Wegman, Mark N.; Carter, J.Lawrence (June 1981). "New hash functions and their use in authentication and set equality" (https://doi.org/10.1016%2F0022-0000%2881%2990033-7) . *Journal of Computer and System Sciences*. **22** (3): 265–279. doi:10.1016/0022-0000(81)90033-7 (https://doi.org/10.1016%2F0022-0000%2881%2990033-7) .

21. Donald E. Knuth (April 24, 1998). *The Art of Computer Programming: Volume 3: Sorting and Searching* (https://dl.acm.org/doi/10.5555/280635) . Addison-Wesley Professional. ISBN 978-0-201-89685-5.

22. Demaine, Erik; Lind, Jeff (Spring 2003). "Lecture 2" (http://courses.csail.mit.edu/6.897/spring03/scribe_notes/L2/lecture2.pdf) (PDF). *6.897: Advanced Data Structures. MIT Computer Science and Artificial Intelligence Laboratory*. Archived (https://web.archive.org/web/20100615203901/http://courses.csail.mit.edu/6.897/spring03/scribe_notes/L2/lecture2.pdf) (PDF) from the original on June 15, 2010. Retrieved June 30, 2008.

23. Culpepper, J. Shane; Moffat, Alistair (2005). "Enhanced Byte Codes with Restricted Prefix Properties". *String Processing and Information Retrieval*. Lecture Notes in Computer Science. Vol. 3772. pp. 1–12. doi:10.1007/11575832_1 (https://doi.org/10.1007%2F11575832_1) . ISBN 978-3-540-29740-6.

24. Willard, Dan E. (2000). "Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree". *SIAM Journal on Computing*. **29** (3): 1030–1049. doi:10.1137/S0097539797322425 (https://doi.org/10.1137%2FS0097539797322425) . MR 1740562 (https://mathscinet.ams.org/mathscinet-getitem?mr=1740562) ..

25. Askitis, Nikolas; Sinha, Ranjan (October 2010). "Engineering scalable, cache and space efficient tries for strings". *The VLDB Journal*. **19** (5): 633–660. doi:10.1007/s00778-010-0183-9 (https://doi.org/10.1007%2Fs00778-010-0183-9) .

26. Askitis, Nikolas; Zobel, Justin (October 2005). "Cache-conscious Collision Resolution in String Hash Tables". *Proceedings of the 12th International Conference, String Processing and Information Retrieval (SPIRE 2005)*. Vol. 3772/2005. pp. 91–102. doi:10.1007/11575832_11 (https://doi.org/10.1007%2F11575832_11) . ISBN 978-3-540-29740-6.

27. Askitis, Nikolas (2009). "Fast and Compact Hash Tables for Integer Keys" (https://web.archive.org/web/20110216180225/http://crpit.com/confpapers/CRPITV91Askitis.pdf) (PDF). *Proceedings of the 32nd Australasian Computer Science Conference (ACSC 2009)*. Vol. 91. pp. 113–122. ISBN 978-1-920682-72-9. Archived from the original (http://crpit.com/confpapers/CRPITV91Askitis.pdf) (PDF) on February 16, 2011. Retrieved June 13, 2010.

28. Tenenbaum, Aaron M.; Langsam, Yedidyah; Augenstein, Moshe J. (1990). *Data Structures Using C*. Prentice Hall. pp. 456–461, p. 472. ISBN 978-0-13-199746-2.

29. Pagh, Rasmus; Rodler, Flemming Friche (2001). "Cuckoo Hashing". *Algorithms — ESA 2001*. Lecture Notes in Computer Science. Vol. 2161. pp. 121–133. CiteSeerX 10.1.1.25.4189 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.4189) . doi:10.1007/3-540-44676-1_10 (https://doi.org/10.1007%2F3-540-44676-1_10) . ISBN 978-3-540-42493-2.

30. 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.

31. Vitter, Jeffery S.; Chen, Wen-Chin (1987). *The design and analysis of coalesced hashing* (https://archive.org/details/designanalysisof0000vitt/) . New York, United States: Oxford University Press. ISBN 978-0-19-504182-8 – via Archive.org.

32. Pagh, Rasmus; Rodler, Flemming Friche (2001). "Cuckoo Hashing". *Algorithms — ESA 2001*. Lecture Notes in Computer Science. Vol. 2161. pp. 121–133. CiteSeerX 10.1.1.25.4189 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.4189) . doi:10.1007/3-540-44676-1_10 (https://doi.org/10.1007%2F3-540-44676-1_10) . ISBN 978-3-540-42493-2.

33. Herlihy, Maurice; Shavit, Nir; Tzafrir, Moran (2008). "Hopscotch Hashing". *Distributed Computing*. Lecture Notes in Computer Science. Vol. 5218. pp. 350–364. doi:10.1007/978-3-540-87779-0_24 (https://doi.org/10.1007%2F978-3-540-87779-0_24) . ISBN 978-3-540-87778-3.

34. Celis, Pedro (1986). *Robin Hood Hashing* (https://cs.uwaterloo.ca/research/tr/1986/CS-86-14.pdf) (PDF). Ontario, Canada: University of Waterloo, Dept. of Computer Science. ISBN 978-0-315-29700-5. OCLC 14083698 (https://search.worldcat.org/oclc/14083698) . Archived (https://web.archive.org/web/20211101071032/https://cs.uwaterloo.ca/research/tr/1986/CS-86-14.pdf) (PDF) from the original on November 1, 2021. Retrieved November 2, 2021.

35. Poblete, P. V.; Viola, A. (July 2019). "Analysis of Robin Hood and Other Hashing Algorithms Under the Random Probing Model, With and Without Deletions". *Combinatorics, Probability and Computing*. **28** (4): 600–617. doi:10.1017/S0963548318000408 (https://doi.org/10.1017%2FS0963548318000408) . S2CID 125374363 (https://api.semanticscholar.org/CorpusID:125374363) .

36. Clarkson, Michael (2014). "Lecture 13: Hash tables" (https://www.cs.cornell.edu/courses/cs3110/2014fa/lectures/13/lec13.html) . Cornell University, Department of Computer Science. Archived (https://web.archive.org/web/20211007011300/https://www.cs.cornell.edu/courses/cs3110/2014fa/lectures/13/lec13.html) from the original on October 7, 2021. Retrieved November 1, 2021 – via cs.cornell.edu.

37. Gries, David (2017). "JavaHyperText and Data Structure: Robin Hood Hashing" (https://www.cs.cornell.edu/courses/JavaAndDS/files/hashing_RobinHood.pdf) (PDF). Cornell University, Department of Computer Science. Archived (https://web.archive.org/web/20210426051503/http://www.cs.cornell.edu/courses/JavaAndDS/files/hashing_RobinHood.pdf) (PDF) from the original on April 26, 2021. Retrieved November 2, 2021 – via cs.cornell.edu.

38. Celis, Pedro (March 28, 1988). *External Robin Hood Hashing* (https://legacy.cs.indiana.edu/ftp/techreports/TR246.pdf) (PDF) (Technical report). Bloomington, Indiana: Indiana University, Department of Computer Science. 246. Archived (https://web.archive.org/web/20211103013505/https://legacy.cs.indiana.edu/ftp/techreports/TR246.pdf) (PDF) from the original on November 3, 2021. Retrieved November 2, 2021.

39. Goddard, Wayne (2021). "Chapter C5: Hash Tables" (https://people.computing.clemson.edu/~goddard/texts/algor/C5.pdf) (PDF). Clemson University. pp. 15–16. Retrieved December 4, 2023.

40. Devadas, Srini; Demaine, Erik (February 25, 2011). "Intro to Algorithms: Resizing Hash Tables" (https://courses.csail.mit.edu/6.006/spring11/rec/rec07.pdf) (PDF). Massachusetts Institute of Technology, Department of Computer Science. Archived (https://web.archive.org/web/20210507102944/https://courses.csail.mit.edu/6.006/spring11/rec/rec07.pdf) (PDF) from the original on May 7, 2021. Retrieved November 9, 2021 – via MIT OpenCourseWare.

41. Thareja, Reema (2014). "Hashing and Collision". *Data Structures Using C*. Oxford University Press. pp. 464–488. ISBN 978-0-19-809930-7.

42. Friedman, Scott; Krishnan, Anand; Leidefrost, Nicholas (March 18, 2003). "Hash Tables for Embedded and Real-time systems" (https://users.cs.northwestern.edu/~sef318/docs/hashtables.pdf) (PDF). *All Computer Science and Engineering Research*. Washington University in St. Louis. doi:10.7936/K7WD3XXV (https://doi.org/10.7936%2FK7WD3XXV). Archived (https://web.archive.org/web/20210609163643/https://users.cs.northwestern.edu/~sef318/docs/hashtables.pdf) (PDF) from the original on June 9, 2021. Retrieved November 9, 2021 – via Northwestern University, Department of Computer Science.

43. Litwin, Witold (1980). "Linear hashing: A new tool for file and table addressing" (https://www.cs.cmu.edu/afs/cs.cmu.edu/user/christos/www/courses/826-resources/PAPERS+BOOK/linear-hashing.PDF) (PDF). *Proc. 6th Conference on Very Large Databases*. Carnegie Mellon University. pp. 212–223. Archived (https://web.archive.org/web/20210506233325/http://www.cs.cmu.edu/afs/cs.cmu.edu/user/christos/www/courses/826-resources/PAPERS+BOOK/linear-hashing.PDF) (PDF) from the original on May 6, 2021. Retrieved November 10, 2021 – via cs.cmu.edu.

44. Dijk, Tom Van (2010). "Analysing and Improving Hash Table Performance" (https://www.tvandijk.nl/pdf/bscthesis.pdf) (PDF). Netherlands: University of Twente. Archived (https://web.archive.org/web/20211106094558/http://www.tvandijk.nl/pdf/bscthesis.pdf) (PDF) from the original on November 6, 2021. Retrieved December 31, 2021.

45. Lech Banachowski. "Indexes and external sorting" (https://ghostarchive.org/archive/HW0hp). pl:Polsko-Japońska Akademia Technik Komputerowych. Archived from the original (https://edux.pjwstk.edu.pl/mat/262/lec/rW9.htm) on March 26, 2022. Retrieved March 26, 2022.

46. Zhong, Liang; Zheng, Xueqian; Liu, Yong; Wang, Mengting; Cao, Yang (February 2020). "Cache hit ratio maximization in device-to-device communications overlaying cellular networks". *China Communications*. **17** (2): 232–238. doi:10.23919/jcc.2020.02.018 (https://doi.org/10.23919%2Fjcc.2020.02.018) . S2CID 212649328 (https://api.semanticscholar.org/CorpusID:212649328) .

47. Bottommley, James (January 1, 2004). "Understanding Caching" (https://www.linuxjournal.com/article/7105) . Linux Journal. Archived (https://web.archive.org/web/20201204195114/https://www.linuxjournal.com/article/7105) from the original on December 4, 2020. Retrieved April 16, 2022.

48. Jill Seaman (2014). "Set & Hash Tables" (https://web.archive.org/web/20220401134706/https://userweb.cs.txstate.edu/~js236/201412/cs5301/week13.pdf) (PDF). Texas State University. Archived from the original on April 1, 2022. Retrieved March 26, 2022.

49. "Transposition Table - Chessprogramming wiki" (https://www.chessprogramming.org/Transposition_Table) . *chessprogramming.org*. Archived (https://web.archive.org/web/20210214110941/https://www.chessprogramming.org/Transposition_Table) from the original on February 14, 2021. Retrieved May 1, 2020.

50. "JavaScript data types and data structures - JavaScript | MDN" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#objects) . *developer.mozilla.org*. Retrieved July 24, 2022.

51. "Map - JavaScript | MDN" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map) . *developer.mozilla.org*. June 20, 2023. Retrieved July 15, 2023.

52. "Programming language C++ - Technical Specification" (https://web.archive.org/web/20220121061142/http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3690.pdf) (PDF). International Organization for Standardization. pp. 812–813. Archived from the original (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf) (PDF) on January 21, 2022. Retrieved February 8, 2022.

53. "The Go Programming Language Specification" (https://go.dev/ref/spec#Map_types) . *go.dev*. Retrieved January 1, 2023.

54. "Lesson: Implementations (The Java™ Tutorials > Collections)" (https://docs.oracle.com/javase/tutorial/collections/implementations/index.html) . *docs.oracle.com*. Archived (https://web.archive.org/web/20170118041252/https://docs.oracle.com/javase/tutorial/collections/implementations/index.html) from the original on January 18, 2017. Retrieved April 27, 2018.

55. Zhang, Juan; Jia, Yunwei (2020). "Redis rehash optimization based on machine learning" (https://doi.org/10.1088%2F1742-6596%2F1453%2F1%2F012048) . *Journal of Physics: Conference Series*. **1453** (1): 3. Bibcode:2020JPhCS1453a2048Z (https://ui.adsabs.harvard.edu/abs/2020JPhCS1453a2048Z) . doi:10.1088/1742-6596/1453/1/012048 (https://doi.org/10.1088%2F1742-6596%2F1453%2F1%2F012048) . S2CID 215943738 (https://api.semanticscholar.org/CorpusID:215943738) .

56. Jonan Scheffler (December 25, 2016). "Ruby 2.4 Released: Faster Hashes, Unified Integers and Better Rounding" (https://blog.heroku.com/ruby-2-4-features-hashes-integers-rounding#hash-changes) . *heroku.com*. Archived (https://web.archive.org/web/20190703145530/https://blog.heroku.com/ruby-2-4-features-hashes-integers-rounding#hash-changes) from the original on July 3, 2019. Retrieved July 3, 2019.

57. "doc.rust-lang.org" (https://doc.rust-lang.org/std/index.html) . Archived (https://web.archive.org/web/20221208155205/https://doc.rust-lang.org/std/index.html) from the original on December 8, 2022. Retrieved December 14, 2022.

58. "HashSet Class (System.Collections.Generic)" (https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1?view=net-7.0) . *learn.microsoft.com*. Retrieved July 1, 2023.

59. dotnet-bot. "Dictionary Class (System.Collections.Generic)" (https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-8.0) . *learn.microsoft.com*. Retrieved January 16, 2024.

60. "VB.NET HashSet Example" (https://www.dotnetperls.com/hashset-vbnet) . *Dot Net Perls*.

Further reading

- Tamassia, Roberto; Goodrich, Michael T. (2006). "Chapter Nine: Maps and Dictionaries". *Data structures and algorithms in Java : [updated for Java 5.0]* (https://archive.org/details/datastructuresal00good_183) (4th ed.). Hoboken, NJ: Wiley. pp. 369 (https://archive.org/details/datastructuresal00good_183/page/n368) –418. ISBN 978-0-471-73884-8.

- McKenzie, B. J.; Harries, R.; Bell, T. (February 1990). "Selecting a hashing algorithm". *Software: Practice and Experience*. **20** (2): 209–224. doi:10.1002/spe.4380200207 (https://doi.org/10.1002%2Fspe.4380200207) . hdl:10092/9691 (https://hdl.handle.net/10092%2F9691) . S2CID 12854386 (https://api.semanticscholar.org/CorpusID:12854386) .

External links

- NIST entry on hash tables (https://xlinux.nist.gov/dads/HTML/hashtab.html)
- Open Data Structures – Chapter 5 – Hash Tables (http://opendatastructures.org/versions/edition-0.1e/ods-java/5_Hash_Tables.html) , Pat Morin
- MIT's Introduction to Algorithms: Hashing 1 (http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-7-hashing-hash-functions/) MIT OCW lecture Video
- MIT's Introduction to Algorithms: Hashing 2 (http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-8-universal-hashing-perfect-hashing/) MIT OCW lecture Video