Hashing

Martin Babka

January 12, 2011

Hashing, Universal hashing, Perfect hashing

Hashing

- Input data is uniformly distributed.
- A dynamic set is stored.

Universal hashing

- Randomised algorithm – uniform choice of a hash function from a universal system. No probabilistic assumptions on the input data.
- A dynamic set is stored.

Perfect hashing

- Stores a static set, but a dynamic variant is possible.
- Guarantees a constant look-up time.

Separate Chaining

Notation

- U - the universe.
- $S \subset U$ - the stored set.
- B - the set of buckets of the hash table.
- $n = |S|$, $m = |B|$, $\alpha = \frac{n}{m}$ - the load factor.

Common hashing

- Uses a single hash function which may create collisions.
- Buckets of colliding elements are usually represented by singly linked lists.
- Table is resized when the load factor is out of the prescribed bounds.

Separate Chaining

Estimates

- Expected length of the chain is α.
- Expected maximal length of the chain when $\alpha \leq 1$ is

$$O\left(\frac{\log n}{\log \log n}\right).$$

- Estimates assume uniform distribution of the input data.

Disadvantages and Advantages

+ Simple to implement and analyse.
+ Predictable behaviour for high load factors.
- Relatively slow. Lacks good cache behaviour.
- Relatively high memory consumption.

Separate Chaining

Other methods for Separate Chaining without explicit linked lists

- Do not store the chains in external linked lists. The buckets of colliding elements are created within the hash table instead.
- Hashing with relocations – uses singly linked lists. Chain may start on a different position than the one given by the hash function.
- Hashing with two pointers – uses doubly linked lists.
- The same lengths – as the common variant.
- Better cache behaviour.

Open Addressing

Common features

- Chains of colliding elements with different hash values merge together.
- Methods do not require additional memory for storing chains. Chains are stored within the hash table.
- The next element in the chain is determined implicitly by another hash function.

Algorithm

- Hash function in the form $h(x, i)$ – hash value of the element x when it is at the i-th position in the chain.
- Iterates the chain until an empty position is found.
- Problematic deletion, false delete.

Open Addressing

Double Hashing

Double hashing

- $h(x, i) = h_1(x) + ih_2(x)$.
- Needs two hash functions h_1 and possibly h_2.
- Results in a slower computation of the hash value $h(x, i)$.

Expected running times

- Number of comparisons in the unsuccessful case is $\frac{1}{1-\alpha}$.
- Number of comparisons in the successful case is $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$.
- Good behaviour for lower load factors. Still reasonable for $\alpha \leq 0.9$.

Linear Probing, Quadratic Probing

Linear probing

- $h(x, i) = h_1(x) + i$.
- May be in the form $h(x, i) = h_1(x) + ci$ but usually $c = 1$.
- Good cache behaviour. This method only iterates the table starting at the position $h_1(x)$.
- Usable with low load factors, suitable for $\alpha \leq 0.75$.
- Number of tests in the unsuccessful case is $\frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right)$.
- Number of tests in the successful case is $\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$.

Quadratic probing

- $h(x, i) = h_1(x) + bi + ai^2$ for $a \neq 0$.

Robin-Hood Hashing

The method

- Variation of linear probing.
- With every stored element we store its position in the chain. This information is stored in each non-empty slot of the table.
- Expected length of the probing sequence is $\frac{1}{\alpha}\ln(1-\alpha)$, the variance is constant.
- The find operation may be improved so that it runs in expected constant time (independently on α).

Robin-Hood Hashing, Insert

Insert

- The new element x should be inserted in the i-th position in its chain.
- Assume that the position $h(x, i)$ is occupied by an element y stored at the j-th position in its chain.
- If $j < i$, then swap the two elements x and y store x and continue with y. Otherwise continue with x.
- $i < j < \mathbf{E}\left[psl\right]$ – using x decreases the variance more than using y.
- $i < \mathbf{E}\left[psl\right] < j$ – using x decreases the variance, using y increases it.
- $\mathbf{E}\left[psl\right] < i < j$ – moving x increases the variance less than using y.

Robin-Hood Hashing, LCFS, FCFS

- Another modification of linear probing is *LCFS* (last come first served)
- Again, only insertion is modified.
- If the position, where the new element x should be placed, is occupied by an element y, then store x at the position and continue with the element y.
- Common linear probing is FCFS (first come first served).
- Both Robin-Hood and LCFS decrease the variance of the length of the probing sequence.

Open Addressing

Hopscotch hashing – Introduction

Hopscotch hashing

- Improvement of linear probing.
- Superb cache behaviour – specifically designed to be suitable for cache.
- Choose a constant H (usually the size of the word).
- Element may be placed only $H - 1$ buckets far from its hash value.
- Every bucket i contains a bitmap of H bits indicating at which positions are the elements with the hash value i.

Algorithms

- Find performs at most H comparisons.

Open Addressing

Hopscotch hashing – Algorithms

Insert

- Let $i = h(x)$. Find the nearest empty position j.
- If j is at most $H - 1$ buckets far from i, insert x in the bucket with the address j.
- Otherwise find an element y such that $i \leq h(y) \leq j$, $j \leq h(y) + H - 1$ and y is placed before j. Place y at j and use the freed bucket.
- If any place near i can not be freed, then the table is resized and rehashed.

Delete

- Remove the element, update the bitmap in the bucket at the element's hash value.

Open Addressing

Hopscotch hashing – Algorithms

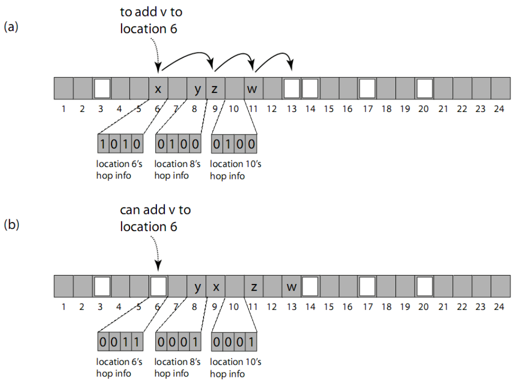


Fig. 1. The blank entries are empty, all others contain items. Here, H is 4. In part (a), we add item v with hash value 6. A linear probe finds entry 13 is empty. Because 13 is more than 4 entries away from 6, we look for an earlier entry to swap with 13. The first place to look is $H - 1 = 3$ entries before, at entry 10. That entry's hop information bit-map indicates that w at entry 11 can be displaced to 13, which we do. Entry 11 is still too far from entry 6, so we examine entry 8. The hop information bit-map indicates that z at entry 9 can be moved to entry 11. Finally, x at entry is moved to entry 9. Part (b) shows the table state just before adding v.

Open Addressing

Hopscotch hashing – Analysis

Analysis

- Probability of failure (rehashing the table when inserting) is less than $\frac{1}{H!}$.
- Expected running time of Insert is constant, the expected number of elements in a bucket is $1 + \frac{e^{2\alpha} - 1 - 2\alpha}{4}$.
- Parameter H may be chosen as a constant provided that α is upper bounded. For $\alpha = 1$ choice $H = 3$ is sufficient.

Delete

- Remove the element, update the bitmap in the bucket at the element's hash value.

Two-way Hashing

Based on the study of balls and bins systems.

Theorem

Consider placing n balls into n buckets using the following process. For every ball randomly and uniformly choose h, $h \geq 2$, buckets. Put the ball inside the bucket containing the smallest number of elements. After inserting the last ball the probability of having a bucket with more than $\frac{\ln \ln n}{\ln h} + O(1)$ balls is $o(1)$.

Applications

- Load balancing.
- Hashing.

Two-way Hashing

Algorithms

- Use h, $h \geq 2$, functions $f_1, \ldots, f_h$.
- The more functions are used the more tests have to be performed. Use $h = 2$.

Insert

- Consider the buckets at positions $f_1(x), \ldots, f_h(x)$.
- Insert into the bucket with the smallest number of elements.

Find

- Consider the buckets $f_1(x), \ldots, f_h(x)$.
- Seek each of the buckets for the element x.

Two-way Hashing

Analysis, pros and cons

- $+$ Good worst-case performance $O(\ln \ln n)$.
- $+$ Many variants are studied nowadays.
- $+$ Promising experimental results.
- 0 Analysis is not easy but not that difficult.
- - Problematic delete. There are dynamic versions of the previous theorem or use a workaround.
- - Straightforward use of the theorem leads to separate chaining.
- - Usage of better methods combined with two-way chaining may be complicated.

Two-way Hashing with Linear Probing

- Tries to solve the problem of two-way hashing without separate chaining.
- Divide the hash table into blocks. When inserting compute two hash values – they uniquely determine the blocks. Insert into the emptiest block and probe only inside the block.
- Leads to a more complicated but still reasonable implementation.
- Worst case number of tests is at most $\frac{\log\log n + O(1)}{1-\alpha} + 1$ with a high probability $(1 - o(1))$.
- Simple implementations have $\Omega\left(\frac{\log n}{\log\log n}\right)$ worst-case behaviour.

Universal hashing

Probability space

- Multiset of functions $H = \{h : U \Rightarrow B \mid ...\}$ is used.
- Probability space – random uniform choice of a function $h \in H$. No probability assumptions on the input data.

Universal systems

System H of functions is

- c-universal if for $x \neq y \in U$: $\mathbf{Pr}\left(h(x) = h(y)\right) \leq \frac{c}{m}$,
- strongly k-universal if for different $x_1, \dots, x_k \in U$ and for $y_1, \dots, y_k$: $\mathbf{Pr}\left(h(x) = h(y)\right) \leq \frac{1}{m^k}$,
- strongly ω-universal if it is strongly universal for every $k \in \mathbb{N}$.
- There are many various systems and many various estimates on the sizes of the systems.

Universal systems

Properties

- Constant c is usually higher than 1.
- Every strongly $k+1$-universal system is also strongly k-universal.
- Constant c is usually higher than 1.

Systems

- System of polynomials $\sum_{i=1}^{k} a_i x^i \bmod m$ is strongly $k+1$-universal.
- System of all functions is strongly ω-universal.
- System of all linear transformations between vector spaces is 1-universal.

Separate chaining as universal hashing

Properties of universal hashing

- At first choose the hash function uniformly at random from the universal system.
- Do separate chaining with the chosen function.
- Another function may be chosen when rehashing.
- Expected $O(1 + c\alpha)$ running time for c-universal systems.
- Expected maximal length of the chain is hard to analyse. For ω-universal systems it is $O\left(\frac{\log n}{\log \log n}\right)$.
- For the system of linear transformations and $n = m \log m$ it is $O(\log m \log \log m)$. This result is obtained by a rather complicated analysis.

Perfect hashing

Algorithm

- Probabilistic approach to finding a suitable function from a universal system which does not create many collisions.
- The expected number of collisions in universal hashing is $\frac{cn^2}{m}$. Use Markov inequality to obtain a function which creates a small number of collisions.
- For $m = O(n)$ there are many functions which create a linear number of collisions.
- Hence $\sum_{i=1}^{n} n_i^2 = O(n)$ where n_i is the number of colliding elements in the i-th bucket.
- For $m = O(n^2)$ there are many function which do not create collisions at all.
- First use $m = O(n)$. Then for representing elements in the bucket i choose $m_i = O(n_i^2)$.

Perfect hashing

Obtained result

- The created hash table is stored in $O(n)$ memory.
- Hash function may be computed in $O(1)$ time.
- Guarantees $O(1)$ look-up time.
- After choosing m_i slightly bigger than for the static version a dynamic version may be obtained.
- The dynamic version achieves $O(1)$ amortised update time and still has $O(1)$ look-up time.
- Other variants and dynamisation techniques are known.
- Used in theory more than in practice. Real-time applications that need $O(1)$ look-up time are possible.

Cuckoo Hashing

Cuckoo hashing

- Uses two tables with the same size $(1 + \epsilon)n$ for $\epsilon > 0$. Thus $\alpha < 0.5$.
- We use two hash functions chosen from a strongly $O(\log n)$-universal system. One function for each of the tables.
- There is such a system which can be evaluated in a constant time. Polynomials would need $O(\log n)$ time.
- Eeach element x must be stored at the positions $h_1(x)$ in T_1 or $h_2(x)$ in T_2.

Find, Delete

- Look for the element only at the prescribed positions.
- They require a constant time only.

Cuckoo Hashing

Algorithms, Insert

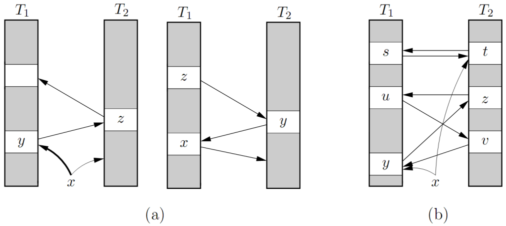


Fig. 1. Examples of Cuckoo Hashing insertion. Arrows show possibilities for moving keys. (a) Key x is successfully inserted by moving keys y and z from one table to the other. (b) Key x cannot be accommodated and a rehash is necessary.

Cuckoo Hashing

Algorithms, Insert

Insert

- Insert places a newly inserted element x in the table T_1 at the position $h_1(x)$. If $h_1(x)$ is already occupied place the former element into T_2. If this position is occupied place the element into T_1 and so on. The maximal number of these iterations is bounded by a value in $O(\log n)$ – the same as the universality of the system.
- Insert is problematic and may fail and need a rehash with choosing another pair of functions.
- Such a failure has a low $O\left(\frac{1}{n^2}\right)$ probability.
- Insert runs in amortised constant time.

Linear Probing as universal hashing

Description

- Needs at least strongly 5-universal system.
- Polynomials of degree 5 are possible but slow to compute.
- Ongoing research of such systems. Tabulation methods.
- Strongly 2-universal systems are not sufficient. There is a set of n elements for which $\Omega(n \log n)$ operations are needed in order to store it in the expected case.
- Slight improvement in the definition of the hash function $h(x, i) = h_1(x) \oplus i$ where h_1 is chosen from a universal system.
- Algorithm is the same as with linear probing.

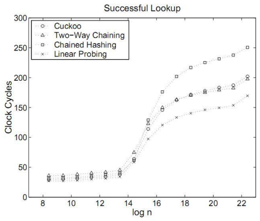
Experimental results

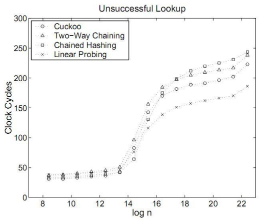
- Some present results show that for low load factors Hopscotch hashing is the best. And it is also secure for higher load factors but the improvement is not so obvious. It should be still the best.
- Cuckoo hashing is regularly beaten by Hopscotch hashing. It is comparable to two-way hashing. Experiments are usually done for $\alpha < 0.5$ since it can not be used with $\alpha > 0.5$.
- When $\alpha \approx 0.5$ cuckoo hashing degrades.

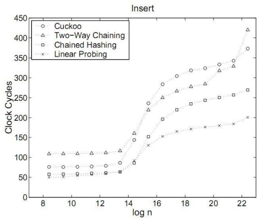
- Experimental comparison of two-way chaining linear probing to other methods is missing. The methods look promising.
- Old methods should be replaced by their newer variants. They give better theoretical results and yet remain simple.

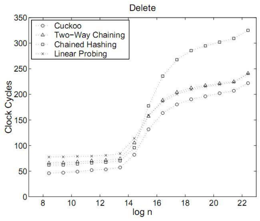
Experimental results

Cuckoo hashing



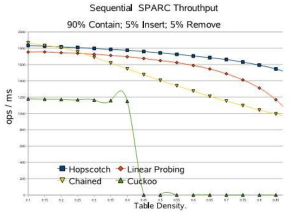






Experimental results

Hopscotch hashing



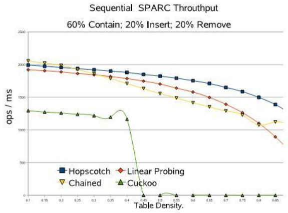
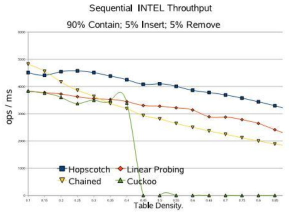


Fig. 4. SPARC throughput as the table density changes.



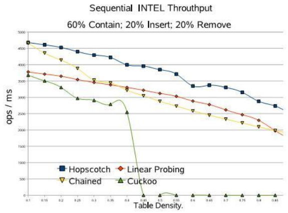


Fig. 5. Intel throughput as the table density changes.

Hash table implementations

Java

Actual implementation

- Uses the most simple separate chaining.
- Predefined maximal load factor is 0.75.
- Does not shrink the table after deletions.
- Predefined maximal size of the table is 2^{30}.

Java 5.0 vs. 6.0

- Timestamp of the last source code change comes from 2006, in 5.0 from 2004.
- Small changes in source code, no algorithmic changes in Java 6.0.

Hash table implementations

C++ - Boost, TR1

Intrusive

- Rather complicated but generic library.
- Generic code allows usage of separate chaining and linear probing.
- Other methods should be possible to use by implementing new traits.

Unordered, TR1

- Interface defined by the TR1 draft assumes usage of separate chaining (iterators invalidation).
- Straightforward implementation.
- Default maximal load factor is 1.0, no shrinking.

Literature

Basic methods

- Mehlhorn, K., Sanders, P.: Data Structures and Algorithms, The Basic Toolbox, Springer (2008)
- Knuth, D. E.: The Art of Computer Programming Volume 3, Addison-Wesley (1997)
- Carter, J. W., Wegman, M. N.: Universal classes of hash functions, STOC '77 (1977).
- Fredman, M., Komlós, J., Szemerédi, E.: Storing a Sparse Table with O(1) Worst Case Access Time, Journal of the ACM (1984)
- Dietzfelbinger, M., Karlin, A., Mehlhorn, K., auf der Heide, F.M., Rohnert, H., Tarjan, R.E.: Dynamic perfect hashing: upper and lower bounds, Foundations of Computer Science (1988)

Literature

Current methods

- Celis, P.: Robin Hood Hashing, Ph.D. thesis, University of Waterloo (1986)
- Herlihy, M., Shavit, N., Tzafrir, M.: Hopscotch Hashing, DISC '08: Proceedings of the 22nd international symposium on Distributed Computing (2008)
- Mitzenmacher M., Upfal, E.: Probability and Computing, Cambridge University Press (2005)
- Malalla, E.: Two-way Hashing with Separate Chaining and Linear Probing, Ph.D. thesis, McGill University (2004)
- Pagh, R., Rodler. F.: Cuckoo Hashing, Journal of Algorithms, 51 (2004)
- Pagh, A., Pagh, R., Ružić, M.: Linear Probing with Constant Independence. SIAM J. Comput. 39, 3 (2009)