Cuckoo Hashing for Undergraduates

Rasmus Pagh
IT University of Copenhagen

March 27, 2006

Abstract

This lecture note presents and analyses two simple hashing algorithms: "Hashing with Chaining", and "Cuckoo Hashing". The analysis uses only very basic (and intuitively understandable) concepts of probability theory, and is meant to be accessible even for undergraduates taking their first algorithms course.

1 Introduction

A *dictionary* is a data structure for storing a set of items (e.g., integers or strings), that supports three basic operations: Lookup(x) which returns `true` if x is in the current set, and `false` otherwise; Insert(x) which adds the item x to the current set if not already present; Delete(x) which removes x from the current set if present. A common variant is to associate some information with each item, which is returned in connection with lookups. It is a simple exercise to augment the algorithms presented here with this capability, but we will describe just the basic variant.

One simple, but not very efficient implementation of a dictionary is a linked list. In this implementation all operations take linear time in the worst case (and even in the average case), assuming that insertions first check whether the item is in the current list. A more scalable implementation of a dictionary is a balanced search tree. In this lecture note we present two even more efficient data structures based on *hashing*. The performance bounds of the algorithms will not hold in the worst case, but be true in the "expected case". This is because the algorithms are not deterministic, but make random choices. The expected performance is the average one over all random choices. More specifically, the behavior of the algorithms will be guided by one, respectively two, *hash functions* that take items as input, and return "random" values in some set $\{1, \ldots, r\}$. To simplify the exposition we make the following assumptions:

- All items to be stored have the same size, and we can compare two items in constant time.

- We have access to hash functions h_1 and h_2 such that any function value $h_i(x)$ is equal to a particular value in $\{1, \ldots, r\}$ with probability[1] $1/r$, and the function values are *independent* of each other (one function value says nothing about other function values). Hash function values can be computed in constant time.
- There is a fixed upper bound n on the number of items in the set.

The space usage of the algorithms will be bounded in terms of n. Specifically, the space usage will be the same as for storing $O(n)$ items, i.e., within a constant factor of the space for storing the items with no space wasted.

We will see that the logarithmic time bounds of balanced search trees can be improved to *constant* expected time per operation. In other words, the time for an operation does not grow even if we fill the dictionary with sets of astronomical size!

2 Hashing with Chaining

The main idea in hashing based dictionaries is to let the hash functions decide where to store each item. An item x will be stored at "position $h_1(x)$" in an array of size $r \geq n$. Note that for this to work, it is crucial that h_1 is really a function, i.e., that $h_1(x)$ is a fixed value. However, it is highly likely that there will be *collisions*, i.e., different items x and y for which $h_1(x) = h_1(y)$. So for each value $a \in \{1, \ldots, r\}$ there is some set S_a of items having this particular value of h_1. An obvious idea is to make a pointer from position a in the array to a data structure holding the set S_a. Such a data structure is called a *bucket*. Surprisingly, it suffices to use a very simple (and inefficient) data structure, namely a linked list (also known as a "chained list"). An instance of the resulting data structure is shown in Figure 1. Intuitively, the reason that linked lists work well is that the sets are very small on average. In the following we will make a precise analysis.

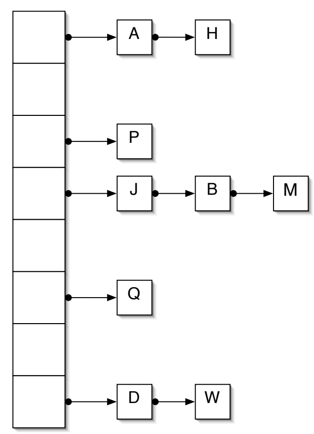


Figure 1. Hashing with chaining. The items J, B and M have the same value of h_1 and have been placed in the same "bucket", a linked list of length 3, starting at position $h_1(J)$ in the array.

We start out with two observations that will go into the analysis:

[1] Note that this is only possible if hash functions are somehow chosen in a random fashion. However, in this lecture note we will not describe how to do this.

1. For any two distinct items x and y, the probability that x hashes to the bucket of y is $O(1/r)$. This follows from our assumptions on h_1.

2. The time for an operation on an item x is bounded by some constant times the number of items in the bucket of x.

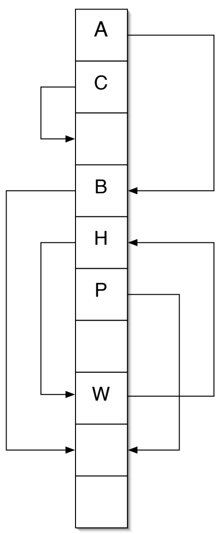
The whole analysis will rest only on these observations. This will allow us to reuse the analysis for Cuckoo Hashing, where the same observations hold for a suitable definition of "bucket".

Let us analyze an operation on item x. By observation 2, we can bound the time by bounding the expected size of the bucket of x. For any operation, it might be the case that x is stored in the data structure when the operation begins, but this can cost only constant time extra, compared to the case in which x is not in the list. Thus, we may assume that the bucket of x contains only items different from x. Let S be the set of items that were present at the beginning of the operation. For any $y \in S$, observation 1 says that the probability that the operation spends time on y is $O(1/r)$. Thus, the expected (or "average") time consumption attributed to y is $O(1/r)$. To get the total expected time, we must sum up the expected time usage for all elements in S. This is $|S| \cdot O(1/r)$, which is $O(1)$ since we chose r such that $r \geq n \geq |S|$. In conclusion, the expected time for any operation is constant.

3 Cuckoo Hashing

Suppose that we want to be able to look up items in constant time in the *worst case*, rather than just in expectation. What could be done to achieve this? One approach is to maintain a hash function that has no collisions for elements in the set. This is called a *perfect hash function*, and allows us to insert the items directly into the array, without having to use a linked list. Though this approach can be made to work, it is quite complicated, in particular when dealing with insertions of items. Here we will consider a simpler way, first described in *(R. Pagh and F. Rodler, Cuckoo Hashing, Proceedings of European Symposium on Algorithms, 2001).* In these notes the algorithm is slightly modified to enable a simplified analysis.

Instead of requiring that x should be stored at position $h_1(x)$, we give two alternatives: Position $h_1(x)$ and position $h_2(x)$. We allow at most one element to be stored at any position, i.e., there is no need for a data structure holding colliding items. This will allow us to look up an item by looking at just two positions in the array. When inserting a new element x it may of course still happen that there is no space, since both position $h_1(x)$ and position $h_2(x)$ can be occupied. This is resolved by imitating the nesting habits of the European cuckoo: Throw out the current occupant y of position $h_1(x)$ to make room! This of course leaves the problem of placing y in the array. If the alternative position for y is vacant, this is no problem. Otherwise y, being a victim of the ruthlessness of x, repeats the behavior of x and throws out the occupant. This is continued until the procedure finds a vacant position or has taken too long. In the latter case, new hash functions are chosen, and the whole data structure is rebuilt ("rehashed"). The pseudocode for the insertion procedure is shown in Figure 2. The variable "pos" keeps track of the position in which we are currently trying to insert an item. The notation $a \leftrightarrow b$ means that the contents of a and b are swapped. Note that the procedure does not start out with examining whether any of positions $h_1(x)$ and $h_2(x)$ are vacant, but simply inserts x in position $h_1(x)$.



procedure insert(x)
  if $T[h_1(x)] = x$ **or** $T[h_2(x)] = x$ **then return**;
  pos $\leftarrow h_1(x)$;
  loop n **times** {
    if $T[\text{pos}] =$ `NULL` **then** { $T[\text{pos}] \leftarrow x$; **return**};
    $x \leftrightarrow T[\text{pos}]$;
    if pos$= h_1(x)$ **then** pos$\leftarrow h_2(x)$ **else** pos$\leftarrow h_1(x)$;}
  rehash(); insert(x)
end

Figure 2. Cuckoo hashing insertion procedure and illustration. The arrows show the alternative position of each item in the dictionary. A new item would be inserted in the position of A by moving A to its alternative position, currently occupied by B, and moving B to its alternative position which is currently vacant. Insertion of a new item in the position of H would not succeed: Since H is part of a cycle (together with W), the new item would get kicked out again.

3.1 Examples and discussion

Figure 2 shows an example of the *cuckoo graph*, which is a graph that has an edge for each item in the dictionary, connecting the two alternative positions of the item. In the figure we have put an arrow on each edge to indicate which element can move along the edge to its alternative position. We can note that the picture has some similarities with Hashing with Chaining. For example, if we insert an item in the position of A, the insertion procedure traverses the path, or chain, involving the positions of A and B. Indeed, it is a simple exercise to see that when inserting an item x, the insertion procedure will only visit positions to which there is a path in the cuckoo graph from either position $h_1(x)$ or $h_2(x)$. Let us call this the *bucket* of x. The bucket may have a more complicated structure than in Hashing with Chaining, since the cuckoo graph can have *cycles*. The graph in Figure 2, for example, has a cycle involving items H and W. If we insert Z where $h_1(\text{Z})$ is the position of W, and $h_2(\text{Z})$ the position of A, the traversal of the bucket will move items W, H, Z, A, and B, in this order. In some cases there is no room for the new element in the bucket, e.g., if the possible positions are those occupied by H and W in Figure 2. In this case the insertion procedure will loop n times before it gives up and the data structure is rebuilt ("rehashed") with new, hopefully better, hash functions.

Observe that the insertion procedure can only loop n times if there is a cycle in the cuckoo graph. In particular, every insertion will succeed as long as there is no cycle. Also, the time spent will be bounded by a constant times the size of the bucket, i.e., observation 2 from the analysis of Chained Hashing holds.

3.2 Analysis

By the above discussion, we can show that the expected insertion time (in absence of a rehash) is constant by arguing that observation 1 holds as well. In the following, we consider the *undirected* cuckoo graph, where there is no orientation on the edges. Observe that y can be in the bucket of x only if there is a path between one of the possible positions of x and the position of y in the undirected cuckoo graph. This motivates the following lemma, which bounds the probability of such paths:

Lemma 1 *For any positions i and j, and any $c > 1$, if $r \geq 2cn$ then the probability that in the undirected cuckoo graph there exists a path from i to j of length $\ell \geq 1$, which is a shortest path from i to j, is at most $c^{-\ell}/r$.*

Interpretation. Before showing the lemma, let us give an interpretation in words of its meaning. The constant c can be any number greater than 1, e.g. you may think of the case $c = 2$. The lemma says that if the number r of nodes is sufficiently large compared to the number n of edges, there is a low probability that any two nodes i and j are connected by a path. The probability that they are connected by a path of constant length is $O(1/r)$, and the probability that a path of length ℓ exists (but no shorter path) is exponentially decreasing in ℓ.

Proof: We proceed by induction on ℓ. For the base case $\ell = 1$ observe that there is a set S of at most n items that may have i and j as their possible positions. For each item, the probability that this is true is at most $2/r^2$, since at most 2 out of r^2 possible choices of positions give the alternatives i and j. Thus, the probability is bounded by $\sum_{x \in S} 2/r^2 \leq 2n/r^2 = c^{-1}/r$.

In the inductive step we must bound the probability that there exists a path of length $\ell > 1$, but no path of length less than ℓ. This is the case only if, for some position k:

- There is a shortest path of length $\ell - 1$ from i to k that does not go through j, and
- There is an edge from k to j.

By the induction hypothesis, the probability that the first condition is true is bounded by $c^{\ell-1}/r$. Given that the first condition is true, what is the probability that the second one holds as well, i.e., that there exists an item that has k and j as its possible positions? As before, there is a set S of less than n items that could have j as one of their alternative positions, and for each one of them the probability is at most $2/r^2$. Thus, the probability is bounded by c^{-1}/r, and the probability that both conditions hold for a particular choice of k is less than c^ℓ/r^2. Summing over the r possibilities for k, we get that the probability of a path of length ℓ is at most $c^{-\ell}/r$. $\square$

To show that observation 1 holds, we observe that if x and y are in the same bucket, then there is a path of some length ℓ between $\{h_1(x), h_2(x)\}$ and $\{h_1(y), h_2(y)\}$. By the above lemma, this happens with probability at most $4\sum_{\ell=1}^{\infty} c^{-\ell}/r = \frac{4}{c-1}/r = O(1/r)$, as desired.

Rehashing. We will end with a discussion of the cost of rehashing. We consider a sequence of operations involving ϵn insertions, where ϵ is a small constant, e.g. $\epsilon = 0.1$. Let S' denote the set of items that exist in the dictionary at some time during these insertions. Clearly, a cycle can exist at some point during these insertions only if the undirected cuckoo graph corresponding to the items in S' contain a cycle. A cycle, of course, is a path from a position to itself, so Lemma 1 says that any particular position is involved in a cycle with probability at most $\sum_{\ell=1}^{\infty} c^{-\ell}/r$, if we take $r \geq 2c(1+\epsilon)n$. Thus, the probability that there is a cycle can be bounded by summing over all r positions: $r \cdot \sum_{\ell=1}^{\infty} c^{-\ell}/r = \frac{1}{c-1}$. For $c = 3$, for example, the probability is at most $1/2$ that a cycle occurs during the ϵn insertions. The probability that there will be two rehashes, and thus two independent cycles is $1/4$, and so on. In conclusion, the expected number of rehashes is at most $\sum_{i=1}^{\infty} 2^{-i} = 1$. If the time for one rehashing is $O(n)$, then the expected time for all rehashings is $O(n)$, which is $O(1/\epsilon)$ per insertion. This means that the expected *amortized* cost of rehashing is constant. That a single rehashing takes expected $O(n)$ time follows from the fact that the probability of a cycle in a given attempt is at most $1/2$.

3.3 Final remarks

The analysis of rehashing above is not exact. In fact, the expected amortized cost of rehashing is only $O(1/n)$ per insertion, for any $c > 1$.

To get an algorithm that adapts to the size of the set stored, a technique called *global rebuilding* can be used. Whenever the size of the set becomes too small compared to the size of the hash table, a new, smaller hash table is created. Conversely, if the hash table fills up to its capacity, a new, larger hash table can be created. To make this work efficiently, the size of the hash table should always be increased or decreased with a constant factor (larger than 1), e.g. doubled or halved. Then the expected amortized cost of rebuilding is constant.