Lecture Notes on
Hash Tables

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 13
February 24, 2011

1 Introduction

In this lecture we introduce so-called *associative arrays*, that is, data structures that are similar to arrays but are not indexed by integers, but other forms of data such as strings. One popular data structures for the implementation of associative arrays are *hash tables*. To analyze the asymptotic efficiency of hash tables we have to explore a new point of view, that of *average case complexity*. Another computational thinking concept that we see for the first time is *randomness*. In order for hash tables to work efficiently in practice we need hash functions whose behavior is predictable (deterministic) but has some aspects of randomness.

2 Associative Arrays

Arrays can be seen as a mapping, associating with every integer in a given interval some data item. It is finitary, because its domain, and therefore also its range, is finite. There are many situations when we want to index elements differently than just by integers. Common examples are strings (for dictionaries, phone books, menus, data base records), or structs (for dates, or names together with other identifying information). They are so common that they are primitive in some languages such as PHP, Python, or Perl and perhaps account for some of the popularity of these languages. In many applications, associative arrays are implemented as hash tables because of their performance characteristics. We will develop them incrementally to understand the motivation underlying their design.

3 Chains

In many applications requiring associative arrays, we are storing complex data values and want to access them by a *key* which is derived from the data. A typical example of keys are strings, which are appropriate for many scenarios. For example, the key might be a student id and the data entry might be a collection of grades, perhaps another associative array where the key is the name of assignment or exam and the data is a score. We make the assumption that keys are unique in the sense that in an associative array there is at most one data item associated with a given key. In some applications we may need to complicate the structure of keys to achieve this uniqueness. This is consistent with ordinary arrays, which have a unique value for every valid index.

A first idea to explore is to implement the associative array as a linked list, called a *chain*. If we have a key k and look for it in the chain, we just traverse it, compute the intrisic key for each data entry, and compare it with k. If they are equal, we have found our entry, if not we continue the search. If we reach the end of the chain and do not find an entry with key k, then no entry with the given key exists. If we keep the chain unsorted this gives us $O(n)$ worst case complexity for finding a key in a chain of length n, assuming that computing and comparing keys is constant time.

Given what we have seen so far in our search data structures, this seems very poor behavior, but if we know our data collections will always be small, it may in fact be reasonable on occasion.

Can we do better? One idea goes back to binary search. If keys are ordered we may be able to arrange the elements in an array or in the form of a tree and then cut the search space roughly in half every time we make a comparison. This approch will occupy us for a few lectures after the first midterm. Designing such data structures is a rich and interesting subject. The best we can hope for with this approach is $O(\log(n))$, where n is the number of entries. We have seen that this function grows very slowly, so this is quite a practical approach.

Nevertheless, the challenge arises if we can do better than $O(\log(n))$, say, constant time $O(1)$ to find an entry with a given key. We know that it can done be for arrays, indexed by integers, which allow constant-time access. Can we also do it, for example, for strings?

4 Hashing

The first idea behind hash tables is to exploit the efficiency of arrays. So: to map a key to an entry, we first map a key to an integer and then use the integer to index an array A. The first map is called a *hash function*. We write it as $\mathrm{hash}(\_)$. Given a key k, our access could then simply be $A[\mathrm{hash}(k)]$.

There is an immediate problem with this approach: there are only 2^{31} positive integers, so we would need a huge array, negating any possible performance advantages. But even if we were willing to allocate such a huge array, there are many more strings than `int`'s so there cannot be any hash function that always gives us different `int`'s for different strings.

The solution is to allocate an array of smaller size, say m, and then look up the result of the hash function modulo m, for example, $A[\mathrm{hash}(k)\%m]$. This creates a new problem: it is inevitable that multiple strings will map to the same array index. For example, if the array has size m then if we have more then m elements, at least two must map to the same index. In practice, this will happen much sooner that this.

If two hash functions map a key to the same integer value (modulo m), we say we have a *collision*. In general, we would like to avoid collisions, because some additional operations will be required to deal with them, slowing down operations and taking more space. We analyze the cost of collisions more below.

5 Separate Chaining

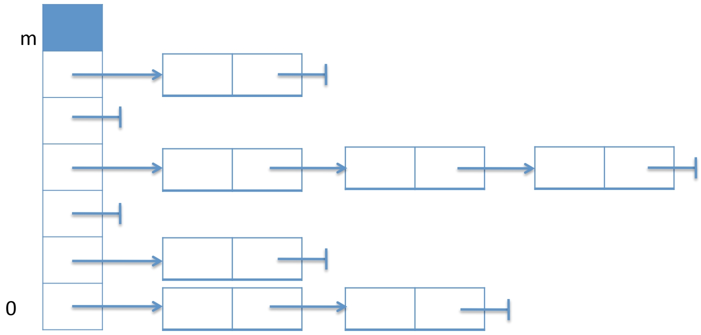
How do we deal with collisions of hash values? The simplest is a technique called *separate chaining*. Assume we have $\mathrm{hash}(k_1)\%m = i = \mathrm{hash}(k_2)\%m$, where k_1 and k_2 are the distinct keys for two data entries e_1 and e_2 we want to store in the table. In this case we just arrange e_1 and e_2 into a chain (implemented as a linked list) and store this list in $A[i]$.

In general, each element $A[i]$ in the array will either be `NULL` or a chain of entries. All of these must have the same hash value for their key (modulo m), namely i. As an exercise, you might consider other data structures here instead of chains and weigh their merits: how about sorted lists? Or queues? Or doubly-linked lists? Or another hash table?

We stick with chains because they are simple and fast, provided the chains don't become too long. This technique is called *separate* chaining because the chains are stored seperately, not directly in the array. Another technique, which we do not discuss, is *linear probing* where we continue by searching (linearly) for an unused spot array itself, starting from the place where the has function put us.

6 Average Case Analysis

Under separate chaining, a snapshot of a hash table might look something like this picture.



How long do we expect the chains to be on average? For a total number n of entries in a table of size m, it is n/m. This important number is also called the *load factor* of the hash table. How long does it take to search for an entry with key k? We follow these steps:

1. Compute $i = \mathrm{hash}(k)\%m$. This will be $O(1)$ (constant time), assuming it takes constant time to compute the hash function.
2. Go to $A[i]$, which again is constant time $O(1)$.
3. Search the chain starting at $A[i]$ for an element whose key matches k. We will analyze this next.

The complexity of the last step depends on the length of the chain. In the *worst case* it could be $O(n)$, because all n elements could be stored in one chain. This worst case could arise if we allocated a very small array (say, $m = 1$), or because the hash function maps all input strings to the same table index i, or just out of sheer bad luck.

Ideally, all the chains would be approximately the same length, namely n/m. Then for a fixed load factor such as $n/m = \alpha = 2$ we would take on the average 2 steps to go down the chain and find k. In general, as long as we don't let the load factor become too large, the *average* time should be $O(1)$.

If the load factor does become too large, we could dynamically adapt its size, like in an unbounded array. As for unbounded arrays, it is beneficial to double the size of the hash table when the load factor becomes too high, or possibly halve it if the size becomes too small. Analyzing these factors is a task for amortized analysis, just as for unbounded arrays.

7 Randomness

The average case analysis relies on the fact that the hash values of the key are relatively evenly distributed. This can be restated as saying that the probability that each key maps to an array index i should be about the same, namely $1/m$. In order to avoid systematically creating collisions, small changes in the input string should result in unpredicable change in the output hash value. We can achieve this by exploiting the idea behind the pseudorandom number generator that we have already used in order to create random tests.

The code for the a hash function based on the a random number generator using linear congruential method can be in the file hash-string.c0.

Handwritten hash functions often do not work well, which can significantly affect the performance of the hash table. Whenever possible, techniques like the above should be used to avoid any systematic bias.

8 Implementation

In the next lecture we will consider the implementation. A particularly critical aspect of that is the interface to hash tables, which is more complicated than the interface to data structures we have considered so far.