Lecture 12 Notes
Hash Tables

15-122: Principles of Imperative Computation (Spring 2016)
Frank Pfenning, Rob Simmons

1 Introduction

In this lecture we re-introduce the *dictionaries* that were implemented as a part of Clac and generalize them as so-called *associative arrays*. Associative arrays are 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 revisit 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.

Relating to our learning goals, we have

Computational Thinking: We consider the importance of *randomness* in algorithms, and also discuss *average case analysis*, which is how we can argue that hash tables have acceptable performance.

Algorithms and Data Structures: We describe a *linear congruential generator*, which is a certain kind of *pseudorandom number generator*. We also discuss hashtables and their implementation with *separate chaining* (an array of linked lists).

Programming: We review the implementation of the `rand` library in C0.

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 Keys and values

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.

We can think of built-in C0 arrays as having a set number of keys: a C0 array of length 3 has three keys `0`, `1`, and `2`. Our implementation of unbounded arrays allowed us to add a specific new key, `3`, to an array; we want to be able to add new keys to the associative array. We want our associative arrays to allow us to have more interesting keys (like strings, or non-sequential integers) while keeping the property that there is a unique value for each valid key.

4 Chains

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 intrinsic 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. Designing such data structures is a rich and interesting subject, but 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?

5 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 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 idea has an obvious 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 — this simple observation is an instance of what is known as the *pigeonhole principle*. In practice, this will happen much sooner than this.

If a hash function maps two keys 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.

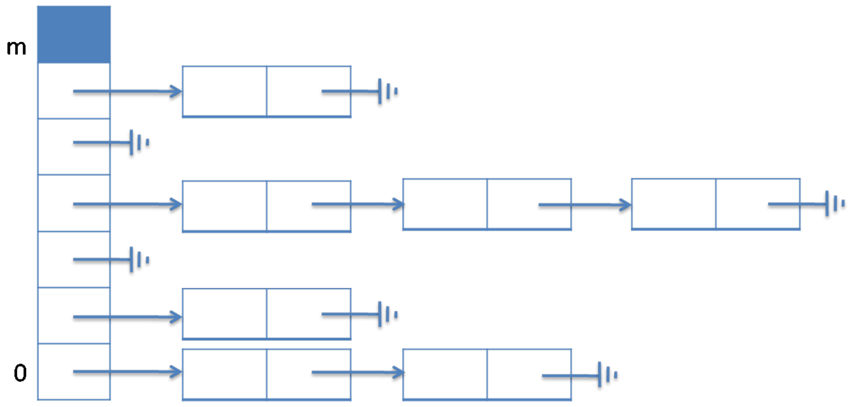
6 Separate Chaining

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 separately, not directly in the array. Another technique, which we will not discuss at length, is *linear probing* where we continue by searching (linearly) for an unused spot in the array itself, starting from the place where the hash function put us.

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



7 Average Case Analysis

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 $i = \text{hash}(k) \% m$. 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 the size of the array, 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.

8 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 unpredictable change in the output hash value that is uniformly distributed over the range of C0 integers. We can achieve this with a *pseudorandom number generator* (PRNG). A pseudorandom number generator is just a function that takes one number and obtains another in a way that is both unpredictable and easy to calculate. The C0 `rand` library is a pseudorandom number generator with a fairly simple interface:

```
/* library file rand.h0 */
typedef struct rand* rand_t;
```

```
rand_t init_rand (int seed);
int rand(rand_t gen);
```

One can generate a random number generator (type `rand_t`) by initializing it with an arbitrary seed. Then we can generate a sequence of random numbers by repeatedly calling `rand` on such a generator.

The `rand` library in C0 is implemented as a *linear congruential generator*. A linear congruential generator takes a number x and finds the next number by calculating $(a \times x) + c$ modulo m, a number that is used as the next x. In C0, it's easiest to say that m is just 2^{32}, since addition and multiplication in C0 are already defined modulo 2^{32}. The trick is finding a good multiplier a and summand c.

If we were using 4-bit numbers (from -8 to 7 where multiplication and addition are modulo 16) then we could set a to 5 and c to 7 and our pseudorandom number generator would generate the following series of numbers:

$$0 \to 7 \to (-6) \to (-7) \to 4 \to (-5) \to (-2) \to \\ -3 \to (-8) \to (-1) \to 1 \to (-4) \to 3 \to 6 \to 5 \to 0 \to \ldots$$

The PRNG used in C0's library sets a to 1664525 and c to 1013904223 and generates the following series of numbers starting from 0:

$$0 \to 1013904223 \to 1196435762 \to (-775096599) \to (-1426500812) \to \ldots$$

This kind of generator is fine for random testing or (indeed) the basis for a hashing function, but the results are too predictable to use it for cryptographic purposes such as encrypting a message. In particular, a linear congruential generator will sometimes have repeating patterns in the lower bits. If one wants numbers from a small range it is better to use the higher bits of the generated results rather than just applying the modulus operation.

It is important to realize that these numbers just *look* random, they aren't really random. In particular, we can reproduce the exact same sequence if we give it the exact same seed. This property is important for both testing purposes and for hashing. If we discover a bug during testing with pseudorandom numbers, we want to be able to reliably reproduce it, and whenever we hash the same key using pseudorandom numbers, we need to be sure we will get the same result.

```
/* library file rand.c0 */
struct rand {
  int seed;
};

rand_t init_rand (int seed) {
  rand_t gen = alloc(struct rand);
  gen->seed = seed;
  return gen;
}

int rand(rand_t gen) {
  gen->seed = gen->seed * 1664525 + 1013904223;
  return gen->seed;
}
```

Observe that some choices of the numbers a and c would be terrible. For example, if we were to work with 4-bit numbers, so that $m = 16$, choosing $a = 0$ would mean that our "pseudo-random" generator always returns c. Were we to choose $a = c = 4$, it would only return values among -8, -4, 0, and 4. In general, we want, at a minimum, that the factor c and the modulus m be *relatively prime*, i.e., that their greatest common divisor be 1.

Exercises

Exercise 1. *What happens when you replace the data structure for separate chaining by something other than a linked list? Discuss the changes and identify benefits and disadvantages when using a sorted list, a queue, a doubly-linked list, or another hash table for separate chaining.*

Exercise 2. *Consider the situation of writing a hash function for strings of length two, that only use the characters* `'A'` *to* `'Z'`*. There are $26 \times 26 = 676$ different such strings. You were hoping to get away with implementing a hash table without collisions, since you are only using 79 out of those 676 two-letter words. But you still see collisions most of the time. Look up the* birthday paradox *and use it to explain this phenomenon.*