Hash Tables

In this lecture we introduce the *Dictionary* ADT and give a simple data structure for it that is efficient in practice.

4.1 Dictionaries

A *Dictionary* stores key–element pairs, which are called *items*. In this and future *Dictionary*–related notes, n will denote the number of items in a dictionary. *Dictionary* allows key duplicates—several items may have the same key. A dictionary provides three basic methods:

- findElement(k): If the dictionary contains an item with key k, then return the element of such an item, otherwise return the special element NO_SUCH_KEY.
- insertItem(k, e): Insert an item with key k and element e.
- removeItem(k): If the dictionary contains an item with key k, then delete such an item and return its element, otherwise return the special element NO_SUCH_KEY.

Of course we must ensure that NO_SUCH_KEY cannot itself be an element that can be associated with any key. An alternative approach is to turn the two relevant methods into boolean functions so that success or otherwise is indicated by the value returned; the element, if any, could be returned via a parameter.

A key point to observe is that *Dictionary* allows the insertion of more than one element with the same key. In this scenario, removeItem(k) removes just one item with key k and not all of them (when there is more than one such item).

An obvious data structure for *Dictionary* can be based on the linked list of Lecture Note 3. If we implement such a *list dictionary* by inserting at the end of the list (not keeping the items in key-sorted order), the method insertItem(k, e) has running time $\Theta(1)$ and the methods findElement(k) and removeItem(k) have running time $\Theta(n)$. In this note and future notes (Lecture Notes 5–7), we will see data structures which achieve better worst-case running times for the *Dictionary* methods. In this note we present Hash Tables.

4.2 Direct Addressing and Bucket Arrays

Suppose we are in the special case where the keys of the items to be inserted into the *Dictionary* are integers in the range $0, \ldots, N-1$, for some $N \in \mathbb{N}$ with $N > 0$, and suppose also that no two elements have the same key (this is a restricted case of *Dictionary*). In this case we can implement a *Dictionary* by setting up an array B of length N of the appropriate type to hold elements. The method insertItem(k, e) simply sets $B[k] \leftarrow e$. The method findElement(k) returns $B[k]$, and the method removeItem(k) returns $B[k]$ and resets $B[k]$ to $null$. The running time

of these methods is $\Theta(1)$. This data structure implementing the *Dictionary* ADT is sometimes called a *direct address table*.

The problem with direct addressing is that we have to keep an array of size N in memory, even if n, the number of items stored in the dictionary, is much smaller than N. If this is the case, we waste a great deal of memory space. Direct addressing is only acceptable if the number of items we expect to store in the dictionary is somewhat close to N. If we are in this fortunate situation, however, this simple data structure is the most efficient implementation of the *Dictionary* ADT.

Another problem with the simple direct addressing implementation is that it assumes no two items have the same key. However we can remove this restriction by changing the structure of our array of length N. We say that when we try to insert an item into a cell of the direct address table which already has an item stored, a *collision* occurs. We can handle collisions by changing the model so that we work on an array B of *Lists* of elements, and store all elements with key k in the list $B[k]$. In this context, the lists $B[i]$, for $0 \leq i \leq N-1$, are often called *buckets*, and the array B is called the *bucket array*. To insert, find, and remove elements, we use the methods insertFirst(), first(), and remove(p) (with $p = 1$) of *List*. As we saw in Lecture Note 3, the methods insertFirst(), first() and remove(p) all have a running-time of $\Theta(1)$.

Bucket arrays are very efficient if N is not much larger than the number of different keys appearing in the dictionary (for example, if we have at least cN elements in the dictionary, for $c > 0$ some constant which is not too small). If this is the case, not much space is wasted. Note that because findElement(k) and removeItem(k) are defined in terms of a key k, and we only require one element with that key to be found/removed, the first element of the list for that key may *always* be taken. Therefore in this simple case, we need not even concern ourselves with the size of the list (or bucket) for each k—the running-time is $\Theta(1)$ for all methods, regardless of how the elements are distributed.

4.3 Hash Tables

Usually, we are not in a situation where keys are small natural numbers. Hash tables extend the idea of bucket arrays to arbitrary keys. Again, we set up a bucket array B of length N, for a suitable integer N. The keys are *mapped* to natural numbers in the range $0, \ldots, N-1$ using a *hash function* h that maps each key k to a number $h(k) \in \{0, \ldots, N-1\}$. When an item with key k arrives to be inserted, it is inserted into the bucket $B[h(k)]$.

Suppose for a moment that we have chosen a hash function h. It is certainly possible that two elements with two different keys k_1 and k_2 end up in the same bucket $B[h(k_1)]$ if $h(k_1) = h(k_2)$. Thus we cannot just store the elements in the buckets, but must store the full items, including their keys. In effect, we let each bucket be a small dictionary. By choosing a good hash function (and by making reasonable assumptions about the distribution on the key k), we *hope* to avoid too many keys having the same hash value. This is important in this new scenario of hash values, even though it did not matter when we defined $h(k) = k$ above. The reason that it becomes important *now* is because when

we implement findElement(k) or removeItem(k), we actually need to search the bucket at $h(k)$ rather than just taking any element of that bucket (not all elements in the bucket have key k). If we have a good hash function (and if our input keys are distributed well) the buckets will usually be small, and we can use simple list dictionaries for the buckets. The technique of using lists to store the elements with hash value h is sometimes called *chaining* in the literature.

A *hash table* realising the *Dictionary* ADT consists of a hash function h and an array B of length N of list dictionaries. Algorithms 4.1–4.3 implement the methods of *Dictionary*.

Algorithm findElement(k)

1. Compute $h(k)$
2. **return** $B[h(k)]$.findElement(k)

Algorithm 4.1

Algorithm insertItem(k, e)

1. Compute $h(k)$
2. $B[h(k)]$.insertItem(k, e)

Algorithm 4.2

Algorithm removeItem(k)

1. Compute $h(k)$
2. **return** $B[h(k)]$.removeItem(k)

Algorithm 4.3

The running time of the methods obviously depends on the time needed to compute $h(k)$ and on the running time of the corresponding methods of the list dictionaries. Let us assume that computing $h(k)$ requires at most T_h computation steps. (Of course T_h may depend on the key, so it would be more precise to write $T_h(n_{\text{key}})$, where n_{key} denotes the size of the key. On the other hand, keys often have constant size.)

Recall that the method insertItem of list dictionary has running time $\Theta(1)$ and the methods findElement and removeItem have a worst-case running time of $\Theta(m)$, when m is the number of items in the dictionary. Then the worst-case running time of insertItem of the hash table dictionary is $T_h + \Theta(1)$. The worst-case running time of both findElement and removeItem is $T_h + \Theta(m)$, where m denotes the size of the bucket $B[h(k)]$ associated with the key k. Since it could happen that all items end up in the same bucket, in the worst case we have

$m = n$. Thus in the worst case, a hash table dictionary is no more efficient then a list dictionary. However, by choosing an appropriate hash function h, *and* by assuming that the input keys are *uniformly distributed*, we can avoid ending up in the worst case in all practically relevant situations. As a matter of fact, we can usually *expect* the size m of all buckets to be $O(1)$, giving us an *expected* running time of $T_h + \Theta(1)$ for all methods.

Remark 4.4. Before discussing the choice of good hash functions, note two facts relating to the basic implementation of hash tables above. First note that instead of using list dictionaries as buckets, we could use any other implementation of *Dictionary*. List dictionaries are the most efficient for small dictionaries, however, and we expect our buckets to be small. Second, note that there are versions of hash tables that do not use external buckets, but store all items directly in the array. While this is a bit more space efficient, it requires sophisticated schemes for handling collisions (usually referred to as *open addressing schemes*). Details can be found in most algorithms textbooks (e.g., Chapter 11 of [CLRS], Section 2.5 of [GT], Chapter 14.3 of Sedgewick).

Another reference is the classic three volume series *The Art of Computer Programming* by D.E. Knuth (Addison Wesley) which contains extensive discussions of many of the algorithms and data structures we will see in this thread. They are seriously challenging books. However, if you really want to cover a topic in great detail, this is often the place to look. Section 6.4 of the third volume (entitled *Sorting and Searching*) is on hashing.

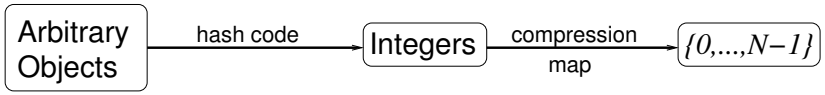
4.4 Hash Functions

We want to find a hash function h that maps keys to natural numbers in a fixed range $0, \ldots, N-1$ in such a way that

(H1) h evenly distributes the keys over the whole range. That is, the total number of keys mapped to the number j is roughly the same for all $j \in 0, \ldots, N$.

(H2) h is easy to compute.

A common strategy is to first map arbitrary objects to integers by a function called the *hash code*, and then map arbitrary integers to integers in the range $0, \ldots, N-1$ by a function called the *compression map*:



Hash Codes

Since every object is represented as a sequence of bits in memory anyway, in principle there is no problem: we can just consider the sequence of bits representing a key as an integer and let this integer be the hash code of the key.

In practice, though, there might be a problem, because integers on a real computer cannot be arbitrarily large. Suppose we are working with a machine

where integers can be at most 32 bits long. The easiest way to assign hash codes in this situation is to only take the leftmost (or rightmost) 32 bits of the bit representation of the key to be the hash code. In general, this may be a dangerous strategy, because all keys whose bit representation starts with the same 32 bits will get the same hash code and will end up in the same bucket. For example, if keys are strings, then all strings starting with the same few letters will get the same hash code. Since the distribution of strings arising in practical applications is hardly ever random, the resulting hash map will not usually satisfy condition (H1) in practical applications.

Summation hash code

If we want to take all bits of the representation of the key into account, we could proceed as follows: instead of considering the bit representation of the key as one integer, we consider it as a sequence of integers, $a_0, \ldots, a_{\ell-1}$. One way of mapping this sequence to a single integer is to sum them all up. Thus if we have a key k whose bit representation corresponds to integers $a_0, \ldots, a_{\ell-1}$, we could let the hash code of k be

$$a_0 + a_1 + \cdots + a_{\ell-1}.$$

We simply disregard overflows in the summation, that is, we actually let the hash code be $\sum_{i=0}^{\ell-1} a_i \bmod 2^{32}$.

While this summation hash code is a quite reasonable choice, there are still some problems with condition (H1). For example, we can recognise some "patterns" in the way sequences get mapped to buckets—keys corresponding to sequences (a_0, a_1, a_2), (a_1, a_2, a_0), and (a_1, a_0, a_2) will get the same hash code, which may be problematic in some applications. More generally any permutation of a sequence is given the same code.

Polynomial hash code

For the *polynomial hash code*, we choose a fixed integer x and let the hash code of a key k corresponding to the sequence $a_0, \ldots, a_{\ell-1}$ be

$$a_0 + a_1 x + a_2 x^2 + \cdots a_{\ell-1} x^{\ell-1}$$

(again disregarding overflows). This polynomial hash code can be especially well-suited to keys of type `String`. Suppose our `String` $s = s_0 \ldots s_{\ell-1}$ is a sequence of 7-bit characters (so each $s_i \in \{0, 1 \ldots, 127\}$), and suppose we want to map to a hash value in the set $\{0, 1, \ldots, N-1\}$. Then we can perform the mapping by choosing an appropriate integer x and evaluating $(\sum_{i=0}^{\ell-1} s_i x^i) \bmod N$. *It is very important, in choosing the value of x, to make sure that x and N are coprime* (i.e., they do not share common factors, in other words the only natural number that divides both of them is 1). If x and N are not coprime, then small strings will not get well distributed among the buckets of the Hash Table. As an example, Java's `HashMap` uses a polynomial hash function with $x = 31$, which is prime, to hash keys of type `String` (for `HashMap`, usually $N = 2^k$ for some integer k). The observations here are based on practical experience rather than any theoretical analysis.

Polynomial hash codes seem to satisfy (H1) (at least on an heuristic level, we have given no proof), it it not clear that they satisfy (H2). However, polynomials can be evaluated quite efficiently, as the following remarks show.

Aside: Evaluating Polynomials

Suppose we are given integers $a_0, \ldots, a_{\ell-1}, x$ and want to compute

$$a_0 + a_1 x + a_2 x^2 + \cdots a_{\ell-1} x^{\ell-1}. \tag{4.1}$$

A naïve way of doing this requires $\Theta(\ell^2)$ arithmetic operations (additions and multiplications). However, there is a clever way of rewriting the polynomial in such a way that it can be evaluated with only $\Theta(\ell)$ arithmetic operations. *Horner's rule* says that the polynomial (4.1) is equal to

$$a_0 + x(a_1 + x(a_2 + \cdots + x(a_{\ell-2} + x \cdot a_{\ell-1}) \cdots)).$$

This expression clearly can be evaluated with only $\Theta(\ell)$ arithmetic operations.

It is interesting to note that Horner's rule has been proved to be optimal. Any algorithm that can evaluate an arbitrary polynomial of degree l for arbitrary x must use at least as many additions/subtractions and at least as many multiplications as Horner's rule (we allow divisions but they do not help). This is a rare example where we know the exact *complexity* of a non-trivial computational problem.

Compression Maps

Many Hash Functions only consist of a "hash code", and do not involve a compression phase. However, depending on the domain, it may be helpful to first perform a hash to a large key space, and then "compress" these large integers to integers in the range $0, \ldots, N-1$. Like the hash code, the compression map should satisfy conditions (H1) and (H2). It turns out that compression functions of the following form do this fairly well: an integer k is mapped to

$$|ak + b| \bmod N,$$

where a, b are just randomly chosen integers. We choose a, b at running time whenever we create a new hash table. This method works particularly well if a and N are coprime.

If a and N are not coprime then the map is definitely bad. Let's assume that a, b and k are non-negative to simplify the discussion. If we are to have a good hash map then, at the very least, for each location r of the array there should be a key that hashes into it, i.e., the equation

$$ak + b \equiv r \bmod N$$

should have a solution (for an integer k, if k is negative we can get a positive value by adding enough multiples of N to it). Note that we may assume $0 \leq b \leq N-1$ because we take the remainder after division by N so any quotient would be lost

anyway. Hence 0 hashes to b since $a \cdot 0 + b = b$ and this stays as b after taking the remainder when divided by N. The preceding equation is the same as requiring that the equation

$$ak + b = qN + r$$

has a solution for integers q, k. Rearranging we have

$$ak - qN = r - b.$$

So if $d = \gcd(a, b)$ divides both a and N then it divides $r - b$, i.e., $r - b = sd$ for some integer s. Hence the only possible values of r are $b + sd$ for integers s. We know that 0 hashes to b but then the next location to which any key might hash is $b + d$. So if $d > 1$ then nothing hashes to $b + 1, b + 2, \ldots, b + d - 1$. In general, our argument shows that the only locations to which anything can hash are all those of the form $b + sd$ where $0 \leq b + sd \leq N - 1$ (remember that s can be negative as well as positive or 0). In fact we can prove (using a simple fact deduced from the Euclidean algorithm for finding greatest common divisors[1]) that all such locations do have a key hashing to them. So, unless $\gcd(a, N) = 1$ the compression map is definitely bad. If $\gcd(a, N) = 1$ then the map still might be bad because the keys do not spread out evenly but this is a property of the distribution of keys which varies with the application domain.

4.5 Load Factors and Re-hashing

The choice of a good size for N obviously depends on the number n of items we intend to store in our hash table. We call the fraction n/N the *load factor* of the hash table. We do not want the load factor to be too high, because that would lead to many collisions. We don't want it to be too low, because that would be a waste of memory space. A common load factor in practice (e.g., with the JVC `HashMap`) is $3/4$.

If we know n, we can choose N to be a prime number of size roughly $(4/3)n$. As N is prime it is automatically coprime to all $a < 2N$ provided $N \neq 2$, which is the case in any sensible application of hashing. If we do not know n in advance, or if n changes over time because items are inserted and removed, we must occasionally choose a new N, create a new hash table, and move all items from the old table to the new one. This is called *re-hashing*. We adopt a similar strategy as the one we used for dynamic arrays in Lecture Note 3. Whenever the load factor gets too far above $3/4$ we choose a new prime number N close to $(4/3)n$ and create a new bucket array of size N. We choose a new compression function and thus obtain a new hash function that maps our keys to the new range $0, \ldots, N - 1$. Then we move all items stored in the old table to the new table. We proceed similarly if the load factor falls too far below $3/4$. To do this we need to be able to find prime numbers of a given size. There are efficient ways of doing this, but beyond the scope of this course. A common trick in practice is to store a table containing for $7 \leq k \leq 31$, the closest prime number to 2^k, which would easily cover all practical applications (though with 2 rather than $4/3$).

[1] This states that the equation $au + bv = c$ where a, b, c are integers has a solution for integers u, v if and only if $\gcd(a, b)$ divides c.

4.6 Hashing in Java

The *Dictionary* ADT is similar (though not the same) to the `Map` ADT of the Java Collections framework. The difference is that `Map` of JVC requires that the keys should all be distinct (does not allow different elements with the same key). One of the implementations of *Map* that is provided in the JVC is the `HashMap` implementation, which is a Hash Table where the number of buckets N may vary (but the default is 16). `HashMap` has parameters which allow the user to specify the (initial) table size N, and the desired "load factor" (the average number of items per bucket). This Java implementation "re-sizes" the table as the number of items n increases (this aspect of the implementation is similar to that of the *dynamic array* that we met in Lecture Note 3). Different hash functions are used, depending on the domains of the `key` which is being hashed. Java also offers a `Hashtable` implementation which is almost identical to `Hashmap`.

4.7 Further Reading

In [GT], the "Maps and Dictionaries" chapter (chapter 8 in edition 3, Chapter 9 in edition 4) has a lot of material on this topic. Sections 8.1, 8.2 and 8.3 are directly relevant to this lecture.

In [CLRS], there is an entire chapter on "Hash Tables".

Chapter 14 of "Algorithms in Java" (3rd Ed), by Robert Sedgewick, has a *very* nice presentation of Hashing.

For information on the *HashMap* interface of JVC:
`http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashMap.html`

Exercises

1. For an ASCII character c, let $A(c)$ denote the ASCII code of c (an integer between 0 and 127). Let f, g be hash codes for ASCII strings defined by

$$\begin{aligned} f(c_0 \ldots, c_{n-1}) &= A(c_0) + \ldots + A(c_{n-1}), \\ g(c_0 \ldots, c_{n-1}) &= A(c_0) + A(c_1) \cdot 3 + \ldots + A(c_{n-1}) \cdot 3^{n-1}. \end{aligned}$$

   Compute f and g for a few example strings and write JAVA methods computing $f(s)$ and $g(s)$, respectively, for a given string.

2. Draw the 11–item hash table resulting from hashing keys with hash codes 12, 44, 12, 88, 23, 94, 11, 39, 20, 16, and 5 using the compression map $g(i) = (2i + 5)$ mod 11.