5.4 Hash tables

A **hash table** is a randomized data structure that supports the INSERT, DELETE, and FIND operations in expected $O(1)$ time. The core idea behind hash tables is to use a *hash function* that maps a large keyspace to a smaller domain of array indices, and then use constant-time array operations to store and retrieve the data.

5.4.1 Dictionary data types

A hash table is typically used to implement a **dictionary data type**, where keys are mapped to values, but unlike an array, the keys are not conveniently arranged as integers $0, 1, 2, \ldots$. Dictionary data types are a fundamental data structure often found in scripting languages like AWK, Perl, Python, PHP, Lua, or Ruby. For example, here is some Python code that demonstrates use of a dictionary accessed using an array-like syntax:

```
title = {}   # empty dictionary
title["Barack"] = "President"
user = "Barack"
print("Welcome" + title[user] + " " + user)
```

In C, we don't have the convenience of reusing `[]` for dictionary lookups (we'd need C++ for that), but we can still get the same effect with more typing using functions. For example, using an abstract dictionary in C might look like this:

```
Dict *title;
const char *user;

title = dictCreate();
dictSet(title, "Barack", "President");
user = "Barack";
printf("Welcome %s %s\n", dictGet(title, user), user);
```

As with other abstract data types, the idea is that the user of the dictionary type doesn't need to know how it is implemented. For example, we could implement the dictionary as an array of `structs` that we search through, but that would be expensive: $O(n)$ time to find a key in the worst case.

Closely related to a dictionary is a **set**, which has keys but no values. It's usually pretty straightforward to turn an implementation of a dictionary into a set (leave out the values) or vice versa (add values to the end of keys but don't use them in searching).

5.4.2 Basics of hashing

If our keys were conveniently named $0, 1, 2, \ldots, n-1$, we could simply use an array, and be able to find a record given a key in constant time. Unfortunately, naming conventions for most objects are not so convenient, and even enumerations like Social Security numbers are likely to span a larger range than we want to allocate. But we would like to get the constant-time performance of an array anyway.

The solution is to feed the keys through some hash function H, which maps them down to array indices. So in a database of people, to find "Smith, Wayland", we would first compute H("Smith, Wayland")\$ = 137\$ (say), and then look in position 137 in the array. Because we are always using the same function H, we will always be directed to the same position 137.

5.4.3 Resolving collisions

But what if H("Smith, Wayland") and H("Hephaestos") both equal 137? Now we have a **collision**, and we have to resolve it by finding some way to either (a) effectively store both records in a single array location, or (b) move one of the records to a new location that we can still find later. Let's consider these two approaches separately.

5.4.3.1 Chaining

We can't really store more than one record in an array location, but we can fake it by making each array location be a pointer to a linked list. Every time we insert a new element in a particular location, we simply add it to this list.

Since the cost of scanning a linked list is linear in its size, this means that the worst-case cost of searching for a particular key will be linear in the number of keys in the table that hash to the same location. Under the assumption that the hash function is a random function (which does not mean that it returns random values every time you call it but instead means that we picked one of the many possible hash functions uniformly at random), on average we get n/m elements in each list.
So on average a failed search takes $O(n/m)$ time.

This quantity n/m is called the **load factor** of the hash table and is often written as α. If we want our hash table to be efficient, we will need to keep this load factor down. If we can guarantee that it's a constant, then we get constant-time searches.

5.4.3.2 Open addressing

With *open addressing*, we store only one element per location, and handle collisions by storing the extra elements in other unused locations in the array.

To find these other locations, we fix some *probe sequence* that tells us where to look if $A[H(x)]$ contains an element that is not x. A typical probe sequence (called *linear probing*) is just $H(x), H(x)+1, H(x)+2, \ldots$, wrapping around at the end of the array. The idea is that if we can't put an element in a particular place, we just keep walking up through the array until we find an empty slot. As long as we follow the same probe sequence when looking for an element, we will be able to find the element again. If we are looking for an element and reach an empty location, then we know that the element is not present in the table.

For open addressing, we always have that $\alpha = n/m$ is less than or equal to 1, since we can't store more elements in the table than we have locations. In fact, we must ensure that the load factor is strictly less than 1, or some searches will never terminate because they never reach an empty location. Assuming $\alpha < 1$ and that the hash function is uniform, it is possible to calculate the worst-case expected cost of a FIND operation, which as before will occur when we have an unsuccessful FIND. Though we won't do this calculation here, the result is bounded by $1/(1 - n/m)$, which gets pretty bad if n/m is very close to 1, but is a constant as long as n/m is bounded by a constant (say 3/4, which makes the expected number of probes at most 4).

5.4.4 Choosing a hash function

Here we will describe three methods for generating hash functions. The first two are typical methods used in practice. The last has additional desirable theoretical properties.

5.4.4.1 Division method

We want our hash function to look as close as it can to a random function, but random functions are (provably) expensive to store. So in practice we do something simpler and hope for the best. If the keys are large integers, a typical approach is to just compute the remainder mod m. This can cause problems if m is, say, a power of 2, since it may be that the low-order bits of all the keys are similar, which will produce lots of collisions. So in practice with this method m is typically chosen to be a large prime.

What if we want to hash strings instead of integers? The trick is to treat the strings as integers. Given a string $a_1 a_2 a_3 \ldots a_k$, we represent it as $\sum_i a_i b^i$, where b is a base chosen to be larger than the number of characters. We can then feed this resulting huge integer to our hash function. Typically we do not actually compute the huge integer directly, but instead compute its remainder mod m, as in this short C function:

```
/* treat strings as base-256 integers */
/* with digits in the range 1 to 255 */
#define BASE (256)
```

```
size_t
hash(const char *s, size_t m)
{
    size_t h;
    unsigned const char *us;

    /* cast s to unsigned const char * */
    /* this ensures that elements of s will be treated as having values >= 0 */
    us = (unsigned const char *) s;

    h = 0;
    while(*us != '\0') {
        h = (h * BASE + *us) % m;
        us++;
    }

    return h;
}
```

The division method works best when m is a prime, as otherwise regularities in the keys can produce clustering in the hash values. (Consider, for example, what happens if m is 256). But this can be awkward for computing hash functions quickly, because computing remainders is a relatively slow operation.

5.4.4.2 Multiplication method

For this reason, the most commonly-used hash functions replace the modulus m with something like 2^32 and replace the base with some small prime, relying on the multiplier to break up patterns in the input. This yields the **multiplication method**. Typical code might look something like this:

```
#define MULTIPLIER (37)

size_t
hash(const char *s)
{
    size_t h;
    unsigned const char *us;

    /* cast s to unsigned const char * */
    /* this ensures that elements of s will be treated as having values >= 0 */
    us = (unsigned const char *) s;

    h = 0;
    while(*us != '\0') {
```

```
        h = h * MULTIPLIER + *us;
        us++;
    }

    return h;
}
```

The only difference between this code and the division method code is that we've renamed `BASE` to `MULTIPLIER` and dropped m. There is still some remainder-taking happening: since C truncates the result of any operation that exceeds the size of the integer type that holds it, the `h = h * MULTIPLIER + *us;` line effectively has a hidden mod 2^32 or 2^64 at the end of it (depending on how big your `size_t` is). Now we can't use, say, 256, as the multiplier, because then the hash value `h` would be determined by just the last four characters of `s`.

The choice of 37 is based on folklore. I like 97 myself, and 31 also has supporters. Almost any medium-sized prime should work.

5.4.4.3 Universal hashing

The preceding hash functions offer no guarantees that the adversary can't find a set of n keys that all hash to the same location; indeed, since they're deterministic, as long as the keyspace contains at least nm keys the adversary can always do so. Universal families of hash functions avoid this problem by choosing the hash function randomly, from some set of possible functions that is small enough that we can write our random choice down.

The property that makes a family of hash functions $\{H_r\}$ universal is that, for any distinct keys x and y, the probability that r is chosen so that $H_r(x) = H_r(y)$ is exactly $1/m$.

Why is this important? Recall that for chaining, the expected number of collisions between an element x and other elements was just the sum over all particular elements y of the probability that x collides with that particular element. If H_r is drawn from a universal family, this probability is $1/m$ for each y, and we get the same n/m expected collisions as if H_r were completely random.

Several universal families of hash functions are known. Here is a simple one that works when the size of the keyspace and the size of the output space are both powers of 2. Let the keyspace consist of n-bit strings and let $m = 2^k$. Then the random index r consists of nk independent random bits organized as n m-bit strings $a_1 a_2 \ldots a_n$. To compute the hash function of a particular input x, compute the bitwise exclusive or of a_i for each position i where the i-th bit of x is 1.

We can implement this in C as

```
/* implements universal hashing using random bit-vectors in x */
/* assumes number of elements in x is at least BITS_PER_CHAR * MAX_STRING_SIZE */
```

```
#define BITS_PER_CHAR (8)       /* not true on all machines! */
#define MAX_STRING_SIZE (128)   /* we'll stop hashing after this many */
#define MAX_BITS (BITS_PER_CHAR * MAX_STRING_SIZE)

size_t
hash(const char *s, size_t x[])
{
    size_t h;
    unsigned const char *us;
    int i;
    unsigned char c;
    int shift;

    /* cast s to unsigned const char * */
    /* this ensures that elements of s will be treated as having values >= 0 */
    us = (unsigned const char *) s;

    h = 0;
    for(i = 0; *us != 0 && i < MAX_BITS; us++) {
        c = *us;
        for(shift = 0; shift < BITS_PER_CHAR; shift++, i++) {
            /* is low bit of c set? */
            if(c & 0x1) {
                h ^= x[i];
            }

            /* shift c to get new bit in lowest position */
            c >>= 1;
        }
    }

    return h;
}
```

As you can see, this requires a lot of bit-fiddling. It also fails if we get a lot of strings that are identical for the first `MAX_STRING_SIZE` characters. Conceivably, the latter problem could be dealt with by growing `x` dynamically as needed. But we also haven't addressed the question of where we get these random values from—see the chapter on randomization for some possibilities.

In practice, universal families of hash functions are seldom used, since a reasonable fixed hash function is unlikely to be correlated with any patterns in the actual input. But they are useful for demonstrating provably good performance.

5.4.5 Maintaining a constant load factor

All of the running time results for hash tables depend on keeping the load factor α small. But as more elements are inserted into a fixed-size table, the load factor grows without bound. The usual solution to this problem is rehashing: when the load factor crosses some threshold, we create a new hash table of size $2n$ or thereabouts and migrate all the elements to it.

This approach raises the worst-case cost of an insertion to $O(n)$. However, we can bring the *expected* cost down to $O(1)$ by rehashing only with probability $O(1/n)$ for each insert after the threshold is crossed. Or we can apply **amortized analysis** to argue that the amortized cost (total cost divided by number of operations) is $O(1)$ assuming we double the table size on each rehash. Neither the expected-cost nor the amortized-cost approaches actually change the worst-case cost, but they make it look better by demonstrating that we at least don't incur that cost every time.

With enough machinery, it may be possible to **deamortize** the cost of rehashing by doing a little bit of it with every insertion. The idea is to build the new hash table incrementally, and start moving elements to it once it is fully initialized. This requires keeping around two copies of the hash table and searching both, and for most purposes is more trouble than it's worth. But a mechanism like this is often used for real-time garbage collection, where it's important not to have the garbage collector lock up the entire system while it does its work.

5.4.6 Examples

5.4.6.1 A low-overhead hash table using open addressing

Here is a very low-overhead hash table based on open addressing. The application is rapidly verifying ID numbers in the range 000000000 to 999999999 by checking them against a list of known good IDs. Since the quantity of valid ID numbers may be very large, a goal of the mechanism is to keep the amount of extra storage used as small as possible. This implementation uses a tunable overhead parameter. Setting the parameter to a high value makes lookups fast but requires more space per ID number in the list. Setting it to a low value can reduce the storage cost arbitrarily close to 4 bytes per ID, at the cost of increasing search times.

Here is the header file giving the interface:

```
typedef struct idList *IDList;

#define MIN_ID (0)
#define MAX_ID (999999999)

/* build an IDList out of an unsorted array of n good ids */
```

```
/* returns 0 on allocation failure */
IDList IDListCreate(int n, int unsortedIdList[]);

/* destroy an IDList */
void IDListDestroy(IDList list);

/* check an id against the list */
/* returns nonzero if id is in the list */
int IDListContains(IDList list, int id);
```

examples/hashTables/idList/idList.h

And here is the implementation:

```
#include <stdlib.h>
#include <assert.h>

#include "idList.h"

/* overhead parameter that determines both space and search costs */
/* must be strictly greater than 1 */
#define OVERHEAD (1.1)
#define NULL_ID (-1)


struct idList {
    int size;
    int ids[1];         /* we'll actually malloc more space than this */
};

IDList
IDListCreate(int n, int unsortedIdList[])
{
    IDList list;
    int size;
    int i;
    int probe;

    size = (int) (n * OVERHEAD + 1);

    list = malloc(sizeof(*list) + sizeof(int) * (size-1));
    if(list == 0) return 0;

    /* else */
    list->size = size;

    /* clear the hash table */
```

```
    for(i = 0; i < size; i++) {
        list->ids[i] = NULL_ID;
    }

    /* load it up */
    for(i = 0; i < n; i++) {

        assert(unsortedIdList[i] >= MIN_ID);
        assert(unsortedIdList[i] <= MAX_ID);

        /* hashing with open addressing by division */
        /* this MUST be the same pattern as in IDListContains */
        for(probe = unsortedIdList[i] % list->size;
            list->ids[probe] != NULL_ID;
            probe = (probe + 1) % list->size);

        assert(list->ids[probe] == NULL_ID);

        list->ids[probe] = unsortedIdList[i];
    }

    return list;
}

void
IDListDestroy(IDList list)
{
    free(list);
}

int
IDListContains(IDList list, int id)
{
    int probe;

    /* this MUST be the same pattern as in IDListCreate */
    for(probe = id % size;
        list->ids[probe] != NULL_ID;
        probe = (probe + 1) % size) {
        if(list->ids[probe] == id) {
            return 1;
        }
    }

    return 0;
}
```

examples/hashTables/idList/idList.c

5.4.6.2 A string to string dictionary using chaining

Here is a more complicated string to string dictionary based on chaining.

```
typedef struct dict *Dict;

/* create a new empty dictionary */
Dict DictCreate(void);

/* destroy a dictionary */
void DictDestroy(Dict);

/* insert a new key-value pair into an existing dictionary */
void DictInsert(Dict, const char *key, const char *value);

/* return the most recently inserted value associated with a key */
/* or 0 if no matching key is present */
const char *DictSearch(Dict, const char *key);

/* delete the most recently inserted record with the given key */
/* if there is no such record, has no effect */
void DictDelete(Dict, const char *key);
```

examples/hashTables/dict/dict.h

```
#include <stdlib.h>
#include <assert.h>
#include <string.h>

#include "dict.h"

struct elt {
    struct elt *next;
    char *key;
    char *value;
};

struct dict {
    int size;           /* size of the pointer table */
    int n;              /* number of elements stored */
    struct elt **table;
};

#define INITIAL_SIZE (1024)
#define GROWTH_FACTOR (2)
```

```
#define MAX_LOAD_FACTOR (1)

/* dictionary initialization code used in both DictCreate and grow */
Dict
internalDictCreate(int size)
{
    Dict d;
    int i;

    d = malloc(sizeof(*d));

    assert(d != 0);

    d->size = size;
    d->n = 0;
    d->table = malloc(sizeof(struct elt *) * d->size);

    assert(d->table != 0);

    for(i = 0; i < d->size; i++) d->table[i] = 0;

    return d;
}

Dict
DictCreate(void)
{
    return internalDictCreate(INITIAL_SIZE);
}

void
DictDestroy(Dict d)
{
    int i;
    struct elt *e;
    struct elt *next;

    for(i = 0; i < d->size; i++) {
        for(e = d->table[i]; e != 0; e = next) {
            next = e->next;

            free(e->key);
            free(e->value);
            free(e);
        }
    }
```

```
    free(d->table);
    free(d);
}

#define MULTIPLIER (97)

static unsigned long
hash_function(const char *s)
{
    unsigned const char *us;
    unsigned long h;

    h = 0;

    for(us = (unsigned const char *) s; *us; us++) {
        h = h * MULTIPLIER + *us;
    }

    return h;
}

static void
grow(Dict d)
{
    Dict d2;            /* new dictionary we'll create */
    struct dict swap;   /* temporary structure for brain transplant */
    int i;
    struct elt *e;

    d2 = internalDictCreate(d->size * GROWTH_FACTOR);

    for(i = 0; i < d->size; i++) {
        for(e = d->table[i]; e != 0; e = e->next) {
            /* note: this recopies everything */
            /* a more efficient implementation would
             * patch out the strdups inside DictInsert
             * to avoid this problem */
            DictInsert(d2, e->key, e->value);
        }
    }

    /* the hideous part */
    /* We'll swap the guts of d and d2 */
    /* then call DictDestroy on d2 */
    swap = *d;
```

```
    *d = *d2;
    *d2 = swap;

    DictDestroy(d2);
}

/* insert a new key-value pair into an existing dictionary */
void
DictInsert(Dict d, const char *key, const char *value)
{
    struct elt *e;
    unsigned long h;

    assert(key);
    assert(value);

    e = malloc(sizeof(*e));

    assert(e);

    e->key = strdup(key);
    e->value = strdup(value);

    h = hash_function(key) % d->size;

    e->next = d->table[h];
    d->table[h] = e;

    d->n++;

    /* grow table if there is not enough room */
    if(d->n >= d->size * MAX_LOAD_FACTOR) {
        grow(d);
    }
}

/* return the most recently inserted value associated with a key */
/* or 0 if no matching key is present */
const char *
DictSearch(Dict d, const char *key)
{
    struct elt *e;

    for(e = d->table[hash_function(key) % d->size]; e != 0; e = e->next) {
        if(!strcmp(e->key, key)) {
            /* got it */
```

```
            return e->value;
        }
    }

    return 0;
}

/* delete the most recently inserted record with the given key */
/* if there is no such record, has no effect */
void
DictDelete(Dict d, const char *key)
{
    struct elt **prev;          /* what to change when elt is deleted */
    struct elt *e;              /* what to delete */

    for(prev = &(d->table[hash_function(key) % d->size]);
        *prev != 0;
        prev = &((*prev)->next)) {
        if(!strcmp((*prev)->key, key)) {
            /* got it */
            e = *prev;
            *prev = e->next;

            free(e->key);
            free(e->value);
            free(e);

            return;
        }
    }
}
```

examples/hashTables/dict/dict.c

And here is some (very minimal) test code.

```
#include <stdio.h>
#include <assert.h>

#include "dict.h"

int
main()
{
    Dict d;
    char buf[512];
    int i;
```

```
    d = DictCreate();

    DictInsert(d, "foo", "hello world");
    puts(DictSearch(d, "foo"));
    DictInsert(d, "foo", "hello world2");
    puts(DictSearch(d, "foo"));
    DictDelete(d, "foo");
    puts(DictSearch(d, "foo"));
    DictDelete(d, "foo");
    assert(DictSearch(d, "foo") == 0);
    DictDelete(d, "foo");

    for(i = 0; i < 10000; i++) {
        sprintf(buf, "%d", i);
        DictInsert(d, buf, buf);
    }

    DictDestroy(d);

    return 0;
}
```

examples/hashTables/dict/test_dict.c

5.5 Generic containers

The first rule of programming is that you should never write the same code twice. Suppose that you happen to have lying around a dictionary type whose keys are `int`s and whose values are strings. Tomorrow you realize that what you really want is a dictionary type whose keys are strings and whose values are `int`s, or one whose keys are `int`s but whose values are stacks. If you have n different types that may appear as keys or values, can you avoid writing n^2 different dictionary implementations to get every possible combination?

Many languages provide special mechanisms to support **generic types**, ones for which part of the type is not specified. It's as if you could declare an array in C to be an array of some type to be specified later, and then write functions that operate on any such array without knowing what the missing type is going to be (**templates** in C++ are an example of such a mechanism). Unfortunately, C does not provide generic types. But by aggressive use of function pointers and `void *`, it is possible to fake them.