C/HashTables

[FrontPage] [TitleIndex] [WordIndex]

Note: You are looking at a static copy of the former PineWiki site, used for class notes by James Aspnes from 2003 to 2012. Many mathematical formulas are broken, and there are likely to be other bugs as well. These will most likely not be fixed. You may be able to find more up-to-date versions of some of these notes at http://www.cs.yale.edu/homes/aspnes/#classes.

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.

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, ... Dictionary data types are a fundamental data structure often found in WikiPedia: scripting languages like WikiPedia: AWK, WikiPedia: Perl, WikiPedia: Python, WikiPedia: PHP, WikiPedia: Lua, or WikiPedia: Ruby. For example, here is some Python code that demonstrates use of a dictionary accessed using an array-like syntax:

Toggle line numbers

```
1 title = {}    # empty dictionary
2 title["Barack"] = "President"
3 user = "Barack"
4 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:

Toggle line numbers

```
1 Dict *title;
2 const char *user;
3 title = dictCreate();
4 dictSet(title, "Barack", "President");
5 user = "Barack";
6 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.

2. Basics of hashing

If our keys were conveniently named 0, 1, 2, ..., 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.

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.

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), we can analyze the expected cost of a failed search as a function of the load factor $\alpha = n/m$, where n is the number of elements in the table and m is the number of locations. We have

$$\begin{aligned}&\text{E[number of elements hashed to the same location as x]}\\ &= \sum_{\text{all elements y}} \text{Pr[y is hashed to the same location as x]} = n * (1/m) = n/m.\end{aligned}$$

So as long as the number of elements is proportional to the available space, we get constant-time FIND operations.

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, ..., 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, we can calculate the worst-case expected cost of a FIND operation, which as before will occur when we have an unsuccessful FIND.

Let T(n,m) be the expected number of probes in an unsuccessful search in a hash table with open addressing, n elements, and m locations. We always do at least one probe. With probability n/m we found something and have to try again in the next location, at cost T(n-1,m-1). So we have

$$T(n,m) = 1 + (n/m)\, T(n-1,m-1)$$

and

$$T(0,m) = 1.$$

This is an annoying recurrence to have to solve exactly. So instead we will get an upper bound by observing that the probability that we keep going is always less than or equal to n/m (since $(n-i)/(m-i) < n/m$ when $n < m$). If we further leave off the case where $m = 0$, we get a coin-flipping problem where we are waiting to see a coin with probability n/m of coming up heads come up tails. This has the much simpler recurrence

$$T = 1 + (n/m)\, T$$

for which the solution is

$$T = 1/(1-n/m).$$

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.

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 $a_1a_2a_3...a_k$, 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:

Toggle line numbers

```
 1 /* treat strings as base-256 integers */
 2 /* with digits in the range 1 to 255 */
 3 #define BASE (256)
 4
 5 unsigned long
 6 hash(const char *s, unsigned long m)
 7 {
 8     unsigned long h;
 9     unsigned const char *us;
10
11     /* cast s to unsigned const char * */
12     /* this ensures that elements of s will be treated as
having values >= 0 */
13     us = (unsigned const char *) s;
14
15     h = 0;
16     while(*us != '\0') {
17         h = (h * BASE + *us) % m;
18         us++;
19     }
20
21     return h;
22 }
```

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` equals 256). But this can be awkward for computing hash functions quickly, because computing remainders is a relatively slow operation.

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:

Toggle line numbers

```
 1 #define MULTIPLIER (37)
 2
 3 unsigned long
 4 hash(const char *s)
 5 {
 6     unsigned long h;
 7     unsigned const char *us;
 8
 9     /* cast s to unsigned const char * */
10     /* this ensures that elements of s will be treated as
having values >= 0 */
11     us = (unsigned const char *) s;
12
13     h = 0;
14     while(*us != '\0') {
15         h = h * MULTIPLIER + *us;
16         us++;
17     }
18
19     return h;
20 }
```

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 `unsigned long`s are). 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.

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 $a_1,a_2,...a_n$. 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 a 1. Formally, using XOR to mean bitwise exclusive or, we might write this as

$$XOR_i\ x_i * a_i.$$

We can implement this in C as

Toggle line numbers

```
   1 /* implements universal hashing using random bit-vectors in x
*/
   2 /* assumes number of elements in x is at least BITS_PER_CHAR *
MAX_STRING_SIZE */
   3
   4 #define BITS_PER_CHAR (8)        /* not true on all machines! */
   5 #define MAX_STRING_SIZE (128)    /* we'll stop hashing after
this many */
   6 #define MAX_BITS (BITS_PER_CHAR * MAX_STRING_SIZE)
   7
   8 unsigned long
   9 hash(const char *s, unsigned long x[])
  10 {
  11     unsigned long h;
  12     unsigned const char *us;
  13     int i;
  14     unsigned char c;
  15     int shift;
  16
  17     /* cast s to unsigned const char * */
  18     /* this ensures that elements of s will be treated as
having values >= 0 */
  19     us = (unsigned const char *) s;
  20
  21     h = 0;
  22     for(i = 0; *us != 0 && i < MAX_BITS; us++) {
  23         c = *us;
  24         for(shift = 0; shift < BITS_PER_CHAR; shift++, i++) {
  25             /* is low bit of c set? */
  26             if(c & 0x1) {
  27                 h ^= x[i];
  28             }
  29
  30             /* shift c to get new bit in lowest position */
  31             c >>= 1;
  32         }
  33     }
  34
```

```
35      return h;
36  }
```

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 C/Randomization for some possibilities.

Why is this family universal? Consider two distinct inputs x and y. Because they are distinct, there must be some position j where the bits x_j and y_j are different. Assume without loss of generality that x_j is zero and y_j is 1. Let

$$X = \text{XOR}_{i \mathrel{!}= j} x_i a_i$$

and

$$Y = \text{XOR}_{i \mathrel{!}= j} y_i a_i,$$

so that H(x) = X and H(y) = Y XOR a_j. Suppose that we fix all the bits except for the ones in a_j; then H(x) = H(y) precisely when a_j = X XOR Y, where the right-hand side is some constant value independent of a_j. The probability that a_j is chosen to be exactly this value is 1/m.

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. 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 can apply AmortizedAnalysis 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.

6. Examples

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.

This is `idlist.h`:

Toggle line numbers

```
 1 typedef struct id_list *IDList;
 2
 3 #define MIN_ID (0)
 4 #define MAX_ID (999999999)
 5
 6 /* build an IDList out of an unsorted array of n good ids */
 7 /* returns 0 on allocation failure */
 8 IDList IDListCreate(int n, int unsorted_id_list[]);
 9
10 /* destroy an IDList */
11 void IDListDestroy(IDList list);
12
13 /* check an id against the list */
14 /* returns nonzero if id is in the list */
15 int IDListContains(IDList list, int id);
```

And this is `idlist.c`:

Toggle line numbers

```
 1 #include <stdlib.h>
 2 #include <assert.h>
 3 #include "idlist.h"
 4
 5 /* overhead parameter that determines both space and search
costs */
 6 /* must be strictly greater than 1 */
 7 #define OVERHEAD (1.1)
 8 #define NULL_ID (-1)
 9
10
11 struct id_list {
12     int size;
13     int ids[1];         /* we'll actually malloc more space
than this */
14 };
15
16 IDList
17 IDListCreate(int n, int unsorted_id_list[])
18 {
19     IDList list;
20     int size;
21     int i;
22     int probe;
```

```
 23
 24      size = (int) (n * OVERHEAD + 1);
 25
 26      list = malloc(sizeof(*list) + sizeof(int) * (size-1));
 27      if(list == 0) return 0;
 28
 29      /* else */
 30      list->size = size;
 31
 32      /* clear the hash table */
 33      for(i = 0; i < size; i++) {
 34          list->ids[i] = NULL_ID;
 35      }
 36
 37      /* load it up */
 38      for(i = 0; i < n; i++) {
 39
 40          assert(unsorted_id_list[i] >= MIN_ID);
 41          assert(unsorted_id_list[i] <= MAX_ID);
 42
 43          /* hashing with open addressing by division */
 44          /* this MUST be the same pattern as in IDListContains
*/
 45          for(probe = unsorted_id_list[i] % list->size;
 46              list->ids[probe] != NULL_ID;
 47              probe = (probe + 1) % list->size);
 48
 49          assert(list->ids[probe] == NULL_ID);
 50
 51          list->ids[probe] = unsorted_id_list[i];
 52      }
 53
 54      return list;
 55  }
 56
 57  void
 58  IDListDestroy(IDList list)
 59  {
 60      free(list);
 61  }
 62
 63  int
 64  IDListContains(IDList list, int id)
 65  {
 66      int probe;
 67
 68      /* this MUST be the same pattern as in IDListCreate */
 69      for(probe = id % size;
 70          list->ids[probe] != NULL_ID;
 71          probe = (probe + 1) % size) {
 72          if(list->ids[probe] == id) {
 73              return 1;
 74          }
 75      }
 76
 77      return 0;
 78  }
```

6.2. A string to string dictionary using chaining

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

Toggle line numbers

```
 1 typedef struct dict *Dict;
 2
 3 /* create a new empty dictionary */
 4 Dict DictCreate(void);
 5
 6 /* destroy a dictionary */
 7 void DictDestroy(Dict);
 8
 9 /* insert a new key-value pair into an existing dictionary */
10 void DictInsert(Dict, const char *key, const char *value);
11
12 /* return the most recently inserted value associated with a
key */
13 /* or 0 if no matching key is present */
14 const char *DictSearch(Dict, const char *key);
15
16 /* delete the most recently inserted record with the given key
*/
17 /* if there is no such record, has no effect */
18 void DictDelete(Dict, const char *key);
```

dict.h

Toggle line numbers

```
 1 #include <stdlib.h>
 2 #include <assert.h>
 3 #include <string.h>
 4
 5 #include "dict.h"
 6
 7 struct elt {
 8     struct elt *next;
 9     char *key;
10     char *value;
11 };
12
13 struct dict {
14     int size;           /* size of the pointer table */
15     int n;              /* number of elements stored */
16     struct elt **table;
17 };
18
19 #define INITIAL_SIZE (1024)
20 #define GROWTH_FACTOR (2)
21 #define MAX_LOAD_FACTOR (1)
22
```

```
 23 /* dictionary initialization code used in both DictCreate and
grow */
 24 Dict
 25 internalDictCreate(int size)
 26 {
 27     Dict d;
 28     int i;
 29
 30     d = malloc(sizeof(*d));
 31
 32     assert(d != 0);
 33
 34     d->size = size;
 35     d->n = 0;
 36     d->table = malloc(sizeof(struct elt *) * d->size);
 37
 38     assert(d->table != 0);
 39
 40     for(i = 0; i < d->size; i++) d->table[i] = 0;
 41
 42     return d;
 43 }
 44
 45 Dict
 46 DictCreate(void)
 47 {
 48     return internalDictCreate(INITIAL_SIZE);
 49 }
 50
 51 void
 52 DictDestroy(Dict d)
 53 {
 54     int i;
 55     struct elt *e;
 56     struct elt *next;
 57
 58     for(i = 0; i < d->size; i++) {
 59         for(e = d->table[i]; e != 0; e = next) {
 60             next = e->next;
 61
 62             free(e->key);
 63             free(e->value);
 64             free(e);
 65         }
 66     }
 67
 68     free(d->table);
 69     free(d);
 70 }
 71
 72 #define MULTIPLIER (97)
 73
 74 static unsigned long
 75 hash_function(const char *s)
 76 {
 77     unsigned const char *us;
 78     unsigned long h;
```

```
 79
 80      h = 0;
 81
 82      for(us = (unsigned const char *) s; *us; us++) {
 83          h = h * MULTIPLIER + *us;
 84      }
 85
 86      return h;
 87  }
 88
 89  static void
 90  grow(Dict d)
 91  {
 92      Dict d2;            /* new dictionary we'll create */
 93      struct dict swap;   /* temporary structure for brain
transplant */
 94      int i;
 95      struct elt *e;
 96
 97      d2 = internalDictCreate(d->size * GROWTH_FACTOR);
 98
 99      for(i = 0; i < d->size; i++) {
100          for(e = d->table[i]; e != 0; e = e->next) {
101              /* note: this recopies everything */
102              /* a more efficient implementation would
103               * patch out the strdups inside DictInsert
104               * to avoid this problem */
105              DictInsert(d2, e->key, e->value);
106          }
107      }
108
109      /* the hideous part */
110      /* We'll swap the guts of d and d2 */
111      /* then call DictDestroy on d2 */
112      swap = *d;
113      *d = *d2;
114      *d2 = swap;
115
116      DictDestroy(d2);
117  }
118
119  /* insert a new key-value pair into an existing dictionary */
120  void
121  DictInsert(Dict d, const char *key, const char *value)
122  {
123      struct elt *e;
124      unsigned long h;
125
126      assert(key);
127      assert(value);
128
129      e = malloc(sizeof(*e));
130
131      assert(e);
132
133      e->key = strdup(key);
134      e->value = strdup(value);
```

```
 135
 136     h = hash_function(key) % d->size;
 137
 138     e->next = d->table[h];
 139     d->table[h] = e;
 140
 141     d->n++;
 142
 143     /* grow table if there is not enough room */
 144     if(d->n >= d->size * MAX_LOAD_FACTOR) {
 145         grow(d);
 146     }
 147 }
 148
 149 /* return the most recently inserted value associated with a
key */
 150 /* or 0 if no matching key is present */
 151 const char *
 152 DictSearch(Dict d, const char *key)
 153 {
 154     struct elt *e;
 155
 156     for(e = d->table[hash_function(key) % d->size]; e != 0; e =
e->next) {
 157         if(!strcmp(e->key, key)) {
 158             /* got it */
 159             return e->value;
 160         }
 161     }
 162
 163     return 0;
 164 }
 165
 166 /* delete the most recently inserted record with the given key
*/
 167 /* if there is no such record, has no effect */
 168 void
 169 DictDelete(Dict d, const char *key)
 170 {
 171     struct elt **prev;          /* what to change when elt is
deleted */
 172     struct elt *e;              /* what to delete */
 173
 174     for(prev = &(d->table[hash_function(key) % d->size]);
 175         *prev != 0;
 176         prev = &((*prev)->next)) {
 177         if(!strcmp((*prev)->key, key)) {
 178             /* got it */
 179             e = *prev;
 180             *prev = e->next;
 181
 182             free(e->key);
 183             free(e->value);
 184             free(e);
 185
 186             return;
 187         }
```

```
188     }
189 }
```

dict.c

And here is some (very minimal) test code.

Toggle line numbers

```
 1 #include <stdio.h>
 2 #include <assert.h>
 3
 4 #include "dict.h"
 5
 6 int
 7 main()
 8 {
 9     Dict d;
10     char buf[512];
11     int i;
12
13     d = DictCreate();
14
15     DictInsert(d, "foo", "hello world");
16     puts(DictSearch(d, "foo"));
17     DictInsert(d, "foo", "hello world2");
18     puts(DictSearch(d, "foo"));
19     DictDelete(d, "foo");
20     puts(DictSearch(d, "foo"));
21     DictDelete(d, "foo");
22     assert(DictSearch(d, "foo") == 0);
23     DictDelete(d, "foo");
24
25     for(i = 0; i < 10000; i++) {
26         sprintf(buf, "%d", i);
27         DictInsert(d, buf, buf);
28     }
29
30     DictDestroy(d);
31
32     return 0;
33 }
34
35
```

test_dict.c

CategoryProgrammingNotes CategoryAlgorithmNotes

2014-06-17 11:57