Perfect Hashing: FKS Method

Nitin Verma
mathsanew.com

March 3, 2021

In a hashtable, the set of all possible keys is larger than the size of the hashtable. So, if the actual input keys are not known in advance, we can't be sure if they would collide under any given hash-function. But if the input keys are known in advance — we call them "static-keys" — is it possible to find a hash-function giving no collision for these keys? We may call such function a "Collision-Free Function" for the given keys, and such method of hashing "Perfect Hashing". With this hashing, the worst-case time complexity for finding a key is $O(1)$, because there are no collisions.

In this article we discuss a method of Perfect Hashing proposed by Fredman, Komlós and Szemerédi in [1], often called "FKS Method". Book [2]'s chapter "Hash Tables", also provides a good description of this method.

Searching a Collision-Free Function

Say, we are given the set of static-keys S having n keys and we need to find a collision-free hash-function mapping them to $\mathbb{Z}_m = \{0, 1, 2, \ldots, m-1\}$. How can we *search* for such a function? From which collection of functions can we hope to find it, possibly after some trial-and-error?

What helps us here is the *Universal Classes* of hash-functions. They are a collection of hash-functions with the property that any two distinct keys can collide only under a specified maximum number of functions. Please refer to the article titled *Universal Classes of Hash Functions* ([3]) for the definition of Universal Class and ϵ-Universal Class. In this discussion, we will make use of some relations about these classes as derived in that article.

Theorem 1 in [3] tells us that, for any ϵ-Universal class and given any set S of n keys, we are guaranteed to find a certain proportion of collision-free functions in the class, if ϵ and n are related as specified. Specifically, the

subsequent Corollary 2 ensures us that for any $1/m$-Universal class, if we choose $m \geq n(n-1)$, at least half of its functions must be collision-free for S.

This is really useful for our problem. We can choose any $1/m$-Universal class and use the hashtable size $m = n(n-1)$. Then we can randomly select a function from this class and hash all the keys from S to see if there are any collisions. We can repeat until we find a function without collision. The above theorem ensures that we should succeed in a very few attempts because half of the functions are what we are looking for.

Two-Level Hashing

Above method does help us to find a collision-free function in simple steps quickly. But its drawback is that it requires the hashtable to have size m in $O(n^2)$, which may be acceptable only when n is small enough. To solve this issue, we use two level of hashing whenever n is not sufficiently small. The level-1 hash-function h_1 (subscript '1' indicates level-1) maps the n keys of S to $\mathbb{Z}_m$. We will use S_i to denote the set of keys which get mapped to slot i by h_1, and s_i to denote $|S_i|$, for each $i \in \mathbb{Z}_m$.

For any slot i with $s_i \geq 2$, there are collisions. To handle such collisions, we use level-2 hashtables T_i, one for each slot i. For each slot with $s_i \geq 2$, we map its s_i keys to hashtable T_i, using a collision-free hash-function $h_{2,i}$ (subscript '2' indicates level-2). Using the method from the last section, we can search for a collision-free function, given the set of keys S_i. That is, we first choose a $1/m_i$-Universal class with table-size $m_i = s_i(s_i - 1)$ and search in it for the function which is collision-free for keys in S_i. For the $h_{2,i}$ function thus found, we will store its necessary details somewhere, to allow its reconstruction later.

For slots with $s_i = 1$, the table T_i has a single key and hence does not require hashing. For slots with $s_i = 0$, table T_i is simply empty.

To lookup any key x, we first compute $i \leftarrow h_1(x)$. If $s_i \geq 2$, we compute $j \leftarrow h_{2,i}(x)$. Else if $s_i = 1, j \leftarrow 0$. We can then return the key from the level-2 hashtable T_i, from its slot j. If $s_i = 0$, we conclude that the key is not found. Note that if the lookups may also be performed for keys outside the static set S, we must also compare the key thus found with key x to ensure they are equal. Because, other keys outside of set S may still *map* to the same location as any key from S.

To implement this approach, we don't need to keep a level-1 hashtable, and level-2 hashtables (T_i) separately. We can simply maintain a single table

T which is a concatenation of tables $T_0, T_1, T_2, \ldots, T_{m-1}$, in that order. The offsets of all these T_i tables in T will be calculated and stored somewhere. So, the slot j of table T_i is located at this index of T: (offset of T_i) + j.

Number of elements required in T, corresponding to each $T_i, i \in \mathbb{Z}_m$ is:

$$e_i = \begin{cases} s_i, & s_i = 0 \text{ or } 1 \\ s_i(s_i - 1), & s_i \geq 2 \end{cases}$$

Note that, for $s_i \geq 2$, $e_i = s_i(s_i - 1) \leq s_i^2$. Also, for $s_i = 0$ or 1, $e_i = s_i = s_i^2$. Thus, the total number of elements in T are:

$$\sum_{i \in \mathbb{Z}_m} e_i \leq \sum_{i \in \mathbb{Z}_m} s_i^2$$

The last section's method required space in $O(n^2)$. What can we say about the space complexity of table T? Note that if h_1 maps all n keys to one slot, $\sum_{i \in \mathbb{Z}_m} s_i^2 = n^2$, and if it maps each to a different slot, $\sum_{i \in \mathbb{Z}_m} s_i^2 = n$.

We can take help from Corollary 4 in [3]. We can search for h_1 in a $1/m$-Universal class ($\epsilon = 1/m$). That corollary then ensures that for at least half of the functions in the class, $\sum_{i \in \mathbb{Z}_m} s_i^2$ has the given bound:

$$\sum_{i \in \mathbb{Z}_m} s_i^2 < \frac{2n(n-1)}{m} + n$$

Choosing $m = n$, we get the bound as $3n - 2$, which is $O(n)$ and hence very useful. It tells that at least half of the functions in the class will keep the table T size within $3n$. So, we can randomly select a function from this class and hash all the keys from S to see if this $3n$ bound is met. We can repeat until we find such a function, and then use it as h_1.

We could thus create an injective ("collision-free") mapping of an arbitrary set of n keys to the indices in table T, which are at most $3n$ integers $\{0, 1, 2, ..., 3n - 1\}$. This is an interesting achievement of the FKS method.

Variations and an Implementation

Note that h_1 and $h_{2,i}$ can be searched in any ϵ-Universal classes. In above discussion, we just considered classes with $\epsilon = 1/\mathit{TableSize}$, for both level-1 and level-2 hashing. If we use any other ϵ-Universal class for level-2 hashing, then the condition of Theorem 1 in [3] needs to be evaluated with the

corresponding ϵ, to find the new condition which ensures certain collision-free functions. Similarly, if we use any other ϵ-Universal class for level-1 hashing, the Corollary 4 in [3] may give a different upper-bound for the sum $\sum_{i \in \mathbb{Z}_m} s_i^2$. The size of table T would be dependent upon this sum, similar to what we saw above.

The following source code in C demonstrates an implementation of this method for keys of type "unsigned int". The universal class used for selecting h_1 is from example-1 mentioned in page-2 of [3], with $m = n$. The same class type with same p but $m = m_i = s_i(s_i - 1)$ is used for selecting functions $h_{2,i}$. Depending upon the requirement (e.g. different key type), an implementation can choose different universal classes for level-1 and level-2 functions.

Any symbol of the form 'a_b' from the above discussion has been shown as '`a{b}`' in the code comments.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#define N 100           /* size of the set S of static-keys */

/* the keys can be any 32-bit integer, so choosing 'p' as the smallest prime
   greater than 2^32-1 */
#define P 4294967311    /* 2^32 + 15 */

typedef unsigned int uint;

typedef struct object
{
  uint id;
  /* other data */
} object;

typedef struct hashtable
{
  uint   m;
  uint   a1, b1;                    /* for function h{1} */
  uint   a2[N], b2[N];            /* for functions h{2,i} */
  object **T;
  int    T_offset[N+1];
  int    T_size;
} hashtable;

/* "id" of these N objects are all distinct, forming the static-keys set */
object    objects[N];
hashtable ht;
int       *ht_s;
```

```
/* function: h(k) = ((ak+b) mod p) mod TableSize */
#define hash(k, a, b, tsize) (((((unsigned long)a)*k + b) %(P)) %(tsize))

void construct_hashtable()
{
  int i;
  /* the size of each subset S{i} belonging to a level-1 slot; required
     only during this method. */
  int s[N];

  srand(time(NULL));
  ht_s = s;
  ht.m = N;

  /* search for level-1 hash-function h{1} */
  search_h1();

  /* offsets of tables T{i} in T; returns sum of T{i} sizes */
  ht.T_size = calculate_offsets();

  ht.T = malloc(ht.T_size*sizeof(object *));
  memset(ht.T, 0, ht.T_size*sizeof(object *));

  /* distribute input keys among subsets S{i}, using hash-function h{1} */
  create_subsets_of_keys();

  for(i = 0; i < ht.m; i++)
  {
    if(ht_s[i] < 2)
      continue;

    /* search for level-2 hash-function h{2,i} */
    search_h2i(i);

    /* hash subset S{i} to hashtable T{i}, using hash-function h{2,i} */
    hash_subset_of_keys(i);
  }
}
```

```
void search_h1()
{
  int a, b;

  while(1)
  {
    a = 1 + rand();
    b = rand();

    /* check if the randomly selected function satisfies the upper-bound
       3N on the sum of s{i}*s{i} */
    if(check_si_sqr_sum(a, b))
      break;
  }

  ht.a1 = a;
  ht.b1 = b;
}

int check_si_sqr_sum(uint a, uint b)
{
  int i, h;
  int si_sqr_sum = 0;

  memset(ht_s, 0, N*sizeof(int));

  for(i = 0; i < N; i++)
  {
    h = hash(objects[i].id, a, b, ht.m);
    ht_s[h]++;
  }

  for(i = 0; i < N; i++)
    si_sqr_sum += ht_s[i]*ht_s[i];

  return si_sqr_sum < 3*N;
}
```

```
int calculate_offsets()
{
  int i;
  int lastsize = 0;

  ht.T_offset[0] = 0;

  for(i = 0; i < N; i++)
  {
    if(i > 0)
      ht.T_offset[i] = ht.T_offset[i-1] + lastsize;

    if(ht_s[i] > 1)
      lastsize = ht_s[i]*(ht_s[i]-1);
    else
      lastsize = ht_s[i];
  }

  /* extra element T_offset[N] is for simplifying macro SIZE_T() */
  ht.T_offset[N] = ht.T_offset[N-1] + lastsize;

  return ht.T_offset[N];
}

void create_subsets_of_keys()
{
  int i, h1, offset;
  int current_size[N];

  memset(current_size, 0, sizeof(current_size));

  /* we can use the initial part of hashtable T{i} to store subset S{i},
     till we hash its keys using level-2 hashing. */
  for(i = 0; i < N; i++)
  {
    h1 = hash(objects[i].id, ht.a1, ht.b1, ht.m);

    offset = ht.T_offset[h1];

    ht.T[offset + current_size[h1]] = &objects[i];

    current_size[h1]++;
  }
}
```

```
void search_h2i(int i)
{
  int a, b;
  int si = ht_s[i];
  int offset = ht.T_offset[i];
  int tsize = si*(si-1);

  /* the bitmap required for checking collision-free function */
  char *buffer;
  int  buffer_size = (tsize/8) + 1;
  char tmp;

  /* Each subset S{i} is first stored in table T{i} (which is in T) at its
     initial "si" elements, by create_subsets_of_keys(). Its keys are later
     hashed by h{2,i} among the "si*(si-1)" elements of T{i} (if si >= 2).

     For searching collision-free function for S{i}, we need a bitmap of
     size tsize. For this, we utilize the CURRENTLY unused space of T{i},
     which is after the index "offset+si-1". But for si=2, si*(si-1)=si,
     so there is no such unused space. */

  if(si == 2)
    buffer = &tmp;
  else
    buffer = (char*) &(ht.T[offset+si]);

  while(1)
  {
    memset(buffer, 0, buffer_size);

    a = 1 + rand();
    b = rand();

    /* check if the randomly selected function is collision-free for
       subset S{i} */
    if(check_collision_free(i, a, b, buffer))
      break;
  }

  ht.a2[i] = a;
  ht.b2[i] = b;

  if(si != 2)
    memset(buffer, 0, buffer_size);
}
```

```
#define CHECK_BIT(arr, i) (arr[i/8] & (1 << (i%8)))
#define SET_BIT(arr, i)   (arr[i/8] |= (1 << (i%8)))

int check_collision_free(int l1slot, uint a, uint b, char *buffer)
{
  int i, h;
  int offset = ht.T_offset[l1slot];
  int si = ht_s[l1slot];

  for(i = offset; i < offset + si; i++)
  {
    h = hash(ht.T[i]->id, a, b, (si*(si-1)));

    if(CHECK_BIT(buffer, h))
      return 0;

    SET_BIT(buffer, h);
  }

  return 1;
}

void hash_subset_of_keys(int l1slot)
{
  int    i, h2, target;
  int    offset = ht.T_offset[l1slot];
  int    si = ht_s[l1slot];
  uint   a2 = ht.a2[l1slot], b2 = ht.b2[l1slot];
  object *obj, *existing;

  for(i = offset; i < offset + si; i++)
  {
    obj = ht.T[i];
    ht.T[i] = NULL;

    h2 = hash(obj->id, a2, b2, (si*(si-1)));

    target = offset + h2;
    existing = ht.T[target];
    ht.T[target] = obj;

    /* We had used the initial part of hashtable T{i} to store subset S{i},
       and now we are hashing its keys across T{i}. So the "target" index
       may contain a not-yet-hashed key of S{i}. */
    if(existing)
    {
      ht.T[i] = existing;
      i--;
    }
  }
}
```

```
/* size of T{i} can be inferred from offsets */
#define SIZE_T(i) (ht.T_offset[i+1] - ht.T_offset[i])

object *lookup(uint k)
{
  int    h1, h2, tsize, index;
  object *obj;

  h1 = hash(k, ht.a1, ht.b1, ht.m);

  tsize = SIZE_T(h1);

  if(tsize == 0)
    return NULL;

  index = ht.T_offset[h1];

  /* no level-2 hashing was done for tsize=1 */
  if(tsize >= 2)
  {
    h2 = hash(k, ht.a2[h1], ht.b2[h1], tsize);
    index = index + h2;
  }

  obj = ht.T[index];

  if(obj == NULL || obj->id != k)
    return NULL;

  return obj;
}
```

■

References

[1] M. L. Fredman, J. Komlós, E. Szemerédi. *Storing a Sparse Table with O(1) Worst Case Access Time*. J. ACM, Vol 31 (3) (1984), 538–544.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*, Third Edition. The MIT Press (2009).

[3] Nitin Verma. *Universal Classes of Hash Functions.* `https://mathsanew.com/articles/universal_classes_hash_functions.pdf` (2021).