Description of hashing implementation in various programming languages

Nikolaj Hass

Fall 2014

Abstract

Abstract goes here

Read in general: http://lwn.net/Articles/612021/

1 Introduction

This paper will explain how different types of the Hash Table data structure is implemented in different programming languages. Additionally, their methods will be compared where comparable and their choice of solution will be discussed.

In total four programming languages will be covered and five Hash Table implementation. One of the languages chosen, C#, has two different implementations of the Hash Table data structure that are both covered in this paper.

The languages chosen can be found below:

- C#
- Perl
- PHP
- Java

This report only has the focus of looking into and explaining how the mentioned programming languages implement the hash table data structure, and will therefore not explain how the different hash table implementations are protected against various issues that may arise when using a specific programming language.

Additionally, this paper will not cover how the different program languages deal with storing, removing and retrieving values in the memory, but simply how these are retrieved, stored and removed in the hash table.

BUT NOW SOMETHING ABOUT HASH TABLES AND HASH FUNCTIONS.

1.1 The Hash function

The purpose of the hash function is to take a value and return an integer that is between 0 and M-1 where M is the size of the hash table. In other words, the hash function distributes the keys inserted and should be optimized for the least collisions possible, as the more key collision a hash table has the longer it takes to find the value associated with the given key.

Additionally, in some implementations a bad hash function can lead to the hash table having to re-initialize or re-hash itself to a larger hash table prematurely.

When developing a hash function one can look to the following guidelines, defined by Bob Jenkins [1], REFERENCE, :

1. The keys are unaligned variable length byte arrays

[1] About Bob Jenkins

2. Sometimes keys are several such arrays.
3. Sometimes a set of independent hash functions are required.
4. Average key lengths range from 8 bytes to 200 bytes.
5. Keys might be character strings, numbers, bit-arrays, or weirder things
6. Table sizes could be anything, including powers of 2.
7. The hash must be faster than the old one
8. The hash must do a good job.

By point 7 is meant that the hash function one develops, should be better than the previously used one, otherwise one is simply wasting time developing a new function.

TAKE THE WORD FROM ALGS, pge 458 or so By point 8 is meant that the hash function distributes the key efficiently so the previously mentioned consequences are avoided as much as possible.

The perfect hash function is only something you can achieve if you know the values beforehand.

1.1.1 Difference between hash code and hash function different title!

Basically, an objects hash code is not the same value as the one we return when we do a hash function in a hash table.

A Hash Code in most programming languages is used to present the hash value, an integer value, of an object. This can be used for comparing two objects to see if they are identical. A hash function is however, in our case, used to determine

However, a hash function, in our case, is something used when using Hash Tables, and therefore

The hash function is an essential part of the Hash Table data structure and picking a good hash function is vital in order to have an efficient hash table, as the hash function distributes the keys . The Hash function is an essential part of the Hash Table data structure and picking a good hash function is vital in order to have a efficient hash table.

The Hash function computes a given key into an index in the range of 0 and the hash table size minus 1, annotated M-1. Page 459 ALGS

Only interesting to look at how strings are hashed. Ints usually just return their value, as a hash code value. Picking a good hash function is essential

1.2 Ways of implementing the HashTable data structure

It is very hard to pick a perfect hash function, as explained in the previous section, and therefore collisions, where two different key values has the same hash value, are unavoidable.

The solution to this problem is called collision resolution and can be implemented in a variety of ways, and some of them are covered in this section.

The Hash Table solutions covered in this section are not the only ones that exist, merely the most common and used ones. Not all of them are used in the programming languages of today, but they are still worth covering as they all serve their own purpose.

1.2.1 Open addressing

Open addressing is a method where all entries are stored in one large array and every time a key is hashed to the same slot as an already inserted key a different slot is found for the new key by probing in a specified sequence through the array until an empty slot is found. This sequence is usually called the probe sequence and it entirely depends on the given implementation. In most implementations the hash table is resized if no empty slot is found.

HOW DOES ONE FIND A VALU E THEN Then in order to find the inserted value the entry

1.2.1.1 Linear probing

In linear probing the probe sequence is defined as a fix interval, usually 1.

1.2.1.2 Quadratic probing

1.2.1.3 Double hashing

1.2.2 Chained hashing

1.2.3 Cuckoo Hashing

EVERYONE USES BUCKETS!

2 Programming languages

In this section five different implementations will be

2.1 Perl

2.1.1 Sources

text1

text text text text text The hash is found here. tt superDuper

2.1.2 HOW WE HASH

This function takes three variables each called: hash, str and len. Where the hash variable is the variable the caller of the function would like to have equal the found hash code, the str variable is the string to be hashed and the len variable is the length of the string variable to be hashed.

The function performs a few initializations in order not to manipulate anything outside the scope of this function.

Additionally, the variable hash_PeRlHaSH is initialised to equal the given perl hash seed, which per default is 0. It is possible for the user of the hash table to change this default value, if a different hash value calculation is wanted. This value is used to hold the temporary hash value of the given string, during calculations.

The function iterates over every character in the given string and for every iteration the value of the character the iteration has reached, is added to the hash value.

Furthermore, the current hash value, bit-wise shifted to the left 10 times, is added to the current hash value. Then bitwise XOR is used on the newly calculated hash value and the newly calculated hash value - shifted 6 times to the right.

After the iteration over the given string, the hash value, bit wise shifted to the left 3 times, is added to the hash value. Bitwise XOR is then used on the calculated hash value and the calculated hash value - bitwise shifted to the right 11 times.

Finally, the hash value is calculated by adding the hash value to the hash value bitwise shifted to the left 15 times.

```
#define PERL_HASH(hash,str,len) \
STMT_START { \
   register const char *s_PeRlHaSh_tmp = str; \
   register const unsigned char *s_PeRlHaSh = (const unsigned char
       *)s_PeRlHaSh_tmp; \
   register I32 i_PeRlHaSh = len; \
   register U32 hash_PeRlHaSh = PERL_HASH_SEED; \
   while (i_PeRlHaSh--) { \
      hash_PeRlHaSh += *s_PeRlHaSh++; \
      hash_PeRlHaSh += (hash_PeRlHaSh << 10); \
      hash_PeRlHaSh ^= (hash_PeRlHaSh >> 6); \
   } \
   hash_PeRlHaSh += (hash_PeRlHaSh << 3); \
   hash_PeRlHaSh ^= (hash_PeRlHaSh >> 11); \
   (hash) = (hash_PeRlHaSh + (hash_PeRlHaSh << 15)); \
} STMT_END
```

2.1.3 Explanation

In perl as Hash is a hash table (unlike most other languages). In Perl a hash is an array of buckets, where a bucket is an array. This is still chain hashing, but differs from other languages that use Linked Lists.

2.1.3.1 Store When Perl stores, it first hashes the given key. Then finds the bucket we have to look in's index by taking the hash modulus the size of the hash table. Then searches through the found bucket, and if an entry that has the same hash value and key the one that we are trying to insert that node is returned. If no entry is found, nothing is returned. If an entry is found, the entry is updated with the new value and the method terminates. If no entry is found, it means we have to insert a new entry. We insert a new entry by looking at the selected bucket (list) and insert a new node at the beginning of the list. Meaning the new node points at the last inserted node in this bucket. Then we update the hash table to point to the new element at the given hash value slot. We incremenet the amount of keys by one. Everything is linked lists, so the hash table is just an array of linked lists, meaning it only points to one element, but we know it might point to more, so we naturally check if there is more elements when inserting or getting elements.

After the insert has been executed we tcheck if we need to make the hash table double sized. We make the hash table double its current size if it is full and it is NOT the head_node. (evt. ask).

2.1.3.2 is full Checks if the amount of keys in the hash table exceeds or is equal to its size.

2.1.3.3 Double size This method takes the hash table's current size and doubles it. For every slot in the new resized hash table, Perl runs through every old bucket, checking if every entry in the respective buckets still fit in the same bucket. If an entry no longer fits in its current bucket, it is moved to the right bucket. Perl locates entries that are not in the right place by joining the entry's hash with the size of the hash table -1. If the entry has to be moved the To pointer of the entry is updated. If the entry has an entry "behind" it, this one gets updated, if it does not, the entry "infront" of this entry gets set to be the "initial" entry. Then the entry's from pointer needs to be resolved. THe entry's next is sat to the initial element of the correct bucket and the entry is inserted infront of the other entries in the bucket.

If the entry already resides in the correct bucket, the loop just moves on, without tampering with the entry.

When Perl stores a value, it looks up the value. If the value already exists (meaning the look-up succeeded), the value is replaced with the new value. Otherwise, if the value does not exist, the bucket that has been hashed to is updated

2.1.3.4 FETCH

Fetch takes a key. Hashes the key. Finds the size of the hashtable Finds the bucket index by joining the hash and the size -1. Searches the bucket. Returns a value if one exists.

2.1.3.5 search bucket

Takes a bucket index, key and hash. Searches through the specified bucket for a node that has a key and hash equal to the what was specified.

2.1.3.6 Delete

The delete function takes a key. It hashes the key, to find the hash value generated by the key. Then the method finds the correct bucket slot by joining the hash with the size of the table -1. Then the method runs through all the items in the bucket until it finds the entry that has the same hash and key as the previously found hash and given key. If no entry is found, nothing is deleted, however if something is found, then if there is an entry in the list, before the to-be-deleted entry, then that entry's pointer to the next entry is updated from pointing at the to-be-deleted entry to pointing at the entry the to-be-deleted entry points at. If no previous entry exists, meaning the to-be-deleted element is the first element in the list, the first element in the list is sat to being the entry right after the to-be-deleted entry.

http://perldoc.perl.org/functions/shift.html

2.2 Java

2.2.1 SOURCES

text text

2.2.2 Explanation

2.2.2.1 The following explanation is derived from the JAVA source code

2.2.2.2 Calculating an index in JAVA In JAVA a hash table index is calculated by hashing the given key, see hash code, using bitwise AND on this value, with the value "0x7FFFFFFF" and then getting the value of the found value modulus the size of the hash table.

2.2.2.3 The hash code This method first sets an integer h equal to the variable hash. The variable hash is either 0 or a previously calculated hash. If the hash value is equal to zero and the length of the string value is larger than zero, the function calculates the hash and returns it.

The calculation is done by iterating through every character in the string, setting the hash value to 31 times the current hash value, plus the value of the character in the current interation.

```
public int hashCode() {
  int h = hash;
  if (h == 0 && value.length > 0) {
     char val[] = value;

     for (int i = 0; i < value.length; i++) {
        h = 31 * h + val[i];
     }
     hash = h;
  }
  return h;
}
```

2.2.2.4 Put When a new entry is inserted, it is first checked if the value of the entry is null, if it is, an exception is thrown.

Then the put methods checks if there is an entry with the same key already found in the hash table This is done by hashing the key, see REF CALC JAVA. The index that is found doing this contains a list of entries. The entries are iterated through, checking if the given entry's hashcode and key is equal to the newly inserted key and hashcode. If any of the entries matches this, the entry is replaced with the newly inserted entry and returned.

If no entries are matched, it is checked if the amount of entries in the hash table exceeds the calculated threshold. The threshold is calculated by taking the size of the hash table times the load factor. The load factor is found by dividing the number of entries with the number of buckets, which in JAVA defaults to 0.75. This threshold is how JAVA knows that the hash table has grown too large to be efficient.

If the amount of hash table exceeds the calculated threshold, the entire hash table is rehashed. See the description of the rehash function.

Then the entry is inserted at the previously found index and the count of the amount of elements in the hash table is incremented. New key value pair is entered at the correct index. The entry, or bucket, is represented by a linked list. This is done by taking the old entry, and creating a new entry with the new information, and setting the old entry as the "next" element.

2.2.2.5 The Rehash Function Java's rehash function creates an array of entries that is double the size + 1 of the old entry array. The new threshold gets calculated with the new capacity and every entry in the old entry array gets inserted into the new entry array. This is done by running through every entry, rehashing the key of every key-value pair, thereby finding the pair's new slot in the new entry array.

2.2.2.6 Remove The JAVA hash table remove method takes a key of the element that the caller of the method wants to remove. The index of where the

entry might be located is found by CALCULATING THE INDEX, then the found list of entries is iterated through. If an entry is found that matches the hashcode and value of the key, the found list is updated to effectively remove the found entry.

This is done by updating the entry that was just before the found entry, to point at the entry that was just after the found entry and the entry just after the found entry is updated to instead of pointing at the found entry as its previous entry, it now points to the found entry's previous entry.

Finally, the found entry is returned.

2.2.2.7 Get

The get method searches for an entry that matches the given key. This is done by calculating the CALCULATING THE INDEX of the key and searching through the found list of entries. If an entry that matches the given key's hash and value the value of the entry is returned. Otherwise null is returned.

2.3 C#

Hashtable is an old implementation, Dictionary is the new implementation. Interesting to look at, because it is an old implementation, that is different from so many other

2.3.1 Sources

text text TT text

2.4 How C# hashes Strings.

```
[ReliabilityContract(Consistency.WillNotCorruptState, Cer.MayFail),
    SecuritySafeCritical, __DynamicallyInvokable]
public override unsafe int GetHashCode()
{
   if (HashHelpers.s_UseRandomizedStringHashing)
   {
      return InternalMarvin32HashString(this, this.Length, 0L);
   }
   fixed (char* str = ((char*) this))
   {
      char* chPtr = str;
      int num = 0x15051505;
      int num2 = num;
      int* numPtr = (int*) chPtr;
      int length = this.Length;
      while (length > 2)
      {
         num = (((num << 5) + num) + (num >> 0x1b)) ^ numPtr[0];
         num2 = (((num2 << 5) + num2) + (num2 >> 0x1b)) ^ numPtr[1];
```

```
            numPtr += 2;
            length -= 4;
        }
        if (length > 0)
        {
            num = (((num << 5) + num) + (num >> 0x1b)) ^ numPtr[0];
        }
        return (num + (num2 * 0x5d588b65));
    }
}
```

2.4.1 Explaniation, double hashing

USES DOUBLE HASHING

2.4.1.1 Add Takes a key and a value, and calls the method insert.

2.4.1.2 Insert Firstly checks if the key is null, if it is an exception is thrown. Then It is checked if the size of the hash table is larger than the calculated loadsize, if it is, the hash table is expanded (See Expand). When we have passed through these checks we get the hashcode for the given key, in addition to the seed and the incrementer. We also keep track of the "empty slot number", which is used to save the first empty bucket that previously has had an entry and has had a collission. The "empty slot number" is set if a bucket is found that matches the key and has had zero collissions. If we find a bucket that has never had an entry or has contained an entry in the past but never had any collission. The empty slot number is then used to insert the entry in the bucket previously found, which already has a collission.

Collissions are recorded through the hash code's sign bit, which will be 1 if a collission has occurred and otherwise will be 0, so C# just checks if the hashcode is positive (0) or negative (1). If we find a bucket that has never had an entry or previously contained an entry and never experienced a collision, we insert the key and value into this bucket. However, if we have found an "empty slot", through the "empty slot number" not being negative, we insert the key and value into the "empty slot" bucket. The amount of keys and the Hash table version gets incremented.

If we can not find the above described bucket, we check if the current bucket is in use or if it is available, but has had a collision bit set and we have already found an available bucket. In other words, if the current bucket is in use and the entry shares the same key as our new entry, we update the bucket's value to equal our new entry's value.

If this is not the case, that we find an available bucket, then we must conclude that the current bucket is full. If we have found a available slot previously, we insert our new entry there. If we have not found a slot previously, we update the current bucket's collision bit to 1, meaning there has been a collision and we

increment the bucket number with the incrementer value mod the hash table length.

If no collision bits are set in the entire table, and we have found an empty slot we insert our entry in the bucket located at the empty slot.

2.4.1.3 Expand Expands the Hash table by doubling the current size and finding the nearest prime number that is larger than the found number. Then the rehash method is called with this new size.

2.4.1.4 Rehash Resets the occupancy and creates a new array of buckets with the new size, given through parameters.

2.4.1.5 InitHash INitHAsh takes a key, hashsize, out uint and another out uint. The hashcode is calculated getting the hash of the key (GetHash) and joining it with 0x7FFFFFFF. Then the seed, the first out uint, is set to the hash code. Afterwards the second out uint is calculated, being the increment. This is calculated by saying 1 + the seed shifted 5 times to the right + 1 mod the hashsize -1. Then the hashcode is returned.

2.4.1.6 GetHash

2.4.1.7 GetPrime GetPrime returns the nearest prime value that is larger than the argument given. The Prime method first runs through an array of pre-established primes, if none of those primes are higher than the argument specified, the prime is calculated.

2.4.1.8 Get the value using key The read function in C# implementation of the hash table has over all four steps. First the hash is calculated based on the key given by the user and the slot number is found by taking the hash code mod the size of the hash table. Then it is checked if the key found in this bucket equals the key initially given by the user. If it does the selected bucket's value is returned. If they keys do not match, the bucket number is incremented by the specified incrementer, and this is done until there are no more entires or the hash code is less than 0. If no value is found, null is returned.

2.4.1.9 DEFINITIONS Occupancy is the total number of collsion bits set in the hashtable. count is the total number of entries.

Loadfactor is 0.72f times whatever Loadfactor set by the user. When you just create a new hashtable default loadfactor will be 0.72f.

LoadSize is an int calculated as loadFactor times hashsize

The hashsize is found through dividing the capacity with the loadFactor. If the hashsize is below 11 it is set to be 11. This is to avoid a too small hashtable. Loadsize must be less than hashsize, otherwise an exception is thrown. It is a design choice to never fill out the hash table 100 %. (In contrast with many others)

2.4.2 Dictionary implementation C#

A dictionary is another type of Hash table implemented in C#. Note that this implementation is generic. The dictionary has two ways of adding elements, one which does not insert duplicate keys, and one that does.

2.4.2.1 Entry This hash table consists of an array of entries. An entry consists of a hashcode, an index of the next entry, which is -1 if it is the last element in the list. The entry struct also has a key and a value. This datastructure contains an array of ints and an array of entries.

2.4.2.2 Initialize Initialize initalizes a new hash table by first finding the new size of the hash table. THe new size of the hash table is calculated by taking the given capacity and finding the prime number that is closest to that capacity while still being a larger integer. Then our array of ints, which represents our buckets is initialised, our entries array is initialized and our freeList is sat to -1.

The initialize is only called once in a dictionary's lifetime, and that is when the array of ints representing buckets is null, which it will be at the very first insert of an entry.

2.4.2.3 Add Calls insert, but does take duplicate keys. .

2.4.2.4 Get method The get method tries to find the entry key, using the FindEntry method. If a valid key is found, which is a key equal or larger than 0, the entry is returned. If no valid key is found the default value is returned, which is either null or 0, depending on datatype.

2.4.2.5 FindEntry If a key is not null and the hash table is initialized, this method hashes the key and looks through the found bucket to locate an entry that matches the specified key. If such an entry is found the location of the entry in the entries array. If nothing is found -1 is returned.

2.4.2.6 Insert We check if the key is null, if it is we throw an exception. If the hashtable is null, we initialize it. We get the hashcode of the key and join it with 0x7FFFFFFF. Then we find the target bucket entry by taking the hashCode mod the size of the hash table. We then run through the found bucket's entries. If the method is called in the Add method and we find that the key already exists in the hash table, we throw an exception. In all other cases we update the entries value, to the given value.

If no entry with the same key and hashcode as the key and hashcode given is found, it means we have a collision. We then check if there is space for more elements in our array of entries.

If there is, we store the new entry's index number using our freeList variable, and update our freeList variable to point at the next available entry. Then our counter, that keeps tracking of how much space there is for more elements in

our array of entries, gets decremented. If our counter says that there is no more space in our array of entries, we check if the amount of entries is equal to the size of the hash table. If that is the case, we resize the hash table, and find our target bucket again, by taking the hashcode mod the now new hash table size. Afterwards we store the count as the index for when the just inserted entry is supposed to be put and increment the count by one.

Then we insert the entry at the found index, in our array of entries. The "next" value is set to the number of the bucket we are inserting in, found using the buckets array. Additionally, the version count is incremented by 1.

THE BUCKETS ARRAY KEEPS TRACK OF WHERE THE NEXT ELEMENT IN THE BUCKET IS IN THE ENTRIES ARRAY. SO THE ENTRIES ENTRY IS A NATURAL WAY OF RESIZING BUCKETS.

2.4.2.7 Resize The resize method firstly finds a prime number using the ExpandPrime method, and then calls another Resize method. This resize method has the ability to force new hash code, but per default does not.

The resize method checks if the new size given is larger or equal to the amount of entries already in the hash table. If the above holds, a new array of buckets is initialized with the new size and each new entry is set to -1. Then a new entries array is created being the new size, and all the old entries are copied into this array.

If it is demanded by the caller of the function that new hash codes should be forced, the method gives all the entries a new hash code by running through the entire new entries array.

Then every entry in the new entries array that has a hashcode is run through and given a their appropriate bucket. The new bucket is found by taking the entries' hashcode mod the new size.

2.4.2.8 ExpandPrime Expand prime calculates a new size by first doubling the old size of the hash table. To avoid capacity overflow, the newly calculated value is checked against what the .NET framework defines as the maximum prime array length, which is 0x7FEFFFFD in HEX and 2146435069 in decimal. If the calculated value is larger than the maximum prime array length, then the maximum prime array length is returned and thereby the method terminates.

If the value calculated is not larger than the maximum prime array length, then the GetPrime method REF HERE is used to find the closest prime number to the calculated value that is still larger than the calculated value

2.4.2.9 title

2.4.2.10 Variables Version: Version number of the dictionary FreeList: Holds the amount of elements currently in the hash table. FreeCount: Holds the amount of free slots available in the hash table. Entries, an entry array.

2.5 PHP

2.5.1 Sources

text text

2.5.2 Explanation

The hash table implementation of PHP uses two structs, the bucket struct and hash table struct.

2.5.2.1 The bucket struct

The bucket struct consists of The bucket struct consists of a hashcode variable, a key length variable, a key variable. Additionally, the bucket struct consists of pointers to the nextada da

2.5.2.2 Hash table struct

The hash table struct consists of a variety of variables, the ones of importance to us is the following variables:

- Table size (Integer)
- Table mask (Integer)
- Number of elements (Integer)
- Next free element (Long)
- Pointer to the first element in the hash table (Bucket)
- Pointer to the last element in the hash table (Bucket)
- List of pointers to buckets. (Bucket)

2.5.2.3 Insert

IF FLAG HASH NEXT INSERT, WORK AROUND THIS?

The insert method takes

The insert function in PHP first finds the index of which array of buckets

2.5.2.4 The hash table variables

The hash table holds various general variables that is needed for the hash table to execute.

The hash table in PHP has a table size variable,

2.5.2.5 arg1

2.5.2.6 The bucket

A bucket in PHP represents an entry in the hash table. It consists of a hash value, a pointer to the next element

There is a few things I do not know about PHP as of yet.

Checks if the hash table is consistent. Is a PHP thing. Check if key length is creater than 0, if it isn't returns failure.

Hashes the key, findes the bucket index by joining & h and TableMask. The table mask is the size of the hash table - 1. Formed in bits.

Finds the bucket, in the hash table's array of buckets, using the found bucket index. Then PHP runs through every key in the found bucket. PHP can return a FAILURE, meaning nothing has been inserted, on several occasions. If there is found a key that is equal to the key specified to be inserted. If There is a key with the same hash, length and are not equal (using memcmp), the insert fails. Additionally, there is some weird check if flag and HASH_ADD, and flag is not defined.

if the hash tables pDestructor is true, then the value is destructed. Then Update Data method is called, meaning the data that needed to be updated has been found and updated. Method stops. However, if that is never found. Checks if the key value is interned. ttt THE WHILE DOES NOT HAVE TO RUN, AS WE CAN HAVE CREATED A HASH THAT DOES NOT EXIST AS INDEX. If it is, we try and allocate memory to a new bucket, and add the key to that bucket. If we can not allocate memory, it fails. else if it is not interned we create a new bucket, with the memory of a bucket + key length. If we can̂everything fails. If we can create this new bucket, and do some memory stuff.

Then we set the newly created bucket's keylength. Sets INIT_DATA, which I am not quite sure about. Allocates memory for the data, if it can not it fails. We se tthe hash. Then we call Connect_TO_BUCKET_DLLIST, In this method we set the next element of our current bucket element, to be bucket placed at the found "hash" index in the hash table. The last element we set to null, but if we are able to set the next element of the current bucket element, then we set the last element of the next element to be the current element.

Then we check if pDest exists, we set the pointer of pDest to be the current bucket element's pdata.

When all of this is done, we update the global DLList, which fixes the order of the array elements. (More info about this to come). It is a simple last element linked list update. We set the the current bucket element's last element to be the hash tables last element. THen we set the hash table's last element to be the current vbucket element. THen doa lot of checks.

Then we set the bucket placed at the found "hash index" in the hash table to our newly created bucket. Then we increment the number of elements(buckets) in the hash table by one. We then check if the hash table is too big, meaning if the number of elements is larger than the specified Hash Table size, we resize the hashtable to be as double as big.

WE can do all these things directly to the bucket, as it is a linked list, and we are on the first element!

2.5.2.7 HOW DOES PHP HASH Is really good when using Strings, etc. Some stuff about that. PHP's hash function looks like the following:

tt tttt tttt

This hash function is internal, and therefore not in the the standard php library. Meaning it can not be used normally.

The PHP hash function takes a pointer to a char, in other languages this can defined as a string value, and a key length. The function initializes a long value to 5381, this long we are going to modify when hashing any given string. The function creates a for loop that decrements the key length by 8 every iteration and runs until the key length is less or equal to 8. For every iteration the hash is modified 8 times, every modification being overwriting the previously defined hash value with a newly calculated one.

This calculation is done by bit-wise shifting the current hash value 5 times to the left and adding this value to the current hash value. Then this calculated value is added to the value of the char pointer given previously. Finally, the char pointer is incremented by 1.

When the for loop seizes to run, we have a key length of 8 or less. This length is then put into a switch statement that has fall through cases, where each case executes the same calculation as in the for loop, until the 0 case is reached, where the switch statement is breaked.

After the switch statement the function terminates, as the hash value has been calculated.

```
static inline ulong zend_inline_hash_func(const char *arKey, uint
    nKeyLength){
      register ulong hash = 5381;

      /* variant with the hash unrolled eight times */


     for (; nKeyLength >= 8; nKeyLength -= 8) {
          hash = ((hash << 5) + hash) + *arKey++;
          hash = ((hash << 5) + hash) + *arKey++;
          hash = ((hash << 5) + hash) + *arKey++;
          hash = ((hash << 5) + hash) + *arKey++;
          hash = ((hash << 5) + hash) + *arKey++;
          hash = ((hash << 5) + hash) + *arKey++;
          hash = ((hash << 5) + hash) + *arKey++;
          hash = ((hash << 5) + hash) + *arKey++;
      }
      switch (nKeyLength) {
            case 7: hash = ((hash << 5) + hash) + *arKey++; /*
                fallthrough... */
            case 6: hash = ((hash << 5) + hash) + *arKey++; /*
                fallthrough... */
            case 5: hash = ((hash << 5) + hash) + *arKey++; /*
                fallthrough... */
            case 4: hash = ((hash << 5) + hash) + *arKey++; /*
```

```
            fallthrough... */
        case 3: hash = ((hash << 5) + hash) + *arKey++; /*
            fallthrough... */
        case 2: hash = ((hash << 5) + hash) + *arKey++; /*
            fallthrough... */
        case 1: hash = ((hash << 5) + hash) + *arKey++; break;
        case 0: break;
}
```

3 Discussion

3.1 Programming languages hash codes

Every programming language in this paper has a different way of implementing the hash code function, as demonstrated throughout this paper. Therefore, the question arises: *Do the different programming languages hash the same value differently?*

To answer this question tests, using the word "table", has been performed to see if the values generated by the different hash code functions actually differ.

The C# and Java hash code has been tested using their respective standard library, as the hash code used in their hash table implementations is both part of their standard library string implementation.

However, using the standard library for testing the hash code implementation of PHP and Perl has not been possible, as the hash code both languages use for their hash table implementation is an internal one, making it unavailable outside of the implementation's scope.

The solution to this problem has been to implement the two hash code functions in a test environment based on the code shown and explained earlier in this paper. The test implementations of the hash codes uses different data structures than their originals, however any data loss should be minimal and has therefore been disregarded as an issue due to the scope of this test.

The actual test implementations can be found in the appendix section "Hash code function test code" found at page 18.

Programming language	**String Value**	**Hash Value**
C#	"table"	-798614011
Java	"table"	110115790
PHP	"table"	275315341
Perl	"table"	-2062238372

Figure 1: Hashcode test results

Programming language	Included
Java	Yes
PHP	Yes
Perl	Yes
C#	Yes
C++	No
C	No

Not part of the standard library in: c

Gonna be in the future: C++

Make a box of languages where implementation of hash tables are library dependent. FLERE KILDER MED HASH FUNCTION Hash samme string med forskellige inputs, se output! Bokse er nice! NO TESTS WILL BE PERFORMED WE WILL NOT LOOK INTO LANGUAGE SPECIFIC THINGS, SUCH AS OPTIMIZING FOR THREADING.

4 Conclusion

5 Tmp global sourcelist

Stackoverflow. Cprogramming Lix ttt

Appendices

A Hash code function test code

This section contains the test code used to conduct the test discussed in the section "Programming languages hash codes" on page 16.

A.1 The C# test code

The C# test code is based on the .NET 4.5.1 standard library.

A.2 The Java test code

The Java test code is based on the Java 8 standard library.

A.3 The Perl test code

The Perl test code is implemented in the C programming language using GCC-4.8.1 based on the hash code implementation found in hv.c in the Perl standard library. (VERSION NUMBER NEEDED).

```
#include <stdio.h>

long perlHash(char arKey[], long len){

   long hash = 0;
   register const char *s_PeRlHaSh_tmp = arKey;
   register const unsigned char *s_PeRlHaSh = (const unsigned char
       *)s_PeRlHaSh_tmp;
   register long i_PeRlHaSh = len;
   register long hash_PeRlHaSh = 0;

   while (i_PeRlHaSh--) {
      hash_PeRlHaSh += *s_PeRlHaSh++;
      hash_PeRlHaSh += (hash_PeRlHaSh << 10);
      hash_PeRlHaSh ^= (hash_PeRlHaSh >> 6);
   }

   hash_PeRlHaSh += (hash_PeRlHaSh << 3);
   hash_PeRlHaSh ^= (hash_PeRlHaSh >> 11);
   (hash) = (hash_PeRlHaSh + (hash_PeRlHaSh << 15));

   return hash;
}


int main(void) {
   char label[] = "table";
```

```
    int length = 5;

    int test = perlHash(label, length);
    printf("Perl hash value: %d", test);
    return 0;
}
```

A.4 The PHP test code

The PHP test code is implemented in the C programming language using GCC-4.8.1. The implementation is based on the hash code implementation found in PHP WHATEVER in the PHP standard library version XXX.

```
#include <stdio.h>

long php_hash(const char *arKey, long nKeyLength){
    register long hash = 5381;

    for (; nKeyLength >= 8; nKeyLength -= 8) {
        hash = ((hash << 5) + hash) + *arKey++;
        hash = ((hash << 5) + hash) + *arKey++;
        hash = ((hash << 5) + hash) + *arKey++;
        hash = ((hash << 5) + hash) + *arKey++;
        hash = ((hash << 5) + hash) + *arKey++;
        hash = ((hash << 5) + hash) + *arKey++;
        hash = ((hash << 5) + hash) + *arKey++;
        hash = ((hash << 5) + hash) + *arKey++;
    }

    switch (nKeyLength) {
        case 7: hash = ((hash << 5) + hash) + *arKey++; /*
            fallthrough... */
        case 6: hash = ((hash << 5) + hash) + *arKey++; /*
            fallthrough... */
        case 5: hash = ((hash << 5) + hash) + *arKey++; /*
            fallthrough... */
        case 4: hash = ((hash << 5) + hash) + *arKey++; /*
            fallthrough... */
        case 3: hash = ((hash << 5) + hash) + *arKey++; /*
            fallthrough... */
        case 2: hash = ((hash << 5) + hash) + *arKey++; /*
            fallthrough... */
        case 1: hash = ((hash << 5) + hash) + *arKey++; break;
        case 0: break;
    }

    return hash;
}
```

```
int main(void) {
   char label[] = "table";
   char *label2;
   label2 = label;
   int length = 5;

   int test = php_hash(label2, length);
   printf("PHP hash value: %d", test);
   return 0;
}
```