Copenhagen STL — hash map

Christian Boesgaard (pink@diku.dk)
and
Jacob Chr. Poulsen (ja7@diku.dk)
Department of Computer Science, University of Copenhagen,
Universitetsparken 1, DK-2100 Copenhagen East, Denmark

CPH STL Report 2001-5, February 2001

Abstract

We describe the design and prototyping of the hash map in the Copenhagen STL. We explain shortly what hash tables are and discuss the implementation in the SGI STL. We then propose a design optimized for modern memory hierachies and complex keytypes, like strings. The design is implemented and tested. The prototype is not faster than the SGI implementation. We discuss this and conclude with a brief listing of open questions and ideas for a future hash map implementation in the Copenhagen STL.

1 Introduction

This work is a part of the Copenhagen STL project [cphstl]. We are doing the work on the hash table, but the proposed optimizations should be generally useful for hash based containers (i.e. hash set, hash multimap etc.). The Copenhagen STL is a library meant for real work and the hash map should be designed with this in mind – more specifically we are working towards optimizations of tables indexed by complex types like strings.

Actually hash containers is not yet a part of the C++ standard [C++std]. Because the hash containers are not standardized we choose to let our interface design be inspired by the map from the C++ standard and the hash containers from SGI [SGI].

Our primary goal was to propose and try new designs to speed up hash tables, secondary we wanted to make a full blown and *compliant* implementation. The primary goal was aquired, we ended up with a proof of concept implementation of our ideas. The secondary was not quite reached, the implementation is not finished.

The files mentioned in the article can be obtained electronically from the authors.

2 Hash maps

Hash maps are a much used datastructure in all kinds of computer programs, for instance in implementation of symbol tables.

The heuristic concept of hashing was introduced by Dumey in 1956 in [Dumey]. It was introduced as a solution to the dictionary problem (for use with symbol tables).

The dictionary problem

We are given a sequence of INSERT(k, x), DELETE(k) and LOOKUP(k) operations (each operation must be completed before the next is started) on an initially empty set S. The elements in S can be thought of as pairs (k, x). The key k is an arbitrary bit string, x is an arbitrary piece of data (these symbols will be used in the rest of the article).

INSERT(k, x) inserts the data x in S in a place denoted by k. DELETE(k) removes the information indexed by k (k is also removed). LOOKUP(k) returns x indexed by k (if it exists). It is trivially seen that both the INSERT and DELETE operation needs to do a LOOKUP, this operation is the important one.

The goal is to perform the operations as fast and space efficient as possible.

In the general problem we can assume no knowledge about keys or the information associated with keys. In practice we often know something about both keys and data.

A very simple dictionary could store keys and data in a list. The list could be searched linearly. This could be done in space $O(N)$ for N key, data pairs, searching could be done in $O(N)$ time. We can do better if we use a binary tree-structure and keep the set sorted, search time could be reduced to $O(lgN)$.

The hash table is an alternative strategy with worstcase searches in $O(N)$ but with best case and expected case searches in $O(1)$. The idea is to calculate the position of x in a random access array from k. If all the keys to be used are known in advance, it is possible to choose a function which maps the n keys to unique places in an array of size n (this is called perfect hashing). Unfortunately such functions are rare [Knuth, section 6.4] and the data sets are generally not known in advance. We can therefore not guarantee $O(1)$ operations because the possibility of collisions, i.e. n keys could map to the same position. This also means we have to handle collisions. A simple but effective way to handle collisions is to let each position in the array, which we will call table or jump table, hold a structure (we will denote such structures as buckets) with (k, x)-pairs. The right pair is then searched for (this is of course a dictionary problem itself). Most current designs are based on this simple approach and in the rest of this article we will only discuss this kind of designs.

We should note that for keys of arbitrary length the hash function is likely to be $O(l_k)$, where l_k denotes the length of the key. The worst case is thus really $O(l_k) + O(N)$. This is often ignored in practice.

The design of hash tables can be divided into the design of the hash function and the storage layout. The last includes the problem of resizing the table in the case of growing hash tables.

We need a function which can map an arbitrary bit string, the key k to an index in the jump table. In practice this is done by a hash function $h(k)$, that can map arbitrary bitstrings (called pre-images) into fixed size strings (called images) and some reduction function often just modulo the jump table size.

We want to distribute keys evenly in the buckets, this implies the hash function should map the keys evenly in the set of possible images. If the keys

are random it is not hard to make the hash function behave randomly (we can just use the identity function or a simple variant, a variant is only needed for the cases where the key length is greater than the image). It is a lot harder for non-random keys.

3 The SGI hash table

Most free STL implementations available are based on work by Hewlett-Packard Company and Silicon Graphics Computer Systems. In the rest of the article we will simply call this *the SGI implementation* or just something with a *SGI* in it.

The interface implemented is partly defined by the map in [C++std]. It is described in [Stroustrup, page 480-]. The interface supports the functions from the dictionary and some additional functionality (like iterators). We will only look at the basic functions because the others are not relevant in this study.

The implementation is simple. It is based on a jump table the size of a prime. A prime is chosen because this will maximize the bits used from the image. The worst case is to use a 2^n size, this will reduce the image to n-bits, on the other hand this can be done very fast with the shift operation. The buckets are simple linked lists with key-value pairs. The list nodes are allocated independently of the bucket they belong to. The linked lists are searched by comparing keys. Resizing is done by building a new (and bigger) table and moving the values into this.

The hash function is either just the identity function (for simple types) or a very simple deterministic hash function.

The implementation is a little unstable when used on strings (it only supports `char*`-strings by default). We have changed the implementation a little so `char*` is treated correctly and also support for C++ strings was added.

This hash table implementation is very fast and usable. For random strings is it possible to place 3.9 million random strings in a hash table and make find-operations (with functionality like LOOKUP) on all keys in less than 2.5 seconds (the time is for the find-operations only). This can be explained by the simple design which has eased hand tuning and is easy for a compiler to optimize. The use of simple lists also known as *chaining* is showed to be fast in [Jyrki] (other designs are also discussed in that paper).

The SGI design is shown on figure **??**. The arrows denote pointers. The blocks (rectangles) represent independent memory blocks. The figure shows a complex key type where the representation is accesed through a pointer.

4 Design proposal

Our idea is to pay attention to the random memory accesses and the effects this will have on cache misses. Let us take a look at the LOOKUP in the SGI hash table with this in mind. As mentioned ealier we will only deal with complex key-types.

A LOOKUP consists of the following:

Find the right bucket this is an operation done by hashing, which is $O(l_n)$. The image is reduced by modulo the jump table size which is $O(1)$. We

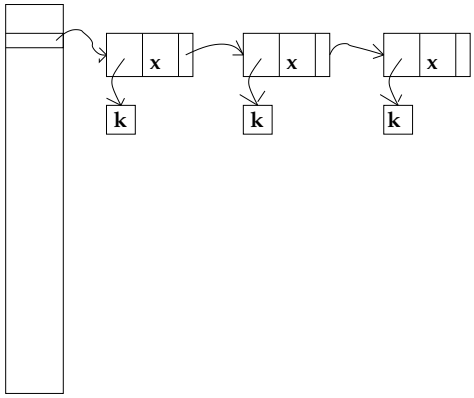

Figure 1: The SGI jump table and bucket.

can expect a cache miss when accessing the key representation

Search the bucket this is an $O(n)$ operation, where n denotes the number of elements in the bucket ($n \leq N$). We can expect a cache miss to get to the bucket (which us the first element in the list). Then we have to compare the key with the one in the first element, which could lead to another miss. If the key did not match we eventually have to proceed with the list. This might as well give us two misses. If the key is present in the list we can on average expect $2n/2$ misses (the key is expected to be found after searching half the list). If the key is not present we can expect $2n$ misses on average.

Because the keys and list nodes are placed at random in memory, some of the cache misses are expected in practice. Even for short lists the number of possible misses are very high related to the computations needed to compare keys.

We will propose two ideas to lower the number of cache misses.

We can allocate the nodes in chunks. This is a trick which we have used earlier to speed up processing of lists. This would bring the cache misses associated with the node access down by a factor determined by the chunk size (in elements). This will increase memory use because we would expect the chunks to be filled only partly. For small (k, x)-pairs this should not be significant, because the overhead used in linked lists (the next pointer) is saved. For big values this strategy could be very expensive in memory.

The other idea is to save the hash image together with the (k, x)-pair. The comparisons can then be reduced to a comparison of images and only a direct comparison of keys when the images match. Assuming only a few of the images will match an image we can reduce actual comparisons (refer to the test of hash functions on page **??**). We can expect the cache misses for comparisons to be one for lists where the key is present and zero for lists where the key is not present.

Overall this could reduce cache misses from $2n/2 + 1$ misses to $(n/s_c)/2 + 2$ misses (s_c is the chunk size in elements) for lists where the key is present, and from $2n + 1$ misses to $(n/s_c) + 1$ misses for lists where the key is not present.

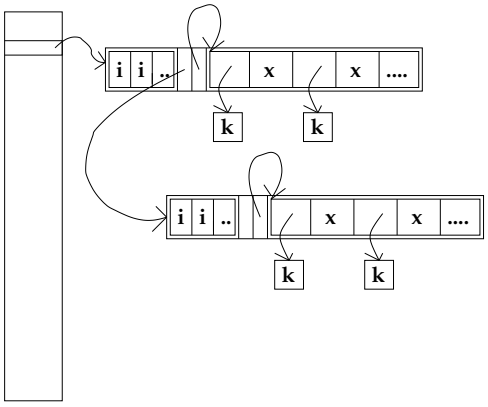

Figure 2: The proposed design, showing two chunks.

It is wellknown that cache misses are expensive (in time) compared to the price of instructions, and we can therefore expect the alternative design to cut down execution time.

The design is shown on figure **??**. The arrows denote pointers. The blocks (rectangles) represent independent memory blocks. The `i`'s denotes the saved images. The figure shows a complex key type where the representation is accesed through a pointer.

5 Implementation

The implementation consists of four parts:

`stl_hash_map.h` – the hash_table class.

`stl_hash_functions.h` – the hash functions.

`fast_bucket.h` – the bucket class used as buckets. A fast bucket is a simple linked list of chunks. All chunks are always filled except the first, which can contain $1 - sizechunk$ elements. We insert elements last in the first chunk. This means we only need a reference to the first chunk. When we remove elements, we move the last element in the first chunk into the free slot. This is done to ease maintaining of the list. On the downside it is extra work when elements are removed from the hash table.

`chunk.h` – the memory chunks used in the fast_bucket. The actual layout is shown on figure **??**, page **??**. The internal pointer is added to ease access to the (k, x)-pairs.

See the corresponding files in Appendix B, page **??**.

6 Benchmarks

We wanted to use a simple test strategy. The goal was to compare our implementaion with the SGI implementation. We divided the tests after the operation

they tested:

insert – in this test we fill an empty hash table with unique elements. This test was done using the `insert()` member function. This corresponds to the INSERT operation.

lookup – in this test we do a lookup for each element in the table following the order they were inserted in. This test was done using the `find()` member function. This corresponds to the LOOKUP operation.

erase – in this test we erase all the elements following the order they were inserted in. This test was done using the `erase()` member function. This corresponds to the DELETE operation.

In all tests the data was small simple types like `int`.

The dictionary test

We first tested the implementations with the `/usr/dict/words` list. This is a simple dictionary with english words. We tested with 45000 words. Basically all the words were inserted and the looked up. The CPHSTL implementation had a fixed jump table of size 8192, the chunksize was 10. The test was performed on the PC. Results:

test	SGI [s]	CPHSTL [s]
insert	0.08	0.06
lookup	0.036	0.041
erase	0.04	0.06

The results are a little disappointing. Our implementation is *losing* all tests. The insert test is not very good because the SGI implementation is resizing under the insert and is therefore doing more work than our implementation.

The random test

The second test was based on random strings of length $1 - 32$ characters. The hash function, i.e. distribution, is not tested here (because the keys are already random). This is a test targeted directly at the storage design. We tested with a different number of words. We first built a hash table with the number of elements and then searched it. The tests was performed at the server. Only the lookup test was performed, and only on existing keys. The size of the jump table in our implementation was set to $N/10$, where N was the number of elements inserted.

The results are shown in figure **??**, page **??**. The SGI implementation is clearly faster, but not much. Also shown is our implementation with the image saving disabled. This is mentioned in the discussion later. The curve for our implementation is not very straight, but this is conjectured to be noise. The ripples are small and were not placed at the same places after each run. The ripples is at very large input sizes, so it should not be a cache effect. On the other hand the data is still hold in main memory so paging can not explain it either.

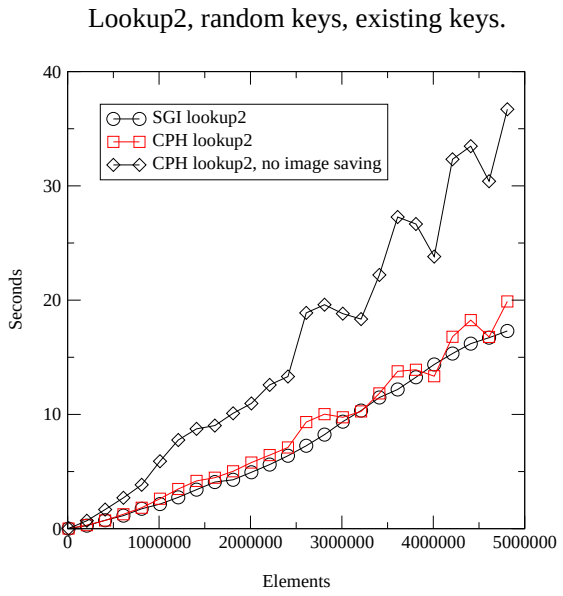

Figure 3: The random test, existing keys.

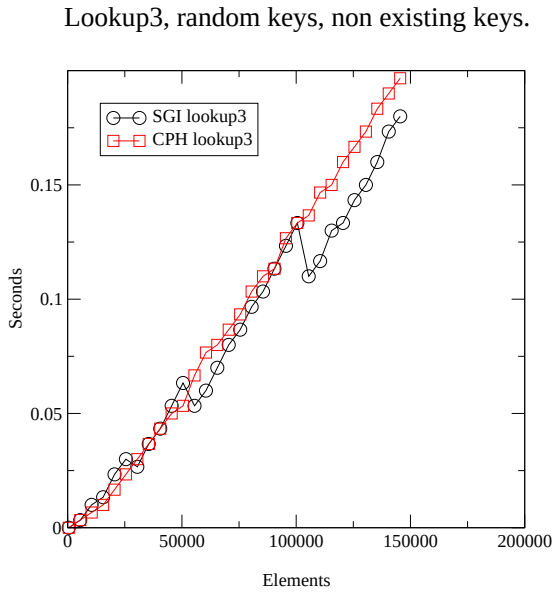

Figure 4: The random test, non existing keys.

The first test did not include lookups on non existing keys, so we did another test where only non existing keys were looked up. This was done on the PC due to problems with the server.

The results areshown in figure **??**, page **??**. The SGI implementation is still faster, but it is now very close.

Test machines and compiler

We used two different machine types for all testing:

Linux PC for small tests we used a Linux pc with an Intel Pentium III Katmai, 500 MHz, 512 KB cache and 1024 MB ram.

SUN Server for the big tests we used a SUN Enterprise 6500 server running Solaris, with 24 400 MHz ultraSPARC II cpus, 8MB cache and 24 GB ram. The tests used one cpu.

On both machines the gcc-compiler was used. The optimization flag `-O6` was used.

7 Discussion

We tested our implementation against the SGI implementation, and our implementation is not faster. Our guess is that the fastbucket idea is not as good as assumed. It is not clear why it is *so* bad, but maybe the complexity, including more loops with branches, and the generally very short lists are part of the explanation. Also the SGI implementation is more *lean*.

We have tried to verify the saving of the image by testing our implementation without support for saving the images. The result was a great performance loss, which can be seen on figure **??**, page **??**. This indicates that the image saving actually is a meaningfull concept. The only thing changed in this configuration was the use of the saved images. The implementation was still suffering from checks and memory use, which gave nothing. Thus the test should not be used to make too optimistic conclusions, but we think we can retrofit the hash image reuse to the SGI implementation and get a faster result. This is planned work.

Another test we did was to substitute the fast_bucket by a simple single linked list – that brought us down on par with SGI, which is actually pretty good. If we assume the SGI implementation is better tuned, it should be possible to tune our implementation as well and get a faster result.

What we really need is tools to analyze cache misses and hits. It seems very important to fully understand what happens with memory before we can improve the design for such simple data structures like hash tables.

The simple resizing strategy in the SGI design is maybe another place to start looking for optimizations.

There is also still space for improvements on *adaptive* hash tables – we could use boost tricks to get information about the keys and data. Size could be used to optimize for caching, key-type could be used to choose a hashfunction.

Status on implementation

The implementation is not ready for use. The code written is working, but first of all the code is not faster than the alternative from SGI and secondly the implementation lacks support for resizing, iterators and some of the helper functions. The current implementation is not reviewed for exception safety and is not namespace compliant.

Other work done

As a part of this project we wrote a simple framework for unit testing. There are some notes in danish on `http://www.diku.dk/undervisning/2000e/e00.505/Minutes-2000.12.04.` the framework will be documented in near future.

Acknowledgement

The authors would like to thank the Copenhagen STL team for discussion and comments, we are especially grateful to Jyrki and Lars.

8 References

C++std The C++ Standard: `http://anubis.dkuug.dk/jtc1/sc22/wg21/`.

cphstl Copenhagen STL: `http://www.cphstl.dk/`.

Dumey A. I Dumey: Computers and Automation 5 (1956) 6-9.

Katajainen Jyrki Katajainen and Mikkel Lykke: Experiments with universal hashing, ISSN 0107-8283.

Knuth Donald E. Knuth: The Art of Computer Programming, 1st edition Addison Wesley, 1973.

SGI The SGI STL `http://www.sgi.com/tech/stl/`.

Stroustrup Bjarne Stroustrup: The C++ Programming Lanuage, 3rd edition, Addison Wesley, 1997.

9 Appendix A: Some testing on hash functions

There is of course a concern other than the quality (i.e. image distribution) of the hash function – it is also important that the function is cheap in instructions to minimize the constants in practical use.

We evaluated a collection of hash functions looking at the distribution and the speed.

We tested distribution by hashing the words in the list `/usr/dict/words` (45403 words) on a Linux-host and looked at how many words hashed to the same value.

	name	speed [Mbyte/s]	distribution
1	rand()	-	perfect
2	md5	18	perfect
3	SGI	250	ok
4	random table (256)	250	bad
5	random table (4*256)	167	ok
6	random table (4*64)	42	ok
7	random table (IV+256)	71	perfect

The full results can be found in `hash_test.out` (page **??**). The code used is in `hash_test.cc` (page **??**).

We used the *RSA Data Security, Inc. MD5 Message-Digest Algorithm*, the implementation was provided by RSA Data Security, Inc.

10 Appendix B: Files

`stl_hash_map.h` (page **??**).

`stl_hash_functions.h` (page **??**).

`fast_bucket.h` (page **??**).

`chunk.h` (page **??**).

`stl_hash_fun.h` The modified SGI implementation, the hash function for `char*` is changed and an implementation for string is added (page **??**).

`stl_hash_map_utest.cc` The unittesting module for fast bucket (page **??**).

`times.cc` The program used for benchmarking against the SGI implementation (page **??**).

`test.h` Some ugly macros used for timing, clock from time.h is used (page **??**).

`hash_test.cc` The program used to benchmark distribution and speed of hash functions (page **??**).

`hash_test.out` The results from `hash_test.cc` (page **??**).

`makefile` (page **??**).

10.1 stl_hash_map.h

```
/*! \file stl_hash_map.h
 * \brief This header provides the definitions and declaration of the hash_map.

 * \author Christian Boesgaard <pink@diku.dk> and Jacob C. Poulsen <ja7@diku.dk>.
 */

#ifndef __STL_HASH_MAP_H__
#define __STL_HASH_MAP_H__

#include <utility>
#include <vector>
#include "fast_bucket.h"
#include "stl_hash_functions.h"

template <class key_t, class value_t, class Hash_t, class Equal>
```

```
class hash_map {

     typedef pair<const key_t,value_t> pair_t;
     typedef pair<const key_t,value_t> value_type;
     typedef const pair_t* const_iterator;
     typedef pair_t* iterator;
     typedef unsigned long hashvalue_t;
     typedef fast_bucket<hashvalue_t,key_t,value_t> fast_bucket_t;
public:

#ifndef JTABLE_SIZE
#define JTABLE_SIZE 8192
#endif

     hash_map(size_t size = JTABLE_SIZE) : jump_table_size_(size), size_(0){
       jump_table = new (fast_bucket_t*)[size];
       memset(jump_table,0,sizeof(fast_bucket_t*)*size);
     }
     ~hash_map() {
  destroy(jump_table, jump_table+jump_table_size());
     }

     iterator end() {return 0;}

     value_t& operator[](const key_t&);

     /*
       NOT IMPLEMENTED:
       pair_t* insert(const key_t&);
       pair<iterator, bool> insert(const value_type& val);
       void erase(iterator pos);
       const_iterator find(const key_t& key);
     */

     iterator find(const key_t& key) {
  hashvalue_t hv = h(key);
  size_t index = hv % jump_table_size();
  if(jump_table[index]) {
       fast_bucket_t* fb = jump_table[index];
       return fb->find(hv, key);
  }
  return end();
     }


// should return size_type?? bjarne 487!!!
     void erase(const key_t& key) {
  hashvalue_t hv = h(key);
  size_t index = hv % jump_table_size();
  if(jump_table[index]) {
       fast_bucket_t* fb = jump_table[index];
       fb->erase(hv, key);
  }
     }

     pair<iterator, bool> insert(const value_type& val) {
  key_t key = val.first;
  hashvalue_t hv = h(key);
  size_t index = hv % jump_table_size();
  if(!jump_table[index]) {
       fast_bucket_t* fb = new fast_bucket_t();
       jump_table[index] = fb;
       fb->insert_last(hv, val);
       return pair<iterator, bool>(fb->end_pair(), true);
  }
  fast_bucket_t* fb=jump_table[index];
  return fb->insert(hv, val);
     }


     size_t size() {return size_;}
     size_t max_size() {return size_*1000;}
private:
     size_t jump_table_size_;
```

```
     size_t jump_table_size() { return jump_table_size_;}
     size_t size_;
     fast_bucket_t** jump_table;
     Hash_t h;
};

template <class key_t, class value_t, class Hash_t, class Equal>
value_t& hash_map<key_t,value_t,Hash_t,Equal>::operator[](const key_t& key) {
     hashvalue_t hv = h(key);
     size_t index = hv % jump_table_size();
     if(!jump_table[index]) {
  fast_bucket_t* fb = new fast_bucket_t();
  jump_table[index] = fb;
  fb->insert_last(hv,pair_t(key,value_t()));
  return fb->end_value();
     }
     fast_bucket_t* fb=jump_table[index];
     return fb->find_or_insert(hv, key);
}

#endif
```

10.2 stl_hash_functions.h

```
/*! \file stl_hash_functions.h
 * \brief This header provides the definitions and declaration of hash functions used in hash_map.

 * \author Christian Boesgaard <pink@diku.dk> and Jacob C. Poulsen <ja7@diku.dk>.
 */


#ifndef __STL_HASH_FUNCTIONS_H__
#define __STL_HASH_FUNCTIONS_H__

#include <string>

typedef unsigned int uint;
typedef unsigned char byte;

class hash_jyrki {
public:
  hash_jyrki(uint s = 0) {
    srand(s);
    for (int i = 0; i < 1024; ++i) {
T[i] = (uint)rand();
    }
  }
  uint operator() (char const* s) {
    uint t = 0;
    byte* p = (byte*)s;
    uint i = 0;
    for (; *p != 0; ++i) {
      t ^= T[((0x3 & i) << 8) + *p++];
    }
    t ^= T[i];
    return t;
  }

private:
  uint T[1024];
};

template <class t>
struct hash_stl {
  long operator() (const t& s);
};

template<>
struct hash_stl<const char*> {
  long operator() (const char * s) {
    uint t = 0;
    while (*s != 0) {
      t = 5 * t + *s++;
```

```
    }
    return t;
  }
};

template<>
struct hash_stl<string> {
  long operator() (const string& ss) {
    const char*  s=ss.c_str();
    uint t = 0;
    while (*s != 0) {
      t = 5 * t + *s++;
    }
    return t;
  };
};



class hash_crypto {
#define mix(a, b, c) { \
a -= b; a -= c; a ^= (c >>13); \
b -= c; b -= a; b ^= (a <<8); \
c -= a; c -= b; c ^= (b >>13); \
a -= b; a -= c; a ^= (c >>12); \
b -= c; b -= a; b ^= (a <<16); \
c -= a; c -= b; c ^= (b >>5); \
a -= b; a -= c; a ^= (c >>3); \
b -= c; b -= a; b ^= (a <<10); \
c -= a; c -= b; c ^= (b >>15); \
}

public:
  uint operator() (char const* k);
};

uint hash_crypto::operator() (char const* k) {
    uint length = 0;
    while (*k++) {++length;}
    uint len = length;
    uint a = 0x9e3779b9;
    uint b = 0x9e3779b9;
    uint c = 0x8f256195;

    while (len >= 12) {
      a += (k[0] + (uint(k[1])<<8) + (uint(k[2])<<16) + (uint(k[3])<<24));
      b += (k[4] + (uint(k[5])<<8) + (uint(k[6])<<16) + (uint(k[7])<<24));
      c += (k[8] + (uint(k[9])<<8) + (uint(k[10])<<16) + (uint(k[11])<<24));
      mix(a, b, c);
      k += 12; len -= 12;
    }

    c += length;
    switch (len) {
      case 11: c+=(uint(k[10])<<24);
      case 10: c+=(uint(k[9])<<16);
      case 9 : c+=(uint(k[8])<<8);

      case 8: b+=(uint(k[7])<<24);
      case 7: b+=(uint(k[6])<<16);
      case 6: b+=(uint(k[5])<<8);
      case 5: b+=k[4];

      case 4: a+=(uint(k[3])<<24);
      case 3: a+=(uint(k[2])<<16);
      case 2: a+=(uint(k[1])<<8);

      case 1: a+=k[0];
    }
    mix(a, b, c);
    return c;
}

#endif
```

10.3 fast_bucket.h

```
/*! \file fast_bucket.h
 * \brief This header provides the definitions and decleration of the fast_bucket class used in hash_map

 * \author Christian Boesgaard <pink@diku.dk> and Jacob C. Poulsen <ja7@diku.dk>.
 */

// using pair
#include <utility>

// using operator new
#include <memory>

#include "chunk.h"

#define chunk_size 10

template <class H, class K, class T>
class fast_bucket {
  fast_bucket(const fast_bucket<H, K, T>&);
  const fast_bucket<H, K, T>& operator=(const fast_bucket<H, K, T>&);
  typedef pair<const K, T> pair_t;
  typedef chunk<H, K, T, chunk_size> chunk_;
  typedef T value_t;
  typedef K key_t;
private:
public:
  // chunks
  chunk_* first; // the first chunk in the single linked list

  // housekeeping for first chunk
  // this is located in the fast_bucket class because it is the only
  // place it is used and it comes with a price in space
  uint end_pos; // is index to the "end iterator" (one past...)

  // find to use when deleting elements, we will move the last
  // element in the bucket to the place
  // hpp is a place to return a pointer to the small hashvalue
  pair_t* find_d(H hh, const K& kk, H** hpp) {
       // check the first chunk
       pair_t* t;
       if (t = first->find_d(hh, kk, hpp, end_pos)) {return t;}

       // check the rest
       chunk_* c = first->next;
       while (c) {
    pair_t* pp;
    if ((pp = c->find_d(hh, kk, hpp)) != 0) {return pp;}
    c = c->next;
       }
       return 0;
  }

  // move last element to a place according to args
  // this is the delete operation, so we need to do it the nice way!
  void move_last_element_to(pair_t* pp, H* hp) {
    // is the first chunk empty?
       if (end_pos == 0) {
    // the chunk is empty -- get rid of it
    chunk_* trash = first;
    first = trash->next;
    end_pos = chunk_size;
    delete trash;
       }

    const uint last_element_pos = --end_pos;
    *hp = first->h[last_element_pos];
    destroy(pp);

    // do inplace new
    new(pp) pair_t(first->d[last_element_pos]);
    destroy(&(first->d[last_element_pos]));
  }
```

```
  // inserting into the first chunk
  pair_t* insert_last(H hh, const pair_t& p){
    uint last_element_pos;
    // common case first to help pipeline
    if (end_pos != chunk_size) {
      last_element_pos = end_pos++;
      first->set(hh, p, last_element_pos);
    }
    else {
      // no place left, add a chunk
      add_chunk();
      last_element_pos = end_pos++;
      first->set(hh, p, last_element_pos);
    }
    return &(first->d[last_element_pos]);
  }

  void add_chunk() {
       chunk_* second = first;
       first = new chunk_;
       first->next = second;
       end_pos = 0;
  }

public:
  fast_bucket() {
    first = new chunk_;
    end_pos = 0; // end position
  }
  ~fast_bucket() {
    // just destroy data and delete the chunks
       chunk_* c = first->next;
    while (c) {
 chunk_* trash = c;
 c = c->next;
 trash->free_data();
 delete trash;
    }
    first->free_data(end_pos);
    delete first;
  }

  // return size of bucket in valid elements
  uint size() {
    uint chunks = 0; // n chunks excl. the last
    chunk_* c = first->next;
    while (c) {
      c = c->next;
      ++chunks;
    }
    // the elements in the first chunk is end_pos
    return (chunks * chunk_size) + end_pos;
  }
  pair_t* find(H hh, const K& kk) {
       // first chunk
       pair_t* pp;
       if (pp = first->find_p(hh, kk, end_pos)) {return pp;}
       // the rest
       chunk_* c = first->next;
       while (c) {
    pair_t* pp;
    if ((pp = c->find_p(hh, kk)) != 0) {return pp;}
    c = c->next;
       }
       return 0;
  }

  // insert element in bucket
  pair<pair_t*, bool> insert(H hh, const pair_t& p) {
    // check if the key is here
    if (pair_t* pp = find(hh, p.first)) {
      // it is -- overwrite
```

```
      pp->second = p.second; ///!!! is this ok??
      return pair<pair_t*, bool>(pp, false);
    }
    else {
      // insert last
      return pair<pair_t*, bool>(insert_last(hh, p), true);
    }
  }

  // find or insert element
  value_t& find_or_insert(H hh, const key_t& k) {
    // check if the key is here
    if (pair_t* pp = find(hh, k)) {
      return pp->second;
    }
    pair_t* pp = insert_last(hh, pair_t(k, value_t()));
    return pp->second;
  }


  // delete element from bucket
  void erase(H hh, const K& k) {
    // check if the key is here
    H* hp;
    pair_t* pp;
    if (pp = find_d(hh, k, &hp)) {
      // it is -- overwrite with the last element
 move_last_element_to(pp, hp);
    }
  }

  // end_value returns a value& to the last element.
  value_t& end_value() {
    uint last_element_pos = end_pos - 1;
    return first->d[last_element_pos].second;
  }

  // end_value returns a pointer to the last pair.
  pair_t* end_pair() {
    uint last_element_pos = end_pos - 1;
    return &(first->d[last_element_pos]);
  }
};
```

10.4 chunk.h

```
/*! \file deque
 * \brief This header provides the definitions and decleration of the chunk class used in hash_map
 * \author Christian Boesgaard <pink@diku.dk> and Jacob C. Poulsen <ja7@diku.dk>.
 */

// #include <alloc.h>
#include <iterator>

// using pair
#include <utility>

// using operator new
#include <memory>

// Type legend:
// H: (small) key type, should support effective ==
// K: key type
// T: content type

typedef unsigned int uint;
```

```
template <class H, class K, class T, uint chunk_size>
class chunk {
  typedef pair<const K, T> pair_t; // <key, value> pair
public:
  chunk(): next(0){
    d = reinterpret_cast<pair_t*>(dd);
  }
  void free_data(size_t end_pos=chunk_size) {
    destroy(d,d+end_pos);
  };


  H h[chunk_size]; // hash values
  chunk* next; // next chunk
///  chunk* prev; // previous chunk
  // we're using a char[] to get some "raw" memory
  pair_t* d; // iterator used to access data pairs
  char dd[chunk_size * sizeof(pair_t)]; // space for data pairs


  // the find functions will return a 0-pointer if the element was not found

  // find to use when doing lookups and inserts
  pair_t* find_p(H hh, const K& kk, uint last = chunk_size) {
    for (uint i = 0; i < last; ++i) {
#ifdef N_USE_HASH
      if (kk == (d[i]).first) {return &d[i];}
#else
      if (hh == h[i] && kk == (d[i]).first) {return &d[i];}
#endif
    }
    return 0;
  }

  // find to use when deleting elements, we will move the last
  // element in the bucket to the place
  // hpp is a place to return a pointer to the small hashvalue

  pair_t* find_d(H hh, const K& kk, H** hpp,
uint last = chunk_size) {
    for (uint i = 0; i < last; ++i) {
#ifdef N_USE_HASH
      if (kk == (d[i]).first) {
#else
      if (hh == h[i] && kk == (d[i]).first) {
#endif
*hpp = &h[i];
return &d[i];
      }
    }
    return 0;
  }
  // set a value
  // this function is the only one used to set elements directly
  void set(H hh, const pair_t& p, uint i) {
    h[i] = hh;
    // do inplace new
    pair_t* pp = &d[i];
    new(pp) pair_t(p);
  }
};
```

10.5 stl_hash_fun.h

```
/*
 * Copyright (c) 1996
 * Silicon Graphics Computer Systems, Inc.
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Silicon Graphics makes no
```

```
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Hewlett-Packard Company makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 */

/* NOTE: This is an internal header file, included by other STL headers.
 *   You should not attempt to use it directly.
 */

#ifndef __SGI_STL_HASH_FUN_H
#define __SGI_STL_HASH_FUN_H

#include <stddef.h>

__STL_BEGIN_NAMESPACE

template <class Key> struct hash { };

inline size_t __stl_hash_string(const char* s)
{
  unsigned long h = 0;
  for ( ; *s; ++s)
    h = 5*h + *s;

  return size_t(h);
}

__STL_TEMPLATE_NULL struct hash<char*>
{
  size_t operator()(const char* s) const { return __stl_hash_string(s); }
};

__STL_TEMPLATE_NULL struct hash<const char*>
{
  size_t operator()(const char* s) const { return __stl_hash_string(s); }
};

#include <string>

__STL_TEMPLATE_NULL struct hash<string>
{
  size_t operator()(const string& s) const {
    return __stl_hash_string(s.c_str());
  }
};

__STL_TEMPLATE_NULL struct hash<const string>
{
  size_t operator()(const string& s) const {
    return __stl_hash_string(s.c_str());
  }
};




__STL_TEMPLATE_NULL struct hash<char> {
  size_t operator()(char x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<unsigned char> {
  size_t operator()(unsigned char x) const { return x; }
```

```
};
__STL_TEMPLATE_NULL struct hash<signed char> {
  size_t operator()(unsigned char x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<short> {
  size_t operator()(short x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<unsigned short> {
  size_t operator()(unsigned short x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<int> {
  size_t operator()(int x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<unsigned int> {
  size_t operator()(unsigned int x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<long> {
  size_t operator()(long x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<unsigned long> {
  size_t operator()(unsigned long x) const { return x; }
};

__STL_END_NAMESPACE

#endif /* __SGI_STL_HASH_FUN_H */

// Local Variables:
// mode:C++
// End:
```

10.6 stl_hash_map_utest.cc

```
#include "utest.h"
#include "stl_hash_map.h"

class stl_hash_map_utest : public Test {
     hash_map<string, int, hash_stl<string>,int> hm;
public:
     typedef pair<hash_map<string, int, hash_stl<string>,int>::iterator, bool> p_;
     typedef hash_map<string, int, hash_stl<string>,int>::value_type v_;

     stl_hash_map_utest() : hm() {}
     void run() {
  string s0("alpha");
  _test(hm[s0] == 0);

  string s1("bravo");
  hm[s1] = 1;
  _test(hm[s0] == 0);
  _test(hm[s1] == 1);

  string s2("charlie");
  hm[s2] = 2;
  _test(hm[s0] == 0);
  _test(hm[s1] == 1);
  _test(hm[s2] == 2);

  _test((hm.find(s2))->second == 2);

  hm.erase(s2);

  _test(hm.find(s2) == 0);


  string s3("delta");
  _test(hm[s3] == 0);


  string s4("echo");
  v_ v1 = v_(s4, 4);

  p_ p1 = hm.insert(v1);
```

```
  _test((p1.first)->second == 4);
  _test(p1.second == true);

  p_ p2 = hm.insert(v_(s4, 8));
  _test((p2.first)->second == 8);
  _test(p2.second == false);


     }
};

int main(int argc, char* argv[]) {
     stl_hash_map_utest t;
     if (argc > 1) {
  t.set_stream(&cout);
     }
     t.run();
     return int(t.report());
}
```

10.7 times.cc

```
#include <cstdlib>
#include <string>
#include <vector>

inline size_t __stl_hash_string2(const char* s)
{
  unsigned long h = 0;
  for ( ; *s; ++s)
    h = 5*h + *s;

  return size_t(h);
}

struct hashstring
{
  size_t operator() (const string& s) const {
    return __stl_hash_string2(s.c_str());
  }
};



#ifdef FAST

#include "stl_hash_map.h"
typedef hash_map<string, int, hashstring, int> h_t;

#else

#include <hash_map>
//#include "stl_hash_fun.h"
//typedef hash_map<const char*, int> h_t;
typedef hash_map<string, int, hashstring> h_t;
#endif

#include <fstream>

#include "pink_types.h"
#include "test.h"




//  inline uint sum_h(uint m, string* s, hash_map<char const*, int>& h) {
//    uint ss = 0;
//    for (uint i = 0; i < m; ++i) {
//      ss ^= h[(s[i]).c_str()];
//    }
```

```
//    return ss;
//  }

//  inline void fill_h(uint m, string* s,hash_map<char const*, int>& h) {
//    for (uint i = 0; i < m; ++i) {
//      h[(s[i]).c_str()] = i;
//    }
//  }

//  inline uint sum_fh(uint m, string* s, fhash& h) {
//    uint ss = 0;
//    for (uint i = 0; i < m; ++i) {
//      ss ^= h.find((s[i]).c_str());
//    }
//    return ss;
//  }



//  inline void fill_(uint m, string* s, h_t& h) {
//    for (uint i = 0; i < m; ++i) {
//      h[s[i].c_str()] = i;
//    }
//  }


inline void fill_s(uint m, string* s, h_t& h) {
  for (uint i = 0; i < m; ++i) {
    h[s[i]] = i;
  }
}

inline void insert_s(uint m, string* s, h_t& h) {
  for (uint i = 0; i < m; ++i) {
    h.insert(pair<string, uint>(s[i], i));
  }
}

inline void lookup_s(uint m, string* s, h_t& h) {
  for (uint i = 0; i < m; ++i) {
    h.find(s[i]);
  }
}

inline void lookup_same_s(uint times, string* s, h_t& h) {
  for (uint i = 0; i < times; ++i) {
    h.find(s[40000]);
  }
}


inline void erase_s(uint m, string* s, h_t& h) {
  for (uint i = 0; i < m; ++i) {
    h.erase(s[i]);
  }
}



// lav fyld delg
inline void m_fill_s(uint m, string* s) {
  h_t h;
  for (uint i = 0; i < m; ++i) {
    h[s[i]] = i;
  }
  for (uint i = 0; i < m; ++i) {
    h.erase(s[i]);
  }
}
```

```
//  inline uint thash(uint m, string* s, hash_base& h) {
//    uint sum = 0;
//    for (uint i = 0; i < m; ++i) {
//      sum += h((s[i]).c_str());
//    }
//    return sum;
//  }


#ifndef SIZE
#define SIZE 100000
#endif

int main(int argc, char* argv[]) {

  char* file = "/usr/dict/words";

// test file
    if (argc > 1) {
      file = argv[1];
    }

// test max size
    uint size = SIZE;
    if (argc > 2) {
      size = atol(argv[2]);
    }
    uint max = size;

// jtable size
    uint jsize = 8192;
    if (argc > 3) {
      jsize = atol(argv[3]);
    }

#ifdef FAST
    //    h_t h = h_t(jsize);
    //cerr << "jumptable: " << jsize << endl;
#else
    //h_t h;
#endif


string* all = new string[SIZE + 10];
  uint n_data = 0;

  // read keys in
  ifstream data(file, ios::in);   // datast-fil(navn)

  cerr << "reading file...\n";

  while (!data.eof() && (n_data < size)) {
    ++n_data;
    string s;
    data >> s;
    all[n_data] = s;
    //    if ((n_data % 10000) == 0) {cerr << "xxx " << s << endl;}
  }
  cerr << "read " << n_data <<" words\n";

  max = n_data;

  const int start = 1000;
  const int step = 10000;

#ifdef INSERT
  cerr << "testing insert ...\n";
  for (uint i = start; i < max; i += step) {
#ifdef FAST
       h_t hh = h_t(jsize);
#else
       h_t hh = h_t();
#endif
       cerr << "insert " << i << " max " << max << endl;
```

```
      TEST3(1, insert_s(i, all, hh), i);
  }
#endif



#ifdef LOOKUP
  cerr << "filling hash ...\n";
  insert_s(n_data, all, h);

  cerr << "testing lookup (10 times)...\n";
  for (uint i = start; i < max; i += step) {
    cerr << "lookup " << i << " max " << max << endl;
      TEST3(2, lookup_s(i, all, h), i);
  }
#endif


#ifdef LOOKUP2
  cerr << "testing lookup II ...\n";
  for (uint i = start; i < max; i += step) {
#ifdef FAST
    h_t hh = h_t(i/10);
#else
      h_t hh = h_t();
#endif
      cerr << "inserting: " << i<< endl;
      insert_s(i, all, hh);
      TEST3(2, lookup_s(i, all, hh), i);
  }
#endif


#ifdef LOOKUP3
  cerr << "testing lookup III ...\n";
  for (uint i = start; i < max; i += step) {
      uint sz = i / 2;
#ifdef FAST
    h_t hh = h_t(sz/10);
#else
      h_t hh = h_t();
#endif
      cerr << "inserting: " << sz << endl;
      insert_s(sz, all, hh);
      TEST3(3, lookup_s(sz, all+(max/2), hh), sz);
  }
#endif



#ifdef ERASE
  cerr << "testing erase ...\n";
  for (uint i = start; i < max; i += step) {
      insert_s(i, all, h);
      cerr << "erase " << i << " max " << max << endl;
      TEST3(1, erase_s(i, all, h), i);
  }
#endif

  cerr << "using file: " << file << ", words tested: " << n_data << endl;
}
```

10.8 test.h

```
#ifndef __TEST_H_

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

/* For simple timebenchmarking
```

```
brug:

TEST(T, CODE)

Hvor T er hvor mange gange koden skal udfres og
CODE selve koden.

Eks.:

TEST(1000, 2+2;)

Husk det sidste ';'!

 */

#define TEST(T, CODE)\
{\
     unsigned long t_ = clock();\
     for (unsigned long i = 0; i < T; i++) {\
  CODE; \
     }\
     t_ = clock() - t_;\
     printf("time: %.6f secs\n", (float)t_ / CLOCKS_PER_SEC);\
}

/* TEST2 tager et ekstra argument som er en unsigned long int, hvor
   tidstagningen gemmes */

#define TEST2(T, CODE, V)\
{\
     V = clock();\
     for (unsigned long i__ = 0; i__ < T; i__++) {\
  CODE; \
     }\
     V = clock() - V;\
}


#define TEST3(T, CODE, I)\
{\
     unsigned long t_ = clock();\
     for (unsigned long i__ = 0; i__ < T; i__++) {\
  CODE; \
     }\
     t_ = clock() - t_;\
     printf("%d\t%.6f\n", I, ((float)t_/T)/ CLOCKS_PER_SEC);\
}


#define __TEST_H_
#endif
```

10.9 hash_test.cc

```
// $Id: cphstl-report-2001-5.tex,v 1.6 2005/02/07 16:18:48 jyrki Exp $


#include <iostream>
#include <fstream>
#include <algorithm>
#include <string>
#include <vector>
#include <list>
#include <map>
#include <cmath>
#include "test.h"


#include <cstdlib>

extern "C" {
#include "global.h"
#include "md5.h"
```

```
}

typedef unsigned char byte;
typedef unsigned int uint;

class hash_function {
public:
     virtual uint hash(void* s, uint size) = 0;
     virtual const char* name() = 0;
     virtual ~hash_function() {}
};

class hash_report {
public:
     hash_report(uint m = 11) {
  max_elements = m;
  hits = new vector<unsigned short>(max_elements, 0);
  max_hits_all_bits = 256;
  hits_all_bits = new vector<uint>(max_hits_all_bits);
  over_max_hits_all_bits = 0;
  speed = 0;
     }
     ~hash_report() {
  delete hits;
  delete hits_all_bits;
     }



     string name;
     uint elements;
     uint buckets;
     uint misses;
     uint max_elements;

// for speed calculations
     double speed;

     vector<unsigned short>* hits;

     uint max_hits_all_bits;
     vector<uint>* hits_all_bits; // hvor mange hasher til samme fulde bitmnster
     uint over_max_hits_all_bits;

     uint over;
     float avg_over;

     void print_hits_all_bits() {
  cout << "================================================================================\n";
  cout << name << endl;
  cout << "================================================================================\n";
  cout << "elements: " << elements << endl;
  cout << "================================================================================\n";
  for (uint i = 1; i < max_hits_all_bits; ++i ) {
       if (uint t = (*hits_all_bits)[i]) {
    cout << "value(s) with " << i << " hits: " << t << endl;
       }
  }
  cout << "================================================================================\n";
     }

     void print() {
  cout << "================================================================================\n";
  cout << name << endl;
  cout << "================================================================================\n";
  cout << "speed: " << speed << "Mb/s\n";
  cout << "================================================================================\n";
  cout << "elements: " << elements << endl;
  cout << "buckets: " << buckets << endl;
  cout << "load factor: " << (float)elements/buckets << endl;
  cout << "================================================================================\n";
  cout << "misses: " << misses << endl;
  cout << "buckets with more than " << max_elements - 1<< " elements: "
       << over << " with " << avg_over << " elements" << endl;
```

```
  cout << "================================================================================\n";
  for (uint i = 1; i < max_elements; ++i ) {
       cout << "buckets with " << i << " element: " << (*hits)[i] << endl;
  }
  cout << "================================================================================\n";
     }
};


//-----------------------------------------------------------//

class hash_eval {
public:
#define dont_care_size 1024 * 1024
     char dont_care[dont_care_size];
     hash_eval(uint b) {
  n_buckets = b;
     }

     // z er max i intervallet af bucketfills der gives tal p
     hash_report* eval(hash_function* h, const char* file = "/usr/dict/words", uint z = 11) {
  hash_report* r = new hash_report(z);
  vector<uint> buckets(n_buckets, 0);
  map<int, int> h_vals;
  typedef map<int, int>::const_iterator CI;
  uint n_data = 0;
  ifstream data(file, ios::in);   // datast-fil(navn)
  while (!data.eof()) {
       ++n_data;
       string s;
       data >> s;
       uint h_ = h->hash((void*)s.c_str(), s.length());
       // Tl vrdi i map op (initaliseres til 0)
       h_vals[h_]++;
       ++(buckets[h_ % n_buckets]);
  }

  // beregn hits (all bits)

  vector<uint>* hits_all_bits = r->hits_all_bits;
  for (CI i = h_vals.begin(); i != h_vals.end(); ++i) {
       uint hits = i->second;
       if (hits < r->max_hits_all_bits) {
    ++((*hits_all_bits)[hits]);
       } else {
    ++(r->over_max_hits_all_bits);
       }
  }

  r->name = string(h->name()) + " [" + file + "]";
  r->elements = n_data;
  r->buckets = n_buckets;
  r->misses = 0;
  r->over = 0;
  vector<unsigned short>* hits = r->hits;

  uint n_over = 0;
  // beregn afvigelse
  for (uint i = 0; i < n_buckets; ++i) {
       uint n = buckets[i]; // antal elementer
       // indst i tabel over afvigelser:
       if (n == 0) {++(r->misses);}
       else if (n < r->max_elements) {++(*hits)[n];}
       else {++(r->over); n_over += n;}
  }
  r->avg_over = r->over ? (float)n_over / r->over : 0;

  //beregn hastighed

  unsigned long int clock_ticks;
  TEST2(5, h(dont_care, dont_care_size), clock_ticks);
  r->speed = ((double)clock_ticks / 5)/ CLOCKS_PER_SEC;
  return r;
     }
```

```
    // z er max i intervallet af bucketfills der gives tal p
    // n = navn
    // N = antal elementer
    // s = strrelse
    hash_report* eval_random(hash_function* h, uint N, uint s = 4, uint z = 11) {
  hash_report* r = new hash_report(z);
  vector<uint> buckets(n_buckets, 0);
  uint n_data = N;
  r->elements = n_data;
  uint size = s;

  ifstream data("/dev/urandom", ios::in);   // datast-fil(navn)
  while (n_data != 0) {
       --n_data;
       byte buf[size];
       data.read(buf, size);
       uint h_ = h->hash((void*)buf, size) % n_buckets;
       ++(buckets[h_]);
  }

  r->name = string(h->name()) + " [random]";
  r->buckets = n_buckets;
  r->misses = 0;
  r->over = 0;
  vector<unsigned short>* hits = r->hits;

  uint n_over = 0;
  // beregn afvigelse
  for (uint i = 0; i < n_buckets; ++i) {
       uint n = buckets[i]; // antal elementer
       // indst i tabel over afvigelser:
       if (n == 0) {++(r->misses);}
       else if (n < r->max_elements) {++(*hits)[n];}
       else {++(r->over); n_over += n;}
  }
  r->avg_over = r->over ? (float)n_over / r->over : 0;
  return r;
    }
private:
    uint n_buckets;
};


// rand
class RAND: public hash_function {
public:
    const char* name() {return "rand() (non-hash)";}
    RAND(uint s = 0) {srand(s);}
    uint hash(void* s, uint size) {
  return (uint)rand();
    }
};



// fra STL-forslaget
class STL: public hash_function {
public:
    const char* name() {return "sgi stl hash (strings)";}
    uint hash(void* s, uint size) {
  uint h = 0;
  byte* b = (byte*)s;
  for (uint i = 0; i < size; ++i) {
       h = 5 * h + b[i];
  }
  return h;
    }
};


/*
```

```
class SGI: public hash_function {
public:
  const char* name() {return "sgi stl hash (strings)";}
  uint hash(void* s, uint size) {
    uint h = 0;
    byte* b = (byte*)s;
    for (uint i = 0; i < size; ++i) {
      h = 5 * h + b[i];
    }
    return h;
  }
};

*/



// rotate
class pnk_rot: public hash_function {
public:
     uint rotate (uint i, int r) {
  return ( (i << r) | ( i >> (sizeof(uint)*4 - r)));
     }
     const char* name() {return "pnk_rot";}
     uint hash(void* s, uint size) {
  int n = size / sizeof(uint);
  uint* p = (uint*)s;
  uint h = p[0];

  for (int i = 1; i < n; ++i) {
       h += rotate (p[i], 3 * i);
  }
  if (int r = size % sizeof(uint)) {
       uint I = 0xffffffff;
       byte* p = (byte*)&I;
       byte* q = (byte*)s;
       for (int i = 0; i < r; ++i) {
    p[i] = q[n * sizeof(uint) + i];
       }
       h ^= I;
  }
  return h;
     }
};



class md5: public hash_function {
public:
     const char* name() {return "md5";}
     uint hash(void* s, uint size) {
  MD5Init (&c);
  MD5Update(&c, (unsigned char*)s, size);
  MD5Final(r, &c);
  uint t = *(uint*)r;
  return t;
     }
private:
     MD5_CTX c;
     unsigned char r[16];
};


class pnk1: public hash_function {
public:
     const char* name() {return "pnk1";}
     uint hash(void* s, uint size) {
  uint t = 0;
  byte* p = (byte*)s;
  for (uint i = 0; i < size; ++i) {
       //    uint s = i & 0x11; // hurtig mod 4...
       uint s = i % 4;
       switch(s) {
       case 0:
```

```
    t += p[i] * 5;
    break;
       case 1:
    t += ~p[i] * 5;
    break;
       case 2:
    t += p[i] * p[i];
    break;
       case 3:
    t *= p[i] << 5;
    break;
       }
  }
  return t;
     }
};


class r_table_1: public hash_function {
public:
     const char* name() {return "random table (1)";}
     r_table_1(uint s = 0) {
  srand(s);
  for (int i = 0; i < 256; ++i) {
       T[i] = (uint)rand();
  }
     }
     uint hash(void* s, uint size) {
  uint t = 0;
  byte* p = (byte*)s;
  for (uint i = 0; i < size; ++i) {
       t ^= T[*p++];
  }
  return t;
     }
private:
     uint T[256];
};

class r_table_1s: public hash_function {
public:
     const char* name() {return "random table (1) using size as IV";}
     r_table_1s(uint s = 0) {
  srand(s);
  for (int i = 0; i < 64; ++i) {
       T[i] = (uint)rand();
  }
     }
     uint rotate (uint i, int r) {
  return ( (i << r) | ( i >> (sizeof(uint)*4 - r)));
     }
     uint hash(void* s, uint size) {
  uint t = T[size % 64];
  byte* p = (byte*)s;
  for (uint i = 0; i < size; ++i) {
       t ^= rotate(T[*p++ % 64], i % 32);
  }
  return t;
     }
private:
     uint T[64];
};


class r_table_4: public hash_function {
public:
     const char* name() {return "random table (4*256)";}
     r_table_4(uint s = 0) {
  srand(s);
  for (int i = 0; i < 4; ++i) {
       for (int j = 0; j < 256; ++j) {
    T[i][j] = (uint)rand();
       }
  }
```

```
    }
    uint hash(void* s, uint size) {
  uint t = 0;
  byte* p = (byte*)s;
  for (uint i = 0; i < size; ++i) {
      t ^= T[i % 4][*p++];
  }
  return t;
    }
private:
    uint T[4][256];
};


class pnk_r_table_4: public hash_function {
public:
    const char* name() {return "random table (4*64)";}
    pnk_r_table_4(uint s = 0) {
  srand(s);
  for (int i = 0; i < 4; ++i) {
      for (int j = 0; j < 64; ++j) {
    T[i][j] = (uint)rand();
      }
  }
    }
    uint hash(void* s, uint size) {
  uint t = 0;
  byte* p = (byte*)s;
  for (uint i = 0; i < size; ++i) {
      t ^= T[i % 4][*p++ % 64];
  }
  return t;
    }
private:
    uint T[4][64];
};



class pnk_r_table_1: public hash_function {
public:
    const char* name() {return "random table (1) with simple addition";}
    pnk_r_table_1(uint s = 0) {
  srand(s);
  for (int i = 0; i < 256; ++i) {
      T[i] = (uint)rand();
  }
    }
    uint hash(void* s, uint size) {
  uint t = 0;
  byte* p = (byte*)s;
  for (uint i = 0; i < size; ++i) {
      t += *p;
      t ^= T[*p++];
  }
  return t;
    }
private:
    uint T[256];
};


//  /*
//  class h: public hash_function {
//  public:
//    uint hash(void* s, uint size) {

//    }
//  };

//  */

/*
//const uint n_b = 4540;
```

```
//const uint n_elem = 45403;

#define n_b 4540
#define n_elem 45403

hash_eval t(n_b);

void eval(hash_function* p) {
     hash_report* r = t.eval(p);
     r->print_hits_all_bits();
     delete r;
}
*/


const uint n_b = 4540;
const uint n_elem = 45403;
//const uint n_b = 45403;
//const uint n_b = 100000;

hash_eval t(n_b);

void eval(hash_function* p) {
  hash_report* r = t.eval(p);
  //  r->print();
  r->print_hits_all_bits();

  delete r;
  //  hash_report* r_ = t.eval_random(p, n_elem);
  //  r_->print();
  //delete r_;
}



int main() {

     list<hash_function*> fs;

     fs.push_back(new RAND);

     fs.push_back(new STL);
     fs.push_back(new md5);
     fs.push_back(new r_table_1);
     fs.push_back(new r_table_1s);
     fs.push_back(new r_table_4);
     fs.push_back(new pnk_r_table_4);

     for_each(fs.begin(), fs.end(), eval);

     return 0;

}
```

10.10 hash_test.out

```
================================================================================
rand() (non-hash) [/usr/dict/words]
================================================================================
speed: infMbyte/s
================================================================================
elements: 45403
================================================================================
value(s) with 1 hits: 45399
value(s) with 2 hits: 2
================================================================================
================================================================================
md5 [/usr/dict/words]
================================================================================
speed: 17.8571Mbyte/s
================================================================================
elements: 45403
================================================================================
```

```
value(s) with 1 hits: 45403
================================================================================
================================================================================
sgi stl hash (strings) [/usr/dict/words]
================================================================================
speed: 166.667Mbyte/s
================================================================================
elements: 45403
================================================================================
value(s) with 1 hits: 41399
value(s) with 2 hits: 1617
value(s) with 3 hits: 212
value(s) with 4 hits: 31
value(s) with 5 hits: 2
================================================================================
================================================================================
random table (1) [/usr/dict/words]
================================================================================
speed: 250Mbyte/s
================================================================================
elements: 45403
================================================================================
value(s) with 1 hits: 26209
value(s) with 2 hits: 3696
value(s) with 3 hits: 1296
value(s) with 4 hits: 593
value(s) with 5 hits: 331
value(s) with 6 hits: 191
value(s) with 7 hits: 102
value(s) with 8 hits: 67
value(s) with 9 hits: 40
value(s) with 10 hits: 37
value(s) with 11 hits: 12
value(s) with 12 hits: 15
value(s) with 13 hits: 6
value(s) with 14 hits: 7
value(s) with 15 hits: 4
value(s) with 16 hits: 4
value(s) with 17 hits: 1
value(s) with 18 hits: 3
value(s) with 19 hits: 3
value(s) with 21 hits: 1
================================================================================
================================================================================
random table (1) using size as IV [/usr/dict/words]
================================================================================
speed: 71.4286Mbyte/s
================================================================================
elements: 45403
================================================================================
value(s) with 1 hits: 45403
================================================================================
================================================================================
random table (4*256) [/usr/dict/words]
================================================================================
speed: 125Mbyte/s
================================================================================
elements: 45403
================================================================================
value(s) with 1 hits: 45272
value(s) with 2 hits: 64
value(s) with 3 hits: 1
================================================================================
================================================================================
random table (4*64) [/usr/dict/words]
================================================================================
speed: 100Mbyte/s
================================================================================
elements: 45403
================================================================================
value(s) with 1 hits: 45272
value(s) with 2 hits: 64
value(s) with 3 hits: 1
================================================================================
```

10.11 makefile

```
# all: times.cc
# g++ -O6 -DFAST times.cc -o tfast
# g++ -O6 times.cc -o t

FILES=stl_hash_map_utest.cc fbucket.cc stl_hash_map.h

CC=g++ -g

SIZ=6000000
JS=600000

grotesk:
./t_insert /tmp/grotesk $(SIZ) > t_insert.res &
sleep 2
./tfast_insert /tmp/grotesk $(SIZ) $(JS) > tfast_insert.res &
sleep 2
./t_lookup /tmp/grotesk $(SIZ) > t_lookup.res &
sleep 2
./tfast_lookup /tmp/grotesk $(SIZ) $(JS)> tfast_lookup.res &
sleep 2
./t_erase /tmp/grotesk $(SIZ) > t_erase.res &
sleep 2
./tfast_erase /tmp/grotesk $(SIZ) $(JS)> tfast_erase.res &

all: tfast t

tfast: tfast_insert tfast_lookup tfast_erase

t: t_insert t_lookup t_erase


t_insert: times.cc $(FILES)
$(CC) -g -DINSERT -DSIZE=6000000 -O6 times.cc -o t_insert

t_lookup: times.cc $(FILES)
$(CC) -DLOOKUP -DSIZE=6000000 -O6 times.cc -o t_lookup

t_lookup2: times.cc $(FILES)
$(CC) -DLOOKUP2 -DSIZE=6000000 -O6 times.cc -o t_lookup2


t_lookup3: times.cc $(FILES)
$(CC) -DLOOKUP3 -DSIZE=6000000 -O6 times.cc -o t_lookup3


t_erase: times.cc $(FILES)
$(CC) -DERASE -DSIZE=6000000 -O6 times.cc -o t_erase

tfast_insert: times.cc $(FILES)
$(CC) -DINSERT -DJTABLE_SIZE=500000 -DSIZE=6000000 -O6 -DFAST times.cc -o tfast_insert

tfast_lookup: times.cc $(FILES)
$(CC) -DLOOKUP -DJTABLE_SIZE=500000 -DSIZE=6000000 -O6 -DFAST times.cc -o tfast_lookup

tfast_lookup2: times.cc $(FILES)
$(CC) -DLOOKUP2 -DSIZE=6000000 -O6 -DFAST times.cc -o tfast_lookup2

tfast_lookup3: times.cc $(FILES)
$(CC) -DLOOKUP3 -DSIZE=6000000 -O6 -DFAST times.cc -o tfast_lookup3

tfast_lookup2_no_keys: times.cc $(FILES)
$(CC) -DLOOKUP2 -DN_USE_HASH -DSIZE=6000000 -O6 -DFAST times.cc -o tfast_lookup2_no_keys

tfast_erase: times.cc $(FILES)
$(CC) -DERASE -DJTABLE_SIZE=500000 -DSIZE=6000000 -O6 -DFAST times.cc -o tfast_erase



unit_t: $(FILES)
clear
$(CC) -Wall stl_hash_map_utest.cc -o utest
```

```
utest: unit_t
echo Running tests
./utest

clean:
rm -f t tfast *.o tfast_insert tfast_lookup tfast_erase t_insert t_lookup t_erase
```