Experiments with universal hashing*

Jyrki Katajainen and Michael Lykke

Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
E-mail: `jyrki@diku.dk` and `mlykke@post3.tele.dk`

Abstract

The practical performance of randomized chain-hashing and double-hashing schemes is studied. The results of this study show that, if the keys are integers, by using multiplicative hash functions a double-hashing scheme is obtained, for which the observed (worst-case) cost of an access operation is the same as the corresponding (average-case) cost achieved by the chain-hashing scheme, which has been the fastest access method according to earlier experiments. The observed (amortized expected) cost of insert and delete operations for a dynamic double-hashing scheme is comparable to the corresponding (worst-case) cost for balanced search trees, whereas the behaviour of dynamic chain hashing is still better on an average. If the keys are strings, dynamic chain hashing with random-table hash functions outperforms clearly balanced search trees.

1 Introduction

Let S be a set of n items each consisting of a key and some information associated with that key. It is assumed that no two items of S can have the same key. A computer representation D of S is called a *dictionary* if it supports the following operations:

dictionary construct(S)	Construct a dictionary containing all the items of S; if S is empty, create an empty dictionary.
item lookup(D,x)	Return the item in D with key x; if no such item exists in D, return a **null** item.
item insert(D,x,d)	Insert a new item with key x and information d into D, and return the item created; if this operation cannot be carried out, return a **null** item.
item delete(D,x)	Delete the item with key x from D and return the item just deleted; if no such item exists, return a **null** item.

A dictionary is *static* if it only supports the operations construct and lookup, and *dynamic* if it also supports the *update* operations insert and delete.

Numerous data structures for implementing a dictionary have been proposed in the computing literature (see, e.g., [11, 13, 16]). In addition to this, ready-to-run programs can found from different libraries of data structures and algorithms (e.g., LEDA [17, 18]). For the sake of simplicity, assume

*A preliminary presentation of this work was made at the *5th DIMACS Implementation Challenge: Priority Queues, Dictionaries, and Multi-Dimensional Point Sets*, Piscataway, New Jersey, October 1996. This research was partially supported by the Danish Natural Science Research Council under contract No. 9400952 (project "Computational Algorithmics").

	Static data structure	Lookup	Construct	Extra space	Reference
a)	Sorted array	$O(\log_2 n)$	$O(n \log_2 n)$	$O(1)$	e.g. [16, Section III.3.1]
	Hash table (chaining)	$\overline{O}(1)$	$\overline{O}(n)$	$O(n)$	e.g. [16, Section III.2.4]
	Hash table (double hashing)	$O(1)$	$\overline{O}(n)$	$O(n)$	[10]

	Dynamic data structure	Lookup	Update	Extra space	Reference
b)	Balanced search trees	$O(\log_2 n)$	$O(\log_2 n)$	$O(n)$	e.g. [1, 12]
	Randomized search trees	$\tilde{O}(\log_2 n)$	$\tilde{O}(\log_2 n)$	$\tilde{O}(n)$	[20, 22, 23]
	Hash table (chaining)	$\overline{O}(1)$	$\overline{O}(1)$	$O(n)$	e.g. [16, Section III.2.4]
	Hash table (double hashing)	$O(1)$	$\overline{O}(1)$	$O(n)$	[8]

Table 1: The theoretical behaviour of various data structures implementing a) a static dictionary and b) a dynamic dictionary for n items with integer keys. Here $O(f(n))$ denotes the worst-case performance, $\overline{O}(f(n))$ the (amortized) expected performance, and $\tilde{O}(f(n))$ the expected performance guaranteed with high probability. The extra space used is counted in words.

that the keys are integers. If the size of the key universe, U, is small, the dictionary should be implemented by using a direct-access array. With this structure the lookup, insert and delete operations can all be carried out in $O(1)$ worst-case time. A structure for n items can also be constructed in $O(n)$ time even though the whole structure uses $O(n+U)$ storage locations (see, for example, [16, Section III.8.1]). If U is large, a dictionary could be implemented by using a sorted array, different kinds of trees, or a hash table. The theoretical performance of some common data structures is summarized in Table 1.

In this paper the practical performance of randomized hashing is compared to other known methods. In *deterministic hashing* a fixed *hash function* h is selected, space for a *hash table* of size m is allocated, and function h is used to map the given items to various locations in that table. The disadvantage of deterministic hashing is that there always exist data collections for which many different items are mapped to the same locations. Since many collisions may degenerate the performance of this strategy, Carter and Wegman [4] introduced a *randomized hashing* strategy, called *universal hashing*. Here one chooses randomly a function h from a universal class $\mathcal{H}$ of hash functions and then proceeds as in the deterministic case. A class $\mathcal{H}$ is said to be *c-universal* if only a fraction c/m of the functions leads to a collision for any distinct pair of keys. Several function classes are known to be c-universal for a constant c (see, e.g., [4, 6, 7, 14, 15]).

Various methods for collision handling have been proposed (more than a dozen of those are described in [11]) but the simple *chaining* method, which keeps the colliding items in a linked list, has turned out to provide the fastest operation in most situations (cf. the experimental results reported, for example, in [11, Table 3.20] and [13, pp. 538–540]). When the *load factor* n/m is kept equal to $\Theta(1)$ in chain hashing, the expected cost of a lookup operation is $O(1)$, the amortized expected cost of an update operation $O(1)$, and the expected cost of a construct operation $O(n)$ (see, e.g. [16, Section III.2.4]). These bounds are guaranteed for any input, since the randomization is done by the algorithm.

The ultimate goal in hashing is to construct a *perfect hash function*, which maps distinct items to distinct locations in the table. After the discovery of a perfect hash function, a lookup operation simply evaluates the value of the function and reports the item (if any) stored at the corresponding location in the table. Unfortunately, such functions are rare in the set of all functions and therefore difficult to discover [13, pp. 506–507]. A natural idea is to relax the definition of a perfect hash function to that of a *near-perfect hash function*, which is allowed to cause a small number of collisions at each location in the table. After finding a near-perfect hash function, the items colliding at any given location can be rehashed into a secondary hash table using a perfect hash function. Fredman et al. [10] showed that there exists a double-hashing scheme that can handle integer keys within the following bounds: a) the data structure can be constructed in $O(n)$ (randomized) expected time, b) stored in $O(n)$ space, and c) with it a lookup operation can be carried out in $O(1)$ worst-case time. A dynamic version of this static dictionary structure, described by Dietzfelbinger et al. [8], uses $O(1)$ worst-case time for lookup operations and $O(1)$ amortized expected time for update operations. Both these constructions work with any universal class of hash functions.

We programmed two randomized hashing schemes: dynamic chain hashing and dynamic double hashing. Our programs rely on two types of hash functions: multiplicative hash functions and random-table hash functions, the properties of which are recalled in Section 2. The implementation details of chain hashing are described in Section 3 and those of double hashing in Section 4. The performance of the programs were compared to the dictionary structures available in LEDA (version R 3.4). The results of our study show that in the case of integer keys by using multiplicative hash functions a double-hashing scheme is obtained, for which the observed (worst-case) cost of a lookup operation is the same as the corresponding (average-case) cost achieved by the chain-hashing scheme. The observed (amortized expected) cost of update operations for dynamic double hashing is comparable to the corresponding (worst-case) cost for balanced search trees, whereas the behaviour of chain hashing is in this respect still much better on an average. On the other hand, by using random-table hash functions string keys can be handled efficiently. Even for string keys dynamic chain hashing outperforms clearly balanced search trees. The experimental results are reported in Section 5. Finally, some concluding remarks are given in Section 6.

2 Universal hashing

In this section we define the universal classes of hash functions used in our implementations. The properties of these function classes are also discussed.

2.1 Multiplicative hash functions

Assume that the keys of the items are integers. The good features of the multiplicative hash functions were already discussed in the book by Knuth [13, p. 508 ff.]. The basic idea in the *multiplication method* is simple: the given integer key is multiplied by some magic number and the middle bits of the product are used as the hash value. Knuth suggested also how the magic multiplier should be selected to get the most uniform distribution of the hash values.

In practical experiments it was observed that most odd multipliers work well for a given set of integers. This leads to a randomized hashing strategy where the multiplier is selected randomly from among the odd numbers in the key universe. More precisely, the class $\mathcal{H}_{2^\ell,2^k}$ of *multiplicative hash functions* is defined as follows:

$$\{h_a \mid h_a : \{0,1,\ldots,2^\ell-1\}\to\{0,1,\ldots,2^k-1\}\ h_a(x) = \left\lfloor \frac{a *_{2^\ell} x}{2^{\ell-k}} \right\rfloor,\ a\in\{1,3,5,\ldots,2^\ell-1\}\},$$

where ℓ and k are positive integers, $\ell \geq k$, and $*_{2^\ell}$ denotes the multiplication modulo 2^ℓ. Dietzfelbinger et al. [7] verified that the class $\mathcal{H}_{2^\ell,2^k}$ is 2-universal.

Only two arithmetical operations are needed to evaluate the value of a hash function $h_a \in \mathcal{H}_{2^\ell,2^k}$: the multiplication modulo 2^ℓ and the right shift by $\ell-k$ bit positions. Assuming that ℓ equals to the number of bits in a word, the modulo 2^ℓ multiplication is done by the hardware in most computers. Also the shift operation is available in most computers. Moreover, to specify the particular function in use only the multiplier a and the shift value $\ell - k$ have to be stored. The C++ class given in

```
#define inclusive-or |
#define right-shift >>
#define w 32                          /*the number of bits in a word*/

class H_{2^w,2^k}{
    friend class hashtable;
    int a;                            /*the multiplier in a function*/
    int s;                            /*the shift in a function*/
    public:
    H_{2^w,2^k}();
    select(int);
    int evaluate(const int) const;};

H_{2^w,2^k}::select(int 2^k){
    a = random(0,2^w-1) inclusive-or 1;
    s = w - k;};

int H_{2^w,2^k}::evaluate(const int &x) const{
    return (a *_{2^w} x) right-shift s;};
```

Figure 1: A C++ implementation of the class $\mathcal{H}_{2^w,2^k}$ of the multiplicative hash functions.

Figure 1 implements the operations for selecting a hash function and evaluating the value of a given function.

2.2 Random-table hash functions

Consider now multi-word keys or variable-length keys which are stored in a computer by using a consecutive sequence of bytes. We interpret these keys as *character strings* having some finite maximum length. A class of hash functions suitable for handling string keys has been described, among the others, by Fox et al. [9] and Majewski et al. [14].

Let the length of a string s be ℓ, and let $s[i]$ denote the ith character of s. Furthermore, let Σ be the alphabet in which the characters are encoded. If each character of Σ is encoded by using b bits, the cardinality of Σ is 2^b. The class $\mathcal{F}_{2^{\ell b},2^k}$ of *random-table hash functions* is as follows:

$$\{f_r \mid f_r : \Sigma^\ell \rightarrow \{0,1,...,2^k-1\}\ f_r(s) = \bigoplus_{i=1}^{\ell} r[i][s[i]],\ r \text{ is a mapping table of size } \ell \times 2^b\},$$

where k is a positive integer, $\ell b \geq k$, and $\oplus$ denotes the bitwise exclusive-or operation. Instead of $\oplus$, any group operator on the set $\{0,1,...,2^k-1\}$ may be used. A member of the class $\mathcal{F}_{2^{\ell b},2^k}$ is selected by generating a table r of size $\ell \times 2^b$ such that each location in r contains a random integer drawn independently and uniformly from the range $\{0,1,...,2^k-1\}$. The C++ implementation of this hashing mechanism is shown in Figure 2.

It was pointed out by Majewski et al. [14] that this class of hash functions is 1-universal (cf. [4, Proposition 8]). That is, only a fraction $1/2^k$ of the functions leads to a collision for any distinct pair of keys. A function can be selected from this class in $O(\ell 2^b)$ time, since we assume the ability to generate random numbers in constant time. To store the table, $O(\ell 2^b)$ words are required. The evaluation of a hash value takes $O(\ell)$ time. The evaluation is fast also in practice, since basically only a table lookup is needed for each character of the given string. It was observed by Pearson

```
#define exclusive-or ^
#define b 8                          /*the number of bits in a byte*/

class $\mathcal{F}_{2^{\ell b},2^k}${
    friend class hashtable;
    int size;                        /*the number of rows in the table*/
    int* table;
    public:
    $\mathcal{F}_{2^{\ell b},2^k}$();
    select(int, int);
    int evaluate(const string) const;};

$\mathcal{F}_{2^{\ell b},2^k}$::select(int $\ell$, int $2^k$){
    size = $\ell$;
    table = new int[$\ell$][$2^b$];
    for (int i = 0;i < $\ell$;i++)
        for (int j = 0;j < $2^b$;j++)
            table[i][j] = random(0,$2^k$-1);};

$\mathcal{F}_{2^{\ell b},2^k}$::evaluate(const string &s) const{
    int value = int j = 0;
    for (int i = 0;i < s.length();i++,j++){
        if (j == size) j = 0;
        value = value exclusive-or table[j][(int) s[i]];};
    return value;};
```

Figure 2: A C++ implementation of the class $\mathcal{F}_{2^{\ell b},2^k}$ of random-table hash functions.

[19] that this hashing scheme works satisfactorily even if the same table is used for each character position. Even though this is consistent with our observations, we recommend a bit larger table which is then used cyclically. Our experiments indicate that these randomized hash functions are as fast and reliable as the best deterministic hash functions used in real-world program applications, e.g., those mentioned in [3].

3 Chain hashing

The basic idea in chain hashing is simple. The colliding items are linked together in a linked list. For example, in the book by Gonnet and Baeza-Yates [11] two different implementations of this method were presented. In *hashing with direct chaining* each location in the hash table contains a pointer to the beginning of the list of colliding items, whereas in *hashing with separate chaining* each location consists of a node containing a pointer to an item and a pointer to another node. In practical experiments the observed performance of hashing with separate chaining has turned out to be a bit better than that with direct chaining (see, [11, Table 3.20] or [13, Figure 44]). Yet another alternative, called here *hashing with circular chaining*, is otherwise as one of the above methods but the linked list is maintained as a circular list.

Our actual implementation is a combination of separate chaining and circular chaining. We let initially each location in the hash table contain a single node which has an item pointer pointing to a *sentinel* and a node pointer pointing to itself. The use of a sentinel gives a tight inner loop for traversing the list; only one test is required to control when the traversal should be stopped.

```
#define null 0
#define w 32                          /*the number of bits in a word*/
#define b 8                           /*the number of bits in a byte*/

class item{
    public:
    keytype key;
    infotype info;};

class node{
    friend class hashtable;
    item* ptr;
    node* succ;};

class hashtable{
    int n;                            /*the number of items*/
    int $\ell$ = 16;                  /*the size of keys in bytes*/
    int $2^k$;                        /*the size of the table*/
    int minsize;                      /*the minimum size of the table*/
    node* table;
    $\mathcal{H}_{2^w,2^k}$ h();
    $\mathcal{F}_{2^{\ell b},2^k}$ f();
    static item sentinel;
    public:
    hashtable(int);                   /*construct*/
    item* lookup(keytype) const;
    item* insert(keytype, infotype);
    item* delete(keytype);};

hashtable::hashtable(int $2^j$){
    minsize = $2^k$ = $2^j$;
    table = new node[$2^j$];
    if (int_type(keytype)) h.select($2^j$);
    else f.select($\ell$,$2^j$);
    for (int i = 0;i < $2^j$;i++){
        table[i].ptr = &sentinel;
        table[i].succ = &table[i];};};

item* hashtable::lookup(keytype x) const{
    node* p;
    sentinel.key = x;
    if (int_type(keytype)) p = &table[h.evaluate(x)];
    else p = &table[f.evaluate(x)];
    do p = p→succ;
    while (p→ptr→key ≠ x);
    if (p→ptr == &sentinel)
        return null;
    return p→ptr;};
```

Figure 3: Part of a C++ class implementing a dictionary with chain hashing.

In other chaining methods two tests are required: the first testing whether the end of the list is reached and the other testing whether the searched key equals to the stored key (cf. the programs given in [11]). A C++ function for the lookup operation is shown in Figure 3.

The insert operation starts by checking whether the key already exists in the data structure. If this is the case, only the information associated with this key is updated. Otherwise a new node is added to the corresponding circular list and the pointer values are updated as required. In both cases the just inserted item is returned. The delete operation works similarly. If the key exists, the corresponding node is removed from the circular list and the deleted item is returned as the outcome of the operation; otherwise a **null** item is returned.

In the static case when the number of items, n, is known beforehand the size of the hash table, m, is fixed to the smallest power of 2 larger than or equal to n. With a c-universal class of hash functions the expected length of a chain is $O(c)$ (see, e.g., [16, Section III.2.4]). Hence, the cost of a construct operation is proportional to $c\ell n$, in which ℓ denotes the size of the keys in bytes.

There are no problems when carrying out insertions and deletions in this structure as far as the *load factor* n/m is within some bounds, e.g., $1/2 < n/m < 2$. To adapt the data structure for dramatic changes in the cardinality of the items, the standard doubling technique is used. Each time the load factor reaches the value 2, the size of the hash table is doubled, all the items are rehashed into this new table, and the space allocated by the old table is freed. On the other hand, when the load factor reaches the value 1/2, the size of the hash table is halved after which a reconstruction is done as above. The size of the hash table should, however, always be at least, say 1024. This is just the way chain hashing is dynamized in LEDA (version R 3.4).

In the reconstructions it is known that all the items in the old table have distinct keys. Therefore, every item can be appended to the corresponding chain without checking whether the item appears in the structure or not. That is, a reconstruction requires $O(\ell n)$ time in the worst case. When the reconstruction time is amortized over the insert and delete operations carried out after the previous reconstruction, the amortized expected cost of a lookup, insert or delete operation is still proportional to c times the size of the keys.

This data structure uses $2m+2n$ storage locations for pointers, that is the extra space required is never more than $6n+O(1)$ words. An advantage with this structure is that it supports fast lookup, but one should observe that it is more space-consuming than both direct chaining and separate chaining. One could also save some space by storing the items inside the list nodes. However, the present solution makes the moves of the items unnecessary, which again may save some time especially if the items are big. This means also that the item references are persistent, i.e., the value returned by any operation remains valid throughout the lifetime of the item.

4 Double hashing

In the double-hashing scheme introduced by Fredman et al. [10] the colliding items are not chained but hashed again by using another hash function. We say that the colliding items form a *bin*. The size of the *primary hash table* is fixed to the smallest power of 2 larger than τn, in which τ is some positive constant and n the number of items to be stored in the structure. Each bin stores a pointer to a hash function and the start address of a *secondary hash table*. Each location in the secondary table contains a pointer to an item. If a bin is empty, it has a pointer to the *dummy* table built above the sentinel.

In a lookup operation the values of the two hash functions are calculated and the pointers followed to an item. If this particular item is the sentinel or if the key of the item is not equal to the given key, a **null** item is returned; otherwise the found item is returned. Figure 4 describes

```
#define null 0
#define w 32                          /*the number of bits in a word*/

class item{
    public:
    int key;
    infotype info;};

class bin{
    friend class hashtable;
    H_{2^w,2^k} h();
    item* 2-table;
    bin();};

bin::bin(){
    h.select(0);
    2-table = &sentinel;};

class hashtable{
    friend class bin;
    int m;                            /*the number of memory locations used*/
    int n;                            /*the number of items*/
    int 2^k;                          /*the size of the primary table*/
    H_{2^w,2^k} h();
    bin* 1-table;
    static item sentinel;
    static bin dummy;
    public:
    hashtable(int);                   /*construct*/
    item* lookup(int) const;
    item* insert(int, infotype);
    item* delete(int);};

item* hashtable::lookup(int x) const{
    bin* p = &1-table[h.evaluate(x)];
    item* q = p→2-table[p → h.evaluate(x)];
    if (q = &sentinel)
        return null;
    if (q→key = x)
        return q;
    return null;};
```

Figure 4: Part of a C++ class implementing a dictionary with double hashing.

the implementation of the lookup operation for integer keys in detail. Of course, in the actual implementation the function evaluate() is declared **inline**.

Let n_i denote the size of the ith bin and 2^k the number of bins, where $2^{k-1} < \tau n \leq 2^k$. To construct the tables efficiently, randomization is used. Assume that the class of hash functions employed is c-universal. At the first level the expected number of collisions between all possible pairs of items is at most $\binom{n}{2}\frac{c}{2^k}$. Another way of expressing this is that the expected value of $\sum_{i=0}^{2^k-1}\binom{n_i}{2}$ is less than $\binom{n}{2}\frac{c}{2^k}$. From Markov's inequality it follows that, for any $\alpha > 1$, $\sum_{i=0}^{2^k-1}\binom{n_i}{2} < \alpha\binom{n}{2}\frac{c}{2^k}$ with probability larger than $1 - 1/\alpha$. By using this fact, we can repeatly select a random hash function from the universal class until one is found which gives less than $\alpha\binom{n}{2}\frac{c}{2^k}$ collisions. The average number of rounds needed here is at most $\frac{1}{1-1/\alpha}$. At the second level a similar method is applied. If a non-empty bin contains n_i items, we reserve a secondary hash table of size 2^{k_i} for this bin, 2^{k_i} being the smallest power of 2 larger than or equal to $\max(1, \beta c\binom{n_i}{2})$, for some $\beta > 1$. Hence, the probability that a random secondary function is injective is larger than $1 - 1/\beta$. On an average, at most $\frac{1}{1-1/\beta}$ rounds are needed before an injective secondary function is found for a bin. The expected time used for the construction of the hash tables is thus proportional to the size of the input.

In our implementation we fixed the parameters τ, α, and β as follows: $\tau = 1$, $\alpha = 3/2$, and $\beta = 4/3$. Let us now analyse the storage requirements of this particular implementation. The primary table uses $2 \cdot 2^k$ storage locations. Since at most n of the 2^k secondary tables can be in use and since $\sum_{i=0}^{2^k-1}\binom{n_i}{2} < \frac{3}{2}\binom{n}{2}\frac{c}{2^k}$, the number of pointers used by the secondary hash tables is at most $n + \frac{2c^2n^2}{2^k}$. If we let $s(n)$ denote an upper bound for the space requirements of the hash tables, then $s(n) = 2\epsilon n + n + \frac{2c^2n}{\epsilon}$, where $1 \leq \epsilon < 2$. The maximum value of the function $t(\epsilon) = 2\epsilon + \frac{2c^2}{\epsilon}$ is reached at $\epsilon = 1$, $\epsilon = c$, or $\epsilon = 2$. Thus, the space usage of the hash tables is at most $6n$ and $11n$ for $c = 1$ and $c = 2$, respectively. Furthermore, some space is needed for storing the hash functions but a lot of space can be saved here by using the same hash functions for several secondary tables as proposed by Schnieder [21]. For further tricks of saving space, we refer to [10] and [21].

This hashing scheme can be dynamized by using the standard doubling technique as shown by Dietzfelbinger et al. [8]. Basically, we have followed their guidelines in our implementation, but we permit insertions and deletions as far as no local or global reconstructions are necessary. The idea is to reserve so much storage for the hash tables that γn_0 update operations, $0 < \gamma < 1$, can be accomplished without much difficulty if the structure contained n_0 items just prior to the previous global reconstruction.

In a *global reconstruction* a structure is created as described above with the parameters τ, α, and β, to be specified later. To carry out a *local reconstruction*, a bin should also store the size of the secondary hash table. However, when multiplicative hash functions are employed, the multiplier and and the shift value are stored within a bin, not the size of the secondary table since it can be deduced from the shift value (cf. Figure 4). When the size of the secondary table is known, the items in the corresponding bin can be gathered together and rehashed as required. The update operations can trigger a local or global reconstruction as follows:

1. A delete operation is carried out weakly by letting the item pointer in the secondary table to point to the sentinel. If the *space limit* is exceeded just after a delete operation, then a global reconstruction is carried out.

2. An insert operation is performed by adding a pointer into the secondary hash table pointing to the new item, provided that the item does not collide with another item. If the item to be inserted causes a collision in the secondary table, three cases are possible: a) if the

corresponding secondary table is large enough, then a new injective function is searched for that; b) if the current table is too small and a larger table does not cause the structure to exceed the space limit, a larger table is allocated and a new injective function is searched for that; c) otherwise a global reconstruction is necessary.

We set the space limit to $\frac{n}{1-\gamma}(1+\gamma+\alpha\beta c^2(1+\gamma)^2/\tau+6\tau)$. In our implementation we fixed the parameters α, β, and γ as follows: $\alpha = 3/2$, $\beta = 4/3$, $\gamma = 1/2$. For $c = 1$, the space limit is $24n$ at $\tau = 1$ and, for $c = 2$, $45n$ at $\tau = 2$. These bounds can be improved only marginally by other values of τ.

The choice of β guarantees that, if a secondary hash table for a bin reaches the size 2^{k_i}, the expected time used in the construction of all the tables for this bin up to the size 2^{k_i} is proportional to 2^{k_i}. Especially, observe that on an average $O(1)$ tables of size 2^i have to be built, for all $i \in \{0, \ldots, k_i\}$. Since on an average $O(1)$ rounds are necessary before a primary hash table is found, for which $\sum_{i=0}^{2^k-1} 2^{k_i} \in O(n_0)$, the overall cost for handling the γn_0 updates is proportional to n_0 times the size of the keys. Sometimes more than γn_0 updates can be handled before the next global reconstruction is necessary; these updates can be considered to be free of cost.

5 Experimental results

We programmed both the chain-hashing and double-hashing schemes described in the previous sections. In order to compare their performance to that of the methods available in LEDA (version R 3.4), we programmed them so that they could be used as LEDAs "user defined data structures" [18, Chapter 13.2].

All the experiments reported in this section were carried out on a personal computer which has an Intel Pentium 120 MHz processor and runs under LINUX 2.0.0. The size of its main memory is 32 MB and it has a virtual address space of size 48 MB. The programs were written in C++ and compiled with GNU's gcc compiler (version 2.7.2) by using the optimization flag "-O". In the generation of random data and the measurements of time we relied on the tools available in LEDA. In all the running times reported the overhead caused by the driver program is excluded.

In our first set of experiments the data structures available in LEDA were tested. Our observations were the following:

1. The sorting and binary-search routines were extremely slow compared to the corresponding running times obtained by using balanced search trees. The main reason for this was that an item class defining a linear order had to be defined which was not the case for tree structures when the key was known to be an integer.

2. In general, AVL-trees [1] turned out to be superior to red-black trees [12].

3. As to randomized search trees, skip lists [20] seemed to perform better than treaps [22].

Therefore, only AVL-trees and skip lists were included in the later experiments.

A dynamic chain-hashing scheme is also available in LEDA. The current implementation is deterministic. If the hash table is of size 2^k, the integer consisting of the k last bits of a key is used as the hash value. This is obtained simply by anding the key with $2^k - 1$. The colliding items are kept in a linked list where the last node contains a pointer to a sentinel. This gives as well a tight inner loop when traversing the list. The dynamic double-hashing scheme available in LEDA has been programmed by Wenzel (as cited in [8]). This implementation uses divisional hash functions as proposed both in [8] and [10].

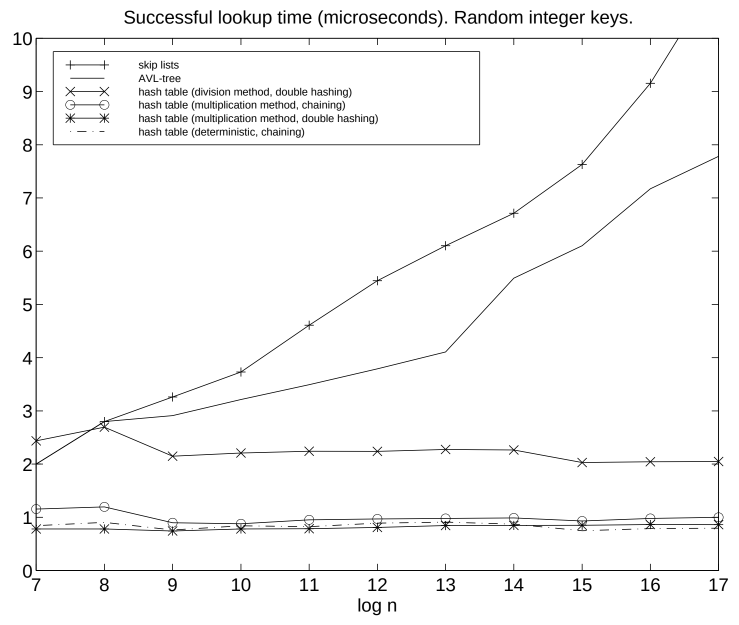

Figure 5: Average time for a successful lookup when the size of the dictionary was n. Four hashing schemes and two search-tree schemes were included in this test.

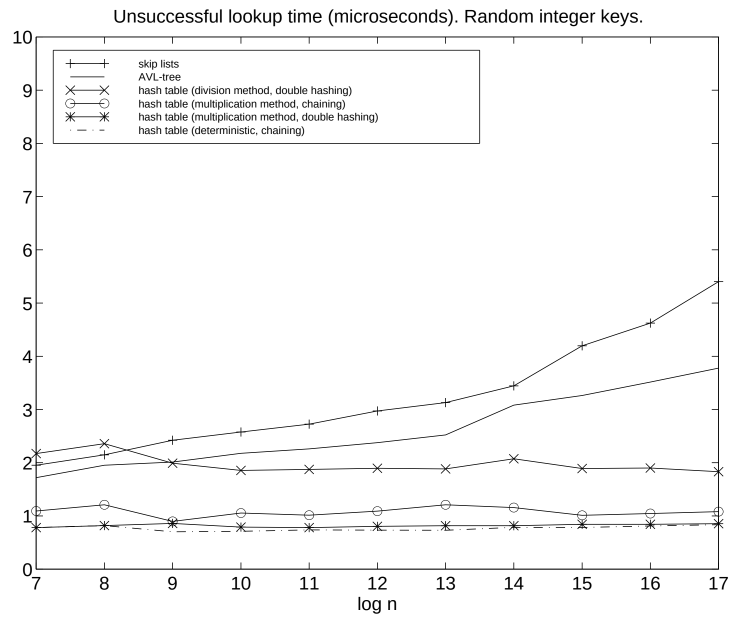

Figure 6: Average time for an unsuccessful lookup when the size of the dictionary was n. Four hashing schemes and two search-tree schemes were included in this test.

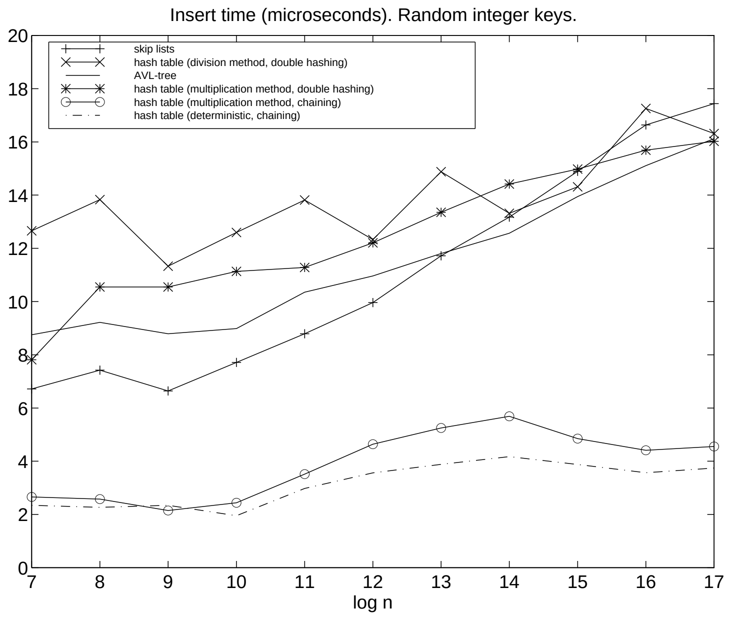

Figure 7: Average insertion time for various methods when constructing a dictionary of size n.

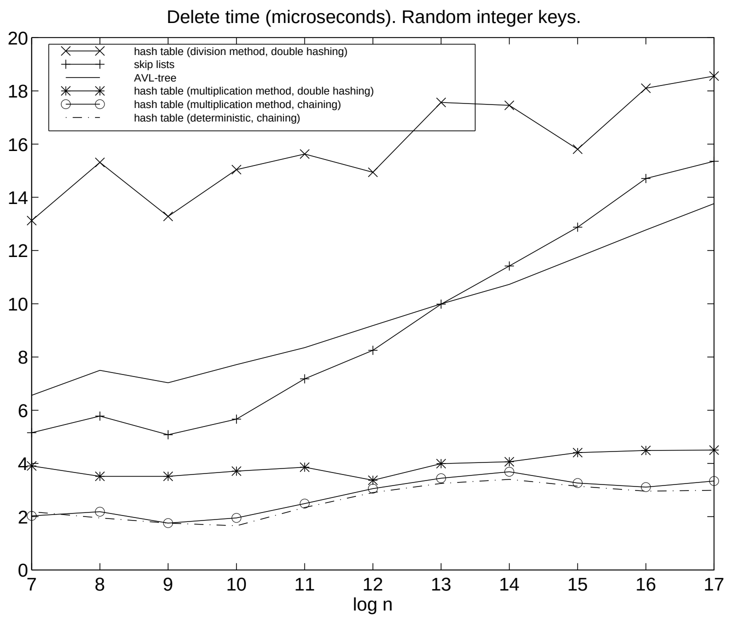

Figure 8: Average deletion time for various methods when a dictionary of size n was emptied.

In our second set of experiments we wanted to compare the average performance of the methods mentioned. Therefore, the input for the programs were generated randomly. The size of the inputs was varied from 2^7 to 2^{17} and for each input size 10 different data collections were used. When testing the lookup time for successful searches, each key was accessed at least 5 times. Figure 5 reports the average times for a successful lookup. Figure 6 gives the corresponding times for an unsuccessful lookup. Random data collections were also used to calculate the average insertion and deletion times. Figure 7 reports the average cost of an insert operation during the construction of the data structures. Figure 8 reports the corresponding cost of a delete operation when the data structures were again emptied.

According to Figure 5 the observed lookup times are about the same for deterministic chain hashing, multiplicative chain hashing, and multiplicative double hashing; tree based methods are clearly slower. The behaviour of the hashing methods does not change even if the key searched is not in the structure. However, the tree methods can carry out an unsuccessful lookup about twice as fast as a successful one. As to insertions and deletions, it can be seen from Figure 7 and 8 that the average performance of the chain-hashing methods is very good, whereas the performance of the double-hashing methods is comparable to that of the tree methods. However, multiplicative double hashing seems to perform better than divisional double hashing, especially in deletions.

In our third set of experiments we tried to evaluate the reliability of chain hashing. First, we generated input data which forced the deterministic method to produce lists of length 8. However, the lookup times for successful searches were still about the same as for the random data. Second, we generated data where the length of every non-empty list was equal to 2^{10}. After this the average cost of an unsuccessful search increased from 1 microsecond to 92 microseconds. If $n = 2^{17}$, the corresponding AVL-tree used 4 microseconds to handle the same search. Randomized chain hashing worked still very well. This makes us to conclude that catastrophic lookup times are rare when multiplicative chain hashing is used.

In the fourth set of experiments the performance of a chained hash table was compared to that of an AVL-tree when the keys of the input items were strings. For this purpose we used the testbed problems available from DIMACS [5]. The observed execution times are reported in Tables 2, 3, and 4. The reported times are average values when the same testbed problems were solved 10 times. In chain hashing the size of the random table needed by the hash function had a fixed size of 16. The current AVL-tree implementation in LEDA does not utilize the fingerprints but uses the string compare function when traversing the tree. As seen a chained hash table is clearly superior to an AVL-tree.

6 Discussion

Already Carter and Wegman [4] pointed out in their seminal paper that there is no time penalty associated with using universal hashing. The present study meets this view. As to the collision handling, our study shows that it is difficult to beat dynamic chain hashing. In double hashing lookup operations are fast but this is achieved at the expense of more costly update operations and larger memory requirements. It is also in place to correct the old myth about the size of the hash tables (see, e.g., [3]): the table size should *not* be a prime number but a power of 2 to make the dynamization easy.

In chain hashing the expected length of the longest chain is at most $\sqrt{2cn} + 1$, assuming that the load factor is less than 2 and a c-universal class of hash functions is employed. If the hash function is selected as carefully as the primary hash function in double hashing, (with $\alpha = 2$) the upper bound $2\sqrt{cn} + 1$ is guaranteed in the worst case. The most problematic chains are those

Random numeric key tests (milliseconds). String keys of length 16.								
Input	Hash table	AVL-tree	Input	Hash table	AVL-tree	Input	Hash table	AVL-tree
id.a1	15	32	id.a2	15	38	id.a3	35	35
id.b1	243	476	id.b2	243	464	id.b3	258	470
id.c1	2 410	8 020	id.c2	2 044	7 435	id.c3	2 335	7 805
iid.a1	27	63	iid.a2	26	63	iid.a3	27	60
iid.b1	447	794	iid.b2	462	816	iid.b3	467	809
iid.c1	6 550	21 320	iid.c2	–	–	iid.c3	–	–
iisd.a1	37	81	iisd.a2	34	75	iisd.a3	34	73
iisd.b1	521	1 071	iisd.b2	533	1 059	iisd.b3	527	1 059
iisd.c1	6 550	21 320	iisd.c2	5 800	19 820	iisd.c3	4 900	20 490
iiud.a1	33	77	iiud.a2	33	79	iiud.a3	33	82
iiud.b1	552	1 043	iiud.b2	538	1 044	iiud.b3	541	1 063
iiud.c1	5 100	20 320	iiud.c2	6 260	21 670	iiud.c3	5 520	20 410
is.a1	15	31	is.a2	11	37	is.a3	18	34
is.b1	243	502	is.b2	238	501	is.b3	247	507
is.c1	2 390	8 040	is.c2	2 500	7 985	is.c3	2 400	8 355
iu.a1	14	33	iu.a2	16	36	iu.a3	16	46
iu.b1	254	543	iu.b2	258	481	iu.b3	259	493
iu.c1	2 390	7 845	iu.c2	2 540	8 060	iu.c3	3 065	8 140

Table 2: Execution times for a chained hash table and an AVL-tree when the input was generated by the DIMACS test generator: dc_random. The problems with "–" could not be solved due to the lack of memory space.

Generic English key tests (milliseconds). String keys of length at most 20.					
Input	Hash table	AVL-tree	Input	Hash table	AVL-tree
ed.id.tr	12	31	jy.id.tr	14	32
ed.iid.tr	22	59	jy.iid.tr	19	64
ed.iisd.tr	29	74	jy.iisd.tr	26	76
ed.iiud.tr	29	74	jy.iiud.tr	28	74
ed.is.tr	11	35	jy.is.tr	12	36
ed.iu.tr	14	35	jy.iu.tr	10	35

Table 3: Execution times for a chained hash table and an AVL-tree when the input was generated by the DIMACS test generator: dc_generic.

Library call numbers (milliseconds). String keys of length 36.						
n	1 000		10 000		100 000	
Input	Hash table	AVL-tree	Hash table	AVL-tree	Hash table	AVL-tree
circ1	17	17	320	533	3 415	8 810
circ2	19	36	334	534	3 285	8 540
circ3	19	40	320	547	3 365	8 475
circ4	21	37	341	543	3 175	8 400
circ5	19	38	324	541	3 430	8 345

Table 4: Chained hash table vs. AVL-tree when solving the circ testbed problems available from DIMACS.

that are longer than say, 2^7. Universal hashing makes these chains rare but they can still appear. If the reliability is an absolute requirement in the application in question, the dictionary should be implemented by other means.

The best practical solution is apparently a hybrid of the methods studied. One possibility is to combine chain hashing with double hashing, e.g., by implementing all small bins as linked lists and all large bins as in double hashing. Only one extra **if**-test has to be added to the lookup function but this hybrid is more space-efficient than pure double hashing. A variant of this idea was already used in Schnieder's and Wenzel's double-hashing implementations. In static double hashing the tradeoff between the speed of the lookup operation and the usage of space was analysed carefully by Schnieder [21]. It is natural to ask whether there exists other methods that could be used to reduce the space requirements of the dynamic structure without sacrifying much of its speed.

Another possibility is to maintain each large bin as a tree. With this structure the lookup operations can be performed in constant average-case time and logarithmic worst-case time. If the bins are implemented as AVL-trees, for a bin of size n_i the height of the corresponding tree is at most $1.4404 \log_2(n_i + 2)$ [13, Section 6.2.3]. A careful implementation of the lookup operation guarantees that the number of key comparisons carried out is never more than the height of the tree plus one [2]. Hence, the tree implementation is always superior to the chain implementation as far as key comparisons are concerned. However, during a tree traversal at every tree level it must also be tested, whether an external node is reached or not, and some extra pointer assignments are necessary compared to a chain traversal. Therefore, n_i has to be quite large before the use of a tree pays off. By using standard techniques this structure can be dynamized but the practical value of this dynamic structure is unclear.

Still another possibility is to use hashing in the connection with balanced search trees. That is, the tree structure could be organized according to the hash values of the keys and only in the case of collisions the whole keys are applied as normally. For instance, when an AVL-tree stores n items with string keys of length ℓ, hashing reduces the worst-case complexity of a lookup operation from $O(\ell \log_2 n)$ to $O(\ell + \ell \log_2 k + \log_2 n)$, where k denotes the number of keys whose hash value is equal to that of the searched key.

Finally, it should be pointed out that the dynamic double-hashing method can lead to memory fragmentation since it might not be possible to reuse the space reserved earlier by the secondary hash tables that have become too small. A similar memory-allocation problem is encountered in the implementation of skip lists. When we carried out our experiments, the test runs were interrupted several times because of problems in memory allocation. We got two types of error messages: "LEDA memory allocation: out of memory" and "Segmentation fault (core dumped)". This memory-allocation problem should be taken seriously by application programmers when selecting their dictionary structure.

Acknowledgement

The authors would like to thank Martin Dietzfelbinger for his valuable advice.

References

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis, An algorithm for the organization of information, *Soviet Mathematics* **3** (1962) 1259–1263.

[2] A. Andersson, A note on searching in a binary search tree, *Software—Practice and Experience* **21** (1991) 1125–1128.

[3] A. Binstock, Hashing rehashed, *Dr. Dobb's Journal* **21**,4 (1996) 24–33.

[4] J. L. Carter and M. N. Wegman, Universal classes of hash functions, *Journal of Computer and System Sciences* **18** (1979) 143–154.

[5] Center for Discrete Mathematics & Theoretical Computer Science, *The 5th DIMACS challenge—Dictionary tests*. See the links in `http://cs.amherst.edu/~ccm/challenge5/dicto/index.html`.

[6] M. Dietzfelbinger, Universal hashing and k-wise independent random variables via integer arithmetic without primes, in *Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science **1046**, Springer (1996) 569–580.

[7] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen, A reliable randomized algorithm for the closest-pair problem, *Journal of Algorithms*, to appear. Also as DIKU Report **93/25**, Department of Computer Science, University of Copenhagen (1993).

[8] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, Dynamic perfect hashing: Upper and lower bounds, *SIAM Journal on Computing* **23** (1994) 738–761.

[9] E. A. Fox, L. S. Heath, Q. F. Chen, and A. M. Daoud, Practical minimal perfect hash functions for large databases, *Communications of the ACM* **35** (1992) 105–121.

[10] M. L. Fredman, J. Komlós, and E. Szemerédi, Storing a sparse table with $O(1)$ worst case access time, *Journal of the ACM* **31** (1984) 538–544.

[11] G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures in Pascal and C*, Addison-Wesley Publishing Company (1991).

[12] L. J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees, in *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press (1978) 8–21.

[13] D. E. Knuth, *The Art of Computer Programming*, Volume 3/ *Sorting and Searching*, Addison-Wesley Publishing Company (1973).

[14] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech, A family of perfect hashing methods, *The Computer Journal*, to appear. A preprint available from `http://dimacs.rutgers.edu/~havas/pubs.html`.

[15] Y. Mansour, N. Nisan, and P. Tiwari, The computational complexity of universal hashing, in *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, ACM Press (1990) 235–243.

[16] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag (1984).

[17] K. Mehlhorn and S. Näher, LEDA: A platform for combinatorial and geometric computing, *Communications of the ACM* **38** (1995) 96–102.

[18] K. Mehlhorn, S. Näher, and C. Uhrig, *The LEDA User Manual, Version R 3.4*, Max-Planck-Institut für Informatik (1996). Available from `http://www.mpi-sb.mpg.de/LEDA/www/papers.html`.

[19] P. K. Pearson, Fast hashing of variable-length text strings, *Communications of the ACM* **33** (1990) 677–680.

[20] W. Pugh, Skip lists: A probabilistic alternative to balanced trees, *Communications of the ACM* **33** (1990) 668–676.

[21] T. Schnieder, Perfektes Hashing statischer Mengen, *Informatik Forschung und Entwicklung* **10** (1995) 82–90.

[22] R. Seidel and C. R. Aragon, Randomized search trees, *Algorithmica* **16** (1996) 464–497.

[23] S. Sen, Some observations on skip-lists, *Information Processing Letters* **39** (1991) 173–176.