CS 225 – Data Structures

ZJUI – Spring 2022

Lecture 6: Hashing

Klaus-Dieter Schewe

ZJU–UIUC Institute, Zhejiang University

International Campus, Haining, UIUC Building, B404

email: kd.schewe@intl.zju.edu.cn

6 Hashing

We now look at another class of data structures realising an ADT $SET(T)$ of **finite sets** (with elements in T)

We spare the definition of this ADT, as predicates such as $\in$, $=$ and $\subseteq$ as well as operations such as $\cup$, $\cap$, $-$, $\times$ are standard and commonly known

However, we will not impose an order on the data structure

Our investigation applies analogously to an ADT $MSET(T)$ of **finite multisets** (with elements in T)—the key difference is that here elements may appear multiple times (we use the notion **multiplicity**)

The key question is how the common set (and multiset) operations can be implemented efficiently without order

The key technique will be **hashing**, by means of which we implement so called **associative arrays**

Example

Take a large library, and let S be a set of reserved books, i.e. S is comparably small to the set of all books and even much smaller than the set T of all book reservations since the library is in operation

In the physical world there may be just several shelves onto which the reserved books are placed

The shelves can be labelled by the last two digits of the library user cards so that every user can easily find the books they reserved

In other words, we provide a way to map the reservation (defined by book and library user) to the address of the shelf

The shelf address may not be unique, but the number of items placed on the same shelf can be assumed to be bound by some constant—in this way it also generalises easily to multisets

6.1 Associative Arrays

Let us abstract from the previous example

Definition. An ***associative array*** comprises a finite set $S \subseteq T$ and an injective function $key : S \rightarrow \mathcal{K}$, where $\mathcal{K}$ is the set of keys

The most important operations on such an associative array are:

insertion. $insert(S, e)$ with $e \in T$ is $S := S \cup \{e\}$

deletion. $delete(S, k)$ with $k \in \mathcal{K}$ is $S := S - \{e\}$, if $e \in S$ is the unique element with $key(e) = k$, and is undefined, if no such $e \in S$ exists

retrieval. $find(S, k)$ with $k \in \mathcal{K}$ returns the unique element $e \in S$ with $key(e) = k$, and is undefined, if no such $e \in S$ exists

Remarks

The terminology "associative array" is motivated by considering the elements $e \in S$ as elements of an array of length $|S|$

Then the retrieval operation $find(S, k)$ can be seen as a random access operator, and the function key "computes" the index in the representing array for S

The set $\mathcal{K}$ is the set of potential array indices, though only a small subset may actually be needed

In our small example S is the set of book reservations, and $\mathcal{K}$ is the set of library card numbers

We choose the last two digits, as it is likely that library cards issued at a similar time (and consequently used at similar times) have the same leading digits (e.g. the year of issue), but differ at the end

Hashing

The idea of a **hash table** to implement an associative array consists in providing a function $h : \mathcal{K} \to \{0, \dots, m-1\}$ mapping keys to small natural numbers

The function h is called a ***hash function***—we simple write $h(e)$ instead of $h(key(e))$ for the **hash value** of $e \in S$

In the library example we used the small numbers given by the last two digits of the library card number

In case $m = |S|$ we might expect that all values $h(e)$ are different, in which case the three basic operations on associative arrays (insert, delete, find) could be executed in constant time $\Theta(1)$

However, in general this is neither possible nor necessary; if there is a fixed bound on the number of elements with the same hash value, time complexity remains in $\Theta(1)$

Variations

If hash values $h(e)$ are pairwise different, we can simply use a table t (represented by an array) with $t(h(e)) = e$

In the library example this is not the case: several reserved books can go to the same shelf, which has to be searched

A generalisation leads to the idea of ***hashing with chaining***: use set-valued entries in the hash table that are represented by a singly linked list

Universal hashing allows us to construct hash functions with probabilistic performance guarantees

Hashing with linear probing builds on the idea to store values $e \in S$ directly in an array indexed by the hash values, but permits to use different entries in case a field is already occupied

6.2 Hashing with Linear Probing

We will first look into an implementation of ***hashing with linear probing***

Clearly, we can represent a hashtable by an array A of length m with three different types of entries:

- $A[i] = e \in S$ with $h(e) \leq i$
- $A[i]$ can be undefined, so there is no $e \in S$ with $h(e) = i$
- $A[i]$ may contain a **placeholder**—this occurs, when a value $e \in S$ stored in $A[i]$ has been deleted, but there may still be elements e' in the hash table with $h(e') = h(e)$

In C++ we do not store values $e \in T$ but rather pointers to such values; then a specific fixed pointer can be used to represent a placeholder, and a null pointer can be used to represent undefinedness

Hash Table Operations

Just consider three basic operations

- **add** inserts a new element $e \in S$ into the hash table:
    - First determine $h(e)$, then conduct a linear search in the array starting from $A[h(e)]$ to find the first free entry $A[i]$ (the stored value is either a placeholder or a null pointer)
    - The search can only terminate, when a null pointer has been found—to ensure that $e \in S$ does not yet occur in the hash table

Hash Table Operations / cont.

- **remove** deletes an element $e \in S$ from the hash table (provided it occurs in there):

  - As for the add operation conduct a linear search srtarting from $A[h(e)]$ to find i with $A[i] = e$

    If the found entry is the last one in a sequence, i.e. the next entry contains a null pointer, deletion frees the space using a null pointer, otherwise a placeholder is stored

  - As for the add operation the search can only terminate, when such an antry or a null pointer has been found

- **member** checks, if an element $e \in S$ occurs in the hash table:

  - The search is analogous to the other operations, but the hash table is not updated; only a truth value is returned

Rehashing

If the array is filled more than a **max fill factor** (default: 0.75) permits or less than a **min fill factor** (default: 0.25) permits, the hash table is reorganised

A larger/smaller hash table is created with roughly double/half the size of the previous one

All entries of the existing hash table are copied to the new one

However, hash values $h(e)$ for the new hash table are different, so each copying requires a computation of the new hash value and an insertionusing the add operation

After rehashing there are initially no placeholders in the hash table

6.3 Hashing with Chaining

As we cannot guarantee that hash values are always pairwise different, we have to explore methods to handle cases, where elements have the same hash value

Hashing with chaining follows the simple idea to use a hash table (represented by an array A), where the entries $A(h(e))$ are lists

In this case the elementary operations *insert*, *delete* and *find* are easily realised:

- To insert an element e determine its hash value $h(e)$, and insert e somewhere in the list in $A(h(e))$
- To delete an element with key k we have to scan the list in $A(h(k))$; if an element e with $key(e) = k$ is found, it is removed from the list
- To find an element with key k we also have to scan the list in $A(h(k))$; if an element e with $key(e) = k$ is found, it is returned—otherwise, an error is raised

Complexity

Clearly, insertions take constant time

However, for delete and find operations in the worst case the complete list $A(h(k))$ has to be scanned, which is linear in the length of this list

As hash functions map keys in $\mathcal{K}$ to a small set $\{0, \dots, m-1\}$, there is always a set of N/m keys (for $N = |\mathcal{K}|$) that are mapped to the same value

This, in the worst case all elements in a set S have the same hash value, and the time complexity of *delete* and *find* will be in $\Theta(n)$

We will therefore explore the average case complexity

Avergage Case Complexity

Theorem. If n elements are stored in a hash table with m entries and a random hash function is used, the expected execution time of a *find* or *delete* operation is in $O(1 + n/m)$.

Note. The hash function is completely random, if it is any function $h : \mathcal{K} \rightarrow \{0, \dots, m-1\}$

As there is no way to provide a descriptive way for all such hash functions, the only possbility is to use the set H of all such functions

However, there are m^N such hash functions, each requiring $N \log m$ bits for its representation

This implies that the assumption to have a completely random hash function is de facto completely unrealistic

Proof

The execution time for *find* or *delete* is determined by the length of the list in $A(h(k))$ plus some constant, i.e. it is in $O(1 + E(X))$, where X is a random variable mapping to this length

If S is the set of elements stored in the hash table, i.e. $|S| = n$, we can write $X = \sum_{e \in S} X_e$ with Bernoulli-distributed random variables $X_e = \begin{cases} 1 & \text{if } h(e) = h(k) \\ 0 & \text{else} \end{cases}$

Then we get $E(X) = E\left(\sum_{e \in S} X_e\right) = \sum_{e \in S} E(X_e) = \sum_{e \in S} P(h(e) = h(k))$

As h is a random hash function, it maps e to each $h(e) \in \{0, \dots, m-1\}$ with the same probability, independent of $h(k)$, i.e. $P(h(e) = h(k)) = \frac{1}{m}$ and $E(X) = \frac{n}{m}$, which completes the proof

6.4 Universal Hashing

The average complexity for hashing with chaining (in the previous theorem) would become much more meaningful, if the prerequisite that "*a random hash function is used*" could be weakened to **hash functions randomly selected from a small class of hash functions**

We are looking for functions that could be specified using constant storage, e.g. if the function is given by an explicit formula, and which are easy to evaluate

Definition. Let $c > 0$. A class H of functions $\mathcal{K} \to \{0, \dots, m-1\}$ is called ***c-universal*** iff any two keys $x, y \in \mathcal{K}$ collide with a probability at most c/m, i.e.

$$|\{h \in H \mid h(x) = h(y)\}| \le \frac{c}{m}|H|$$

or equivalently

$$P(h(x) = h(y)) \le \frac{c}{m} .$$

Average Case Complexity in Case of Universal Hashing

Now we can prove a much more useful refinement of our previous theorem on the average case complexity of hashing with chaining

Theorem. If n elements are stored in a hash table with m entries and a random hash function $h \in H$ is used for hashing with chaining, where H is a c-universal class of hash functions, then the expected execution time of a *find* or *delete* operation is in $O(1 + cn/m)$.

Proof. We proceed completely analogously to the proof of our previous theorem

The execution time for *find* or *delete* is determined by the length of the list in $A(h(k))$ plus some constant, i.e. it is in $O(1 + E(X))$, where X is a random variable mapping to this length

Proof / cont.

If S is the set of elements stored in the hash table, i.e. $|S| = n$, we can write $X = \sum_{e \in S} X_e$ with Bernoulli-distributed random variables $X_e = \begin{cases} 1 & \text{if } h(e) = h(k) \\ 0 & \text{else} \end{cases}$

Then we get $E(X) = E\left(\sum_{e \in S} X_e\right) = \sum_{e \in S} E(X_e) = \sum_{e \in S} P(h(e) = h(k))$

As h is randomly selected from H, then we have $P(h(x) = h(e)) \leq \frac{c}{m}$, because H is c-universal

This implies $E(X) \leq \dfrac{c \cdot n}{m}$, which completes the proof

A 1-Universal Class

We now define a c-universal class of hash functions with $c = 1$

All functions in the class are defined by a simple formula, so the requirements that the specification should be constant space and the function can be easily evaluated will be satisfied

We make the following restrictive assumptions:

- The keys in $\mathcal{K}$ are bit-strings of fixed length
- The size m of the hash table is a prime number p

These assumptions can be relaxed, as will be handled in some exercises

A 1-Universal Class / cont.

Let $\mathbb{F}_p$ denote the field of integers modulo the prime number p—representatives for the elements of $\mathbb{F}_p$ are $\{0, \dots, p-1\}$

Let $w = \lfloor \log p \rfloor$

Consider keys that are bitstrings of length $k \cdot w$ for some $k \in \mathbb{N}$, so each key $\mathbf{x} \in \mathcal{K}$ can be written as a sequence of k bitstrings of length w

Interpret each such substring of length w be interpreted as a natural number in $\{0, \dots, 2^w - 1\}$, so we can identify a key $\mathbf{x}$ with a k-tuple $(x_1, \dots, x_k)$ with $0 \le x_i \le 2^w - 1$

Then for $\mathbf{a} = (a_1, \dots, a_k) \in \mathbb{F}_p^k$ define a **hash function** $h_{\mathbf{a}} : \mathcal{K} \to \mathbb{F}_p$ by

$$h_{\mathbf{a}}(\mathbf{x}) = \mathbf{a} \cdot \mathbf{x} \mod p = \sum_{i=1}^{k} a_i x_i \mod p$$

using the standard **scalar product** $\mathbf{a} \cdot \mathbf{x}$ on $\mathbb{F}^k$

Example

Let $p = 17$ and $k = 4$, then we have $w = 4$

Then keys correspond to quadruples in $\{0, \dots, 15\}^4$, e.g. $\mathbf{x} = (11, 7, 4, 3)$

A hash function is defined for a quadruple $\mathbf{a} \in \mathbb{F}_{17}^4 = \{0, \dots, 16\}^4$, e.g. $\mathbf{a} = (2, 4, 7, 16)$

For these vales of $\mathbf{a}$ and $\mathbf{x}$ we obtain

$$\begin{aligned} h_{\mathbf{a}}(\mathbf{x}) &= (2 \cdot 11 + 4 \cdot 7 + 7 \cdot 4 + 16 \cdot 3) \mod 17 \\ &= 5 + 11 + 11 - 3 \mod 17 \\ &= 7 \end{aligned}$$

A 1-Universal Class / cont.

We now show that the hash functions $h_{\mathbf{a}}$ defined this way give us a c-universal class with $c = 1$

Then the previous theorem states that with hash functions randomly taken from this class we obtain an average case complexity for hashing with chaining in $O(1+n/m)$

So these hash functions are a good choice for hashing with chaining

Theorem. Let p be a prime number, $w = \lfloor \log p \rfloor$, and $\mathcal{K} = \{0, \dots 2^w - 1\}^k$. Then the class of hash functions

$$H = \{h_{\mathbf{a}} : \mathcal{K} \to \mathbb{F}_p \mid \mathbf{a} \in \mathbb{F}_p^k\}$$

is 1-universal.

Proof

Consider two distinct keys $\mathbf{x} = (x_1, \dots, x_k)$ and $\mathbf{y} = (y_1, \dots, y_k)$—we have to determine the probability $P(h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y}))$

For this we count the number of choices for $\mathbf{a} \in \mathbb{F}_p^k$ that give rise to $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$

Let j be an index such that $x_j \neq y_j$—this exists, because $\mathbf{x}$ and $\mathbf{y}$ are distinct

Then we have $x_j - y_j \not\equiv 0 \bmod p$

Consequently, as p is a prime number, every congruence $a_j(x_j - y_j) \equiv b \bmod p$ has a unique solution, i.e. $a_j \equiv (x_j - y_j)^{-1} \cdot b \bmod p$

Here $(x_j - y_j)^{-1}$ denotes the multiplicative inverse of $(x_j - y_j)$ in the field $\mathbb{F}_p$

Proof / cont.

Then we get

$$\begin{aligned} h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y}) &\Leftrightarrow \sum_{i=1}^{k} a_i x_i = \sum_{i=1}^{k} a_i y_i \mod p \\ &\Leftrightarrow a_j(x_j - y_j) = \sum_{i \neq j} a_i(y_i - x_i) \mod p \\ &\Leftrightarrow a_j = (x_j - y_j)^{-1} \sum_{i \neq j} a_i(y_i - x_i) \mod p \end{aligned}$$

That is, for every choice of values a_i for all $i \neq j$ there is a unique a_j such that $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$

There are p^{k-1} possible choices for the values a_i with $i \neq j$, while the total number of choices for **a** is p^k, hence

$$P(h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})) = \frac{p^{k-1}}{p^k} = \frac{1}{p} ,$$

which shows that H is 1-universal

6.5 More on Hashing with Linear Probing

With hashing with chaining each entry of the hash table contains all elements with the hash table index; it is therefore called a **closed** hashing method

In contrast, an **open** hashing method allows an element to be stored at an entry in the hash table with an index different from its hash value

With an open hashing method table entries can be restricted to single elements

There are many open hashing methods; **linear probing** is the simplest one:

The base form of linear probing has been used in our **C++** implementation

Linear Probing – Base Form

- An element e is stored in $A(h(e))$ (here A denotes the representing array) or in $A(i)$ with $h(e) < i$ (we may count position indices modulo the length of the array and treat $h(e)$ as minimum in a total order)

- Unused entries in a hash table are filled with a special element $\perp$ ("undefined", null pointer, ...)

- If an element e is stored in $A(i)$ with $h(e) < i$, then positions $h(e), \dots, i - 1$ are occupied by other elements

Then the implementation of *insert* is straightforward: Starting from $h(e)$ find the first free position i and insert e into $A(i)$

The implementation of *find* is likewise simple: scan the hash table srarting from $A(h(e))$ until either e or a $\perp$ is found

Linear Probing – Delete

For the *delete* operation the situation is slightly more complicated

We have already seen that simply removing an element, i.e. replacing e by $\perp$ is incorrect, as the third property above could be violated

Alternatives are the following:

- Discard *delete* operations completely—clearly, this only makes sense, if the expected growth of the hash table is bounded

- Instead of deleting an element, mark the entry as "free"—this is achieved in our **C++** implementation using Placeholder objects

  The disadvantage is that the presence of many free positions, where the search cannot be stopped, makes search inefficient

- Reorganise the hash table, if such a gap occurs by moving another element to the freed position—this has to be iterated

Hash Table Reorganisation

If an element $A(i) = e$ is deleted, then we apply the following rule:

> Starting from position i search for the smallest j such that $h(A(j)) \leq i$; then move $e' = A(j)$ to position i (i.e. update $A(i) := e'$ and delete e' in $A(j)$)

The same conditions for the search are applied:

- Consider position indices modulo the length of the representing array
- Stop, if $\perp$ is found

As the application of the rule involves the deletion of another element, the rule has to be iterated accordingly

Example

insert : axe, chop, clip, cube, dice, fell, hack, hash, lop, slash

	an	bo	cp	dq	er	fs	gt	hu	iv	jw	kx	ly	mz
t	0	1	2	3	4	5	6	7	8	9	10	11	12
	⊥	⊥	⊥	⊥	axe	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	⊥	axe	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	clip	axe	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	clip	axe	cube	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	clip	axe	cube	dice	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	clip	axe	cube	dice	⊥	⊥	⊥	⊥	fell	⊥
	⊥	⊥	chop	clip	axe	cube	dice	⊥	⊥	⊥	hack	fell	⊥
	⊥	⊥	chop	clip	axe	cube	dice	hash	⊥	⊥	⊥	fell	⊥
	⊥	⊥	chop	clip	axe	cube	dice	hash	lop	⊥	hack	fell	⊥
	⊥	⊥	chop	clip	axe	cube	dice	hash	lop	slash	hack	fell	⊥

remove ⇩ clip

⊥	⊥	chop	~~clip~~	axe	cube	dice	hash	lop	slash	hack	fell	⊥
⊥	⊥	chop	lop	axe	cube	dice	hash	~~lop~~	slash	hack	fell	⊥
⊥	⊥	chop	lop	axe	cube	dice	hash	slash	~~slash~~	hack	fell	⊥
⊥	⊥	chop	lop	axe	cube	dice	hash	slash	⊥	hack	fell	⊥

Example / cont.

We consider the set $S = \{\text{axe, chop, clip, cube, dice, fell, hack, hash, lop, slash}\}$ — these are all synonyms of the English word “to hash”

We use a hash table of size $m = 13$, so each index $0, \dots, 12$ corresponds to two characters of the English alphabet

The hash function h maps $e \in S$ to i iff the last character of e corresponds to i

The upper half of the previous table shows the states of the hash table after inserting the elements one-by-one in the given order

The lower half of the table shows the effect of deleting clip in $A(3)$: applying the rule above requires lop to be moved from $A(8)$ to $A(3)$, then slash to be moved from $A(9)$ to $A(8)$, then stopping, as neither hack nor fell can be moved to $A(9)$

6.6 Dynamic Hashing

In our treatment of hashing we have seen the possibility to determine the index in an array by means of a hash function, i.e. we compute a location from a key of the value stored at that location

Hashing provides also an alternative to the tree-based index structures (such as B-trees and B^+-trees), which we will handle later

However, for relations in external storage we must take care of fast growing amount of tuples, which rules out any static hashing approach

The hashing methods (chaining, linear probing) we explored allow the size of the hash table to be doubled or halved when necessary

However, this is coupled with rehashing, i.e. a complete recomputation of hash values together with a reorganisation of the data

This is inefficient for large collections of data, so we need more sophisticated dynamic hashing methods

Dynamic Hashing: Ground Principles

As before we apply a **hash function** h to keys k resulting in **hash values** $h(k)$

Each hash value is associated with a **bucket**, i.e. the set of keys with the same hash value

We may think of the keys k in one bucket being linked by means of some data pointer to tuples with key k

- If the key is a primary key, a single data pointer per key is sufficient
- Alternatively, if the keys are used for ordering the tuples, then we may identify a bucket with a pair of blocks and exploit binary/linear search to find the tuple corresponding to a key
- If the key is a secondary key, we may again exploit spiders (to be discussed later)

The various dynamic hashing methods basically differ by the way they handle a bucket overflow

Extendible Hashing

Extendible hashing uses a hash function h on keys k

The hash values $h(k)$ are called **pseudo-keys** that are represented as bit-strings

Pseudo-keys define a binary tree with edges labelled by $0, 1$ such that each node at depth d corresponds to a bit-string of length d

For such a bit-tree of depth d the last d bits of the pseudo-keys are used—in this case d is called the **global depth** of the tree

Leaves of the tree are linked to buckets—if the depth of a leaf is $d' \leq d$, then d' is called the **local depth** of the associated bucket

Extendible Hashing / cont.

Extendible hashing uses an array of size 2^d with pairs (str, p) as entries, where d is the global depth, str is a bit-string of length d, and p is a pointer to a bucket

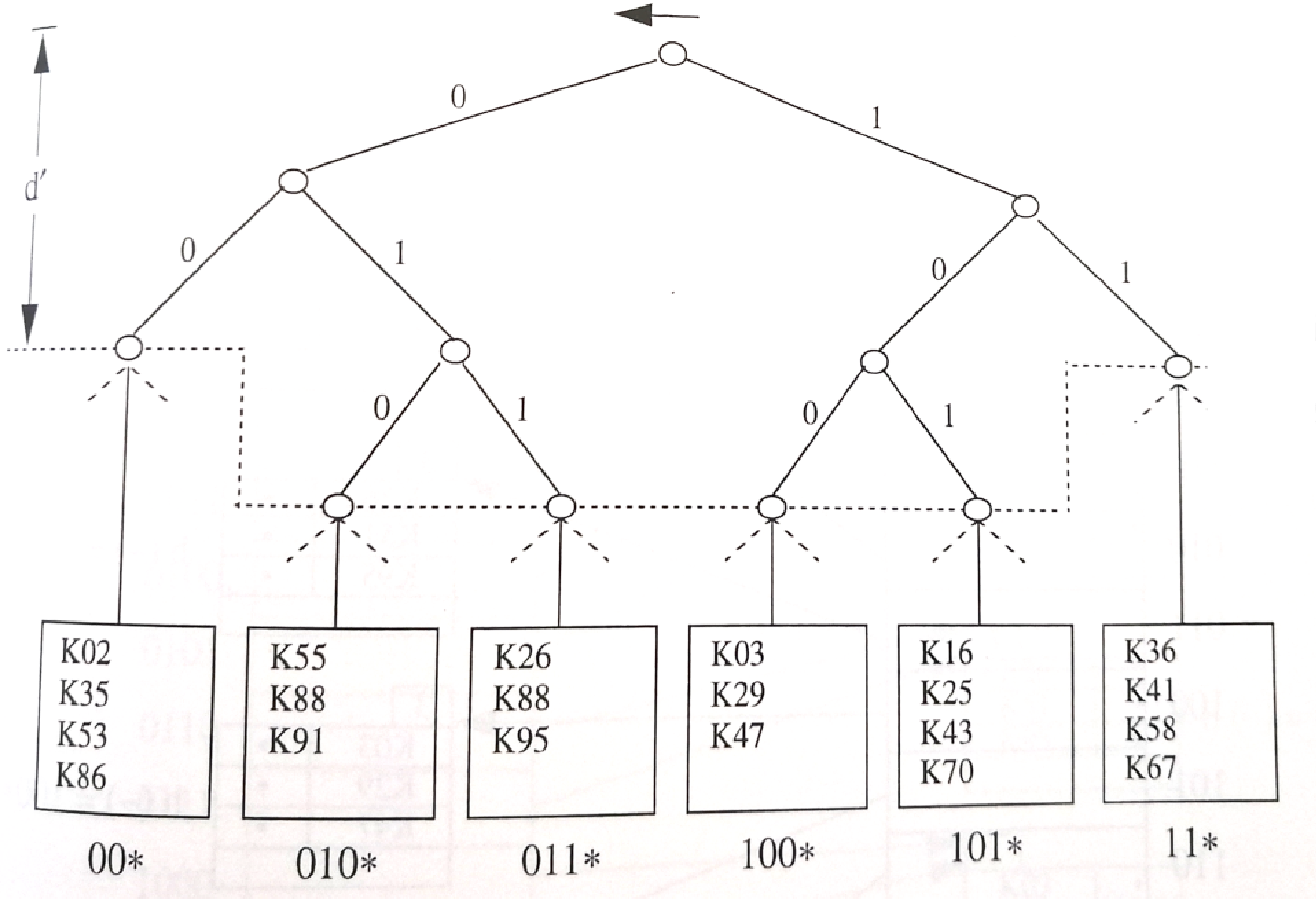
This array is called the **directory**

If a leaf corresponds to a bit-string str' of length d', then there are $2^{d-d'}$ entries(str, p) in the directory, where str' is a prefix of str, and p is a pointer to the same bucket of local depth d'

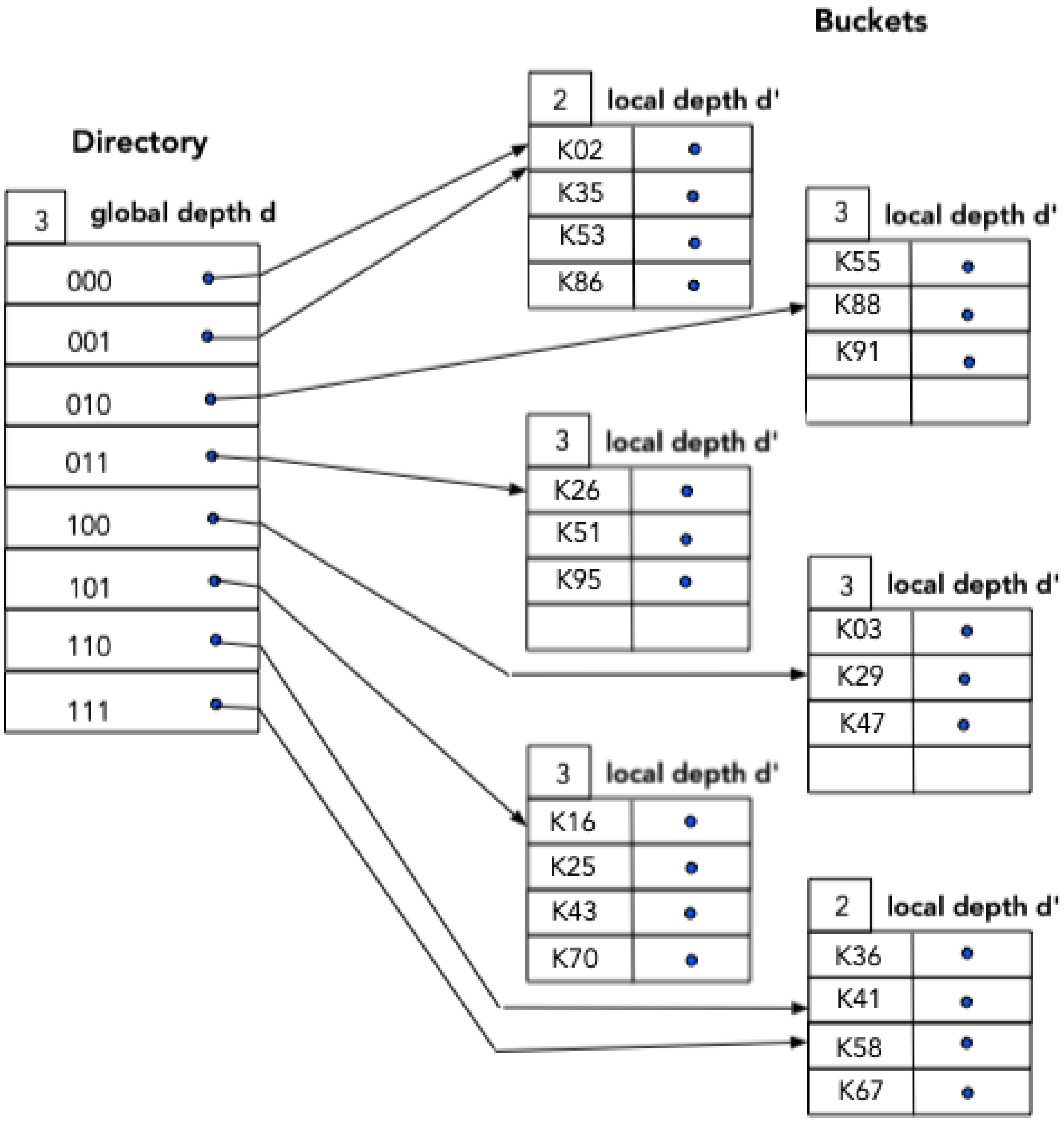
All buckets have the same maximum size b

Example: Take a key $K91$ with pseude-key $k' = h(K91) = 11011010$

Example


d′
0
1
0
1
0
1
0
1
0
1
K02
K35
K53
K86
K55
K88
K91
K26
K88
K95
K03
K29
K47
K16
K25
K43
K70
K36
K41
K58
K67
00*
010*
011*
100*
101*
11*

Example / cont.



Retrieval, Deletion and Insertion

Searching for a tuple with a key k requires several steps:

- First determine the pseudo-key $h(k)$ using a fixed hash function h and the last d bits in its bitstring representation
- Navigate through the bit-tree using the last d' bits to find the data pointer to the bucket of key k
- If the operation is *find*, retrieve the record found following this data pointer (or search in the corresponding bock) and return it
- If operation is *delete*, delete the record found
- If the operation is *insert*, add a new record to main data storage and update the bucket

Splitting of Buckets

As buckets have a fixed size, a sequence of insertions into the same bucket will lead to an overflow

When a bucket is full, it has to be split into two, and the keys k and associated data pointers are distributed according to the next bit in the pseudo-keys $h(k)$—recall that bits are taken from back to front

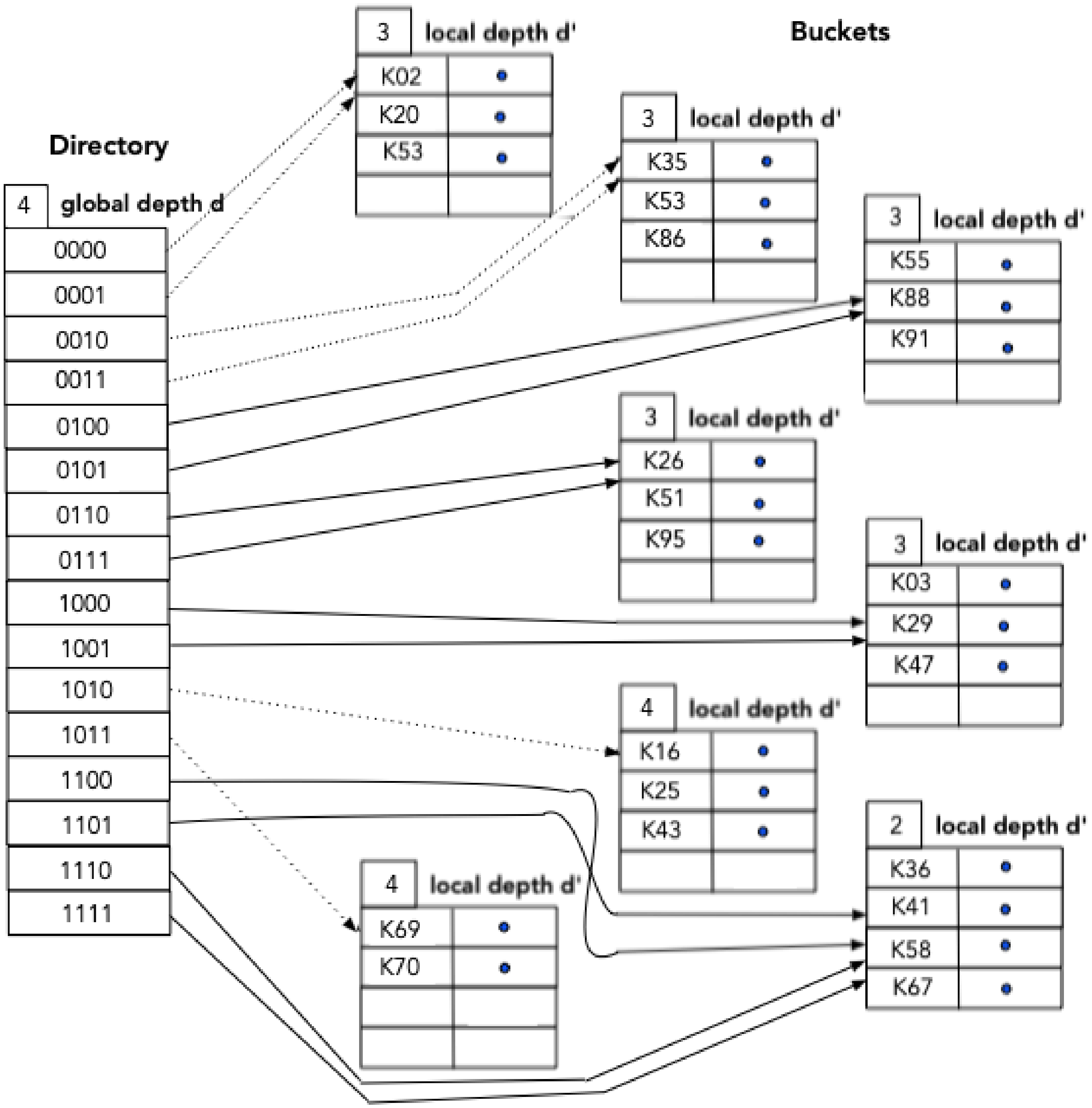
Accordingly the local depth d' of the two new buckets resulting from the split increases by 1, and the pointers to these new buckets are associated with different entries of the directory

In the new local depth d' exceeds the global depth d, the directory is doubled

In this case the pointers to the two new buckets resulting from the split are associated with different new entries of the directory

Buckets not affected by the split will be referenced by twice the number of entries in the directory

Example



Directory

4 global depth d

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Buckets

3 local depth d'

K02	•
K20	•
K53	•

3 local depth d'

K35	•
K53	•
K86	•

3 local depth d'

K55	•
K88	•
K91	•

3 local depth d'

K26	•
K51	•
K95	•

3 local depth d'

K03	•
K29	•
K47	•

4 local depth d'

K16	•
K25	•
K43	•

4 local depth d'

K69	•
K70	•

2 local depth d'

K36	•
K41	•
K58	•
K67	•

Variations

In the example above we first inserted a new key $K20$ leading to a split of the first bucket, then we inserted a new key $K69$ leading to a split of the second last bucket with a doubling of the directory

Extendible hashing is a dynamic hashing method that exploits an additional index structure, the bit-tree

Another method for dynamic hashing with index structures is **virtual hashing**

A disadvantage of these methods is the super-linear growth of the directory

6.7 Linear Hashing

Linear hashing is an alternative dynamic hashing method that does not use an index structure; instead multiple hash functions are used

At each stage we have a **linear sequence of B buckets** with $2^L n \le B < 2^{L+1} n$ with n as the minimum number of buckets—then the buckets are numbered from 0 to B

In each bucket we can store up to b keys, in addition each bucket is associated with an **overflow lists** that is used when the bucket is full, but will nonetheless not be split

Let $p = B - 2^L n$ denote the number of the bucket that will be split next

The criterion for splitting is $\beta > \theta$, where $\beta = \dfrac{N}{B \cdot b}$ is the **fill factor**, N is the total number of keys in the B buckets and overflow lists, and $\theta < 1$ is a **threshold value**

Hash Functions

The assignment of keys to buckets is determined by a sequence $h_0, h_1, \dots$ of hash functions satisfying

$$\begin{aligned} h_0(k) &\in \{0, \dots, n-1\} \\ h_{j+1}(k) &= h_j(k) \text{ or } h_{j+1}(k) = h_j(k) + 2^j n \end{aligned}$$

For instance, hash functions $h_j(k) = k \bmod (2^j n)$ satisfy these requirements

At each stage just two of these hash functions are used: h_L and h_{L+1}

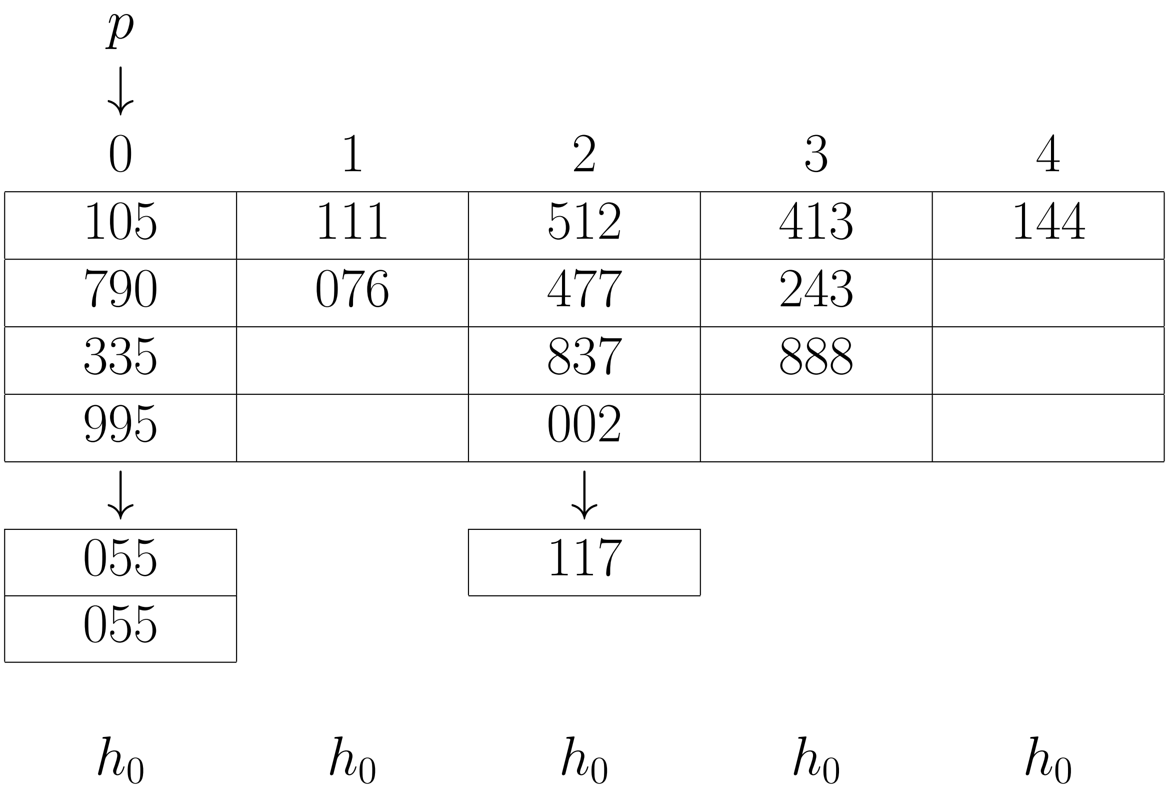
More precisely, we use $h_L(k)$ for $h_L(k) \geq p$; otherwise we use $h_{L+1}(k)$

In this way we obtain growing numbers of buckets—we disregard the shrinking of the number of buckets (use an minimum fill factor) and further optimisations

Example

Let us take $n = 5$, $b = 4$ and $\theta = 0.8$ with hash functions $h_j(k) = k \bmod (2^j n)$

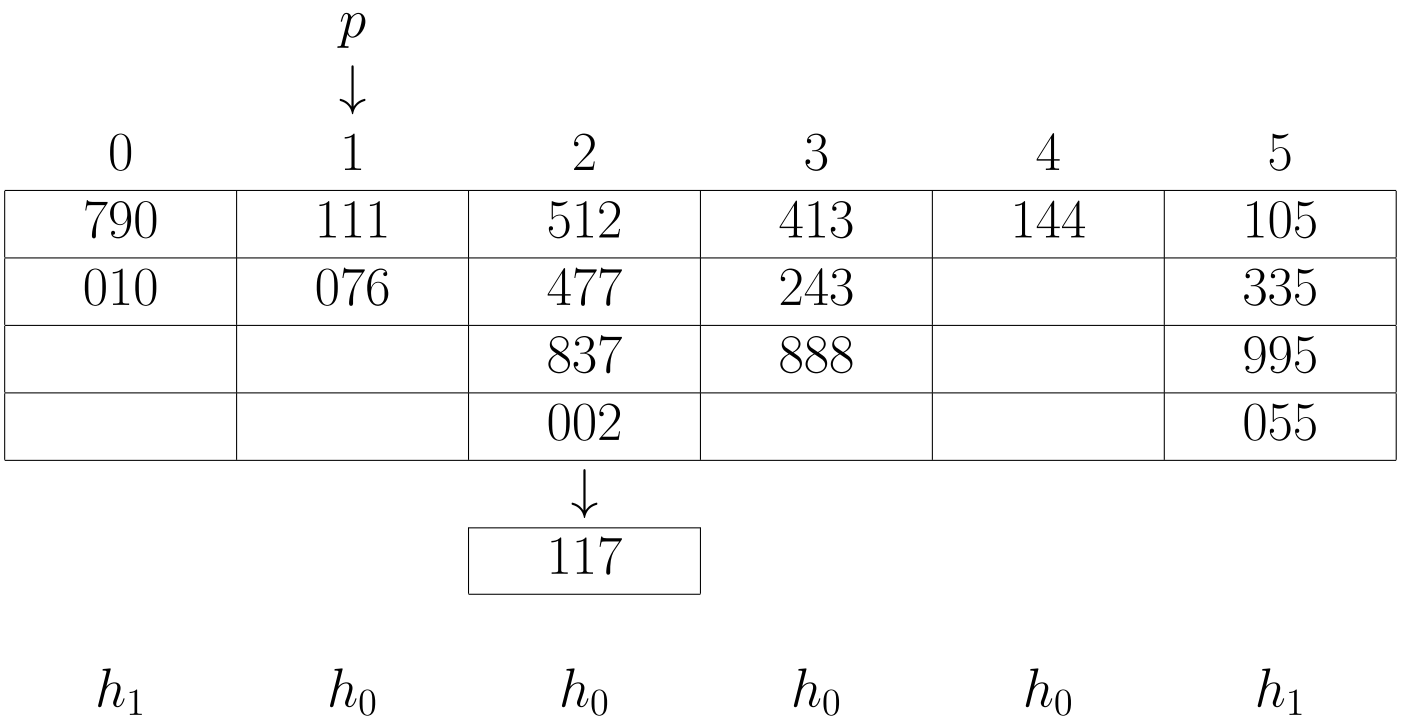
Hash table with $B = 5$ buckets ($L = 0$) before the first splitting of bucket 0:



0	1	2	3	4
105	111	512	413	144
790	076	477	243	
335		837	888	
995		002		
↓		↓		
055		117		
055				
h_0	h_0	h_0	h_0	h_0

Example (cont.)

Hash table after the extensions and splitting bucket 0:

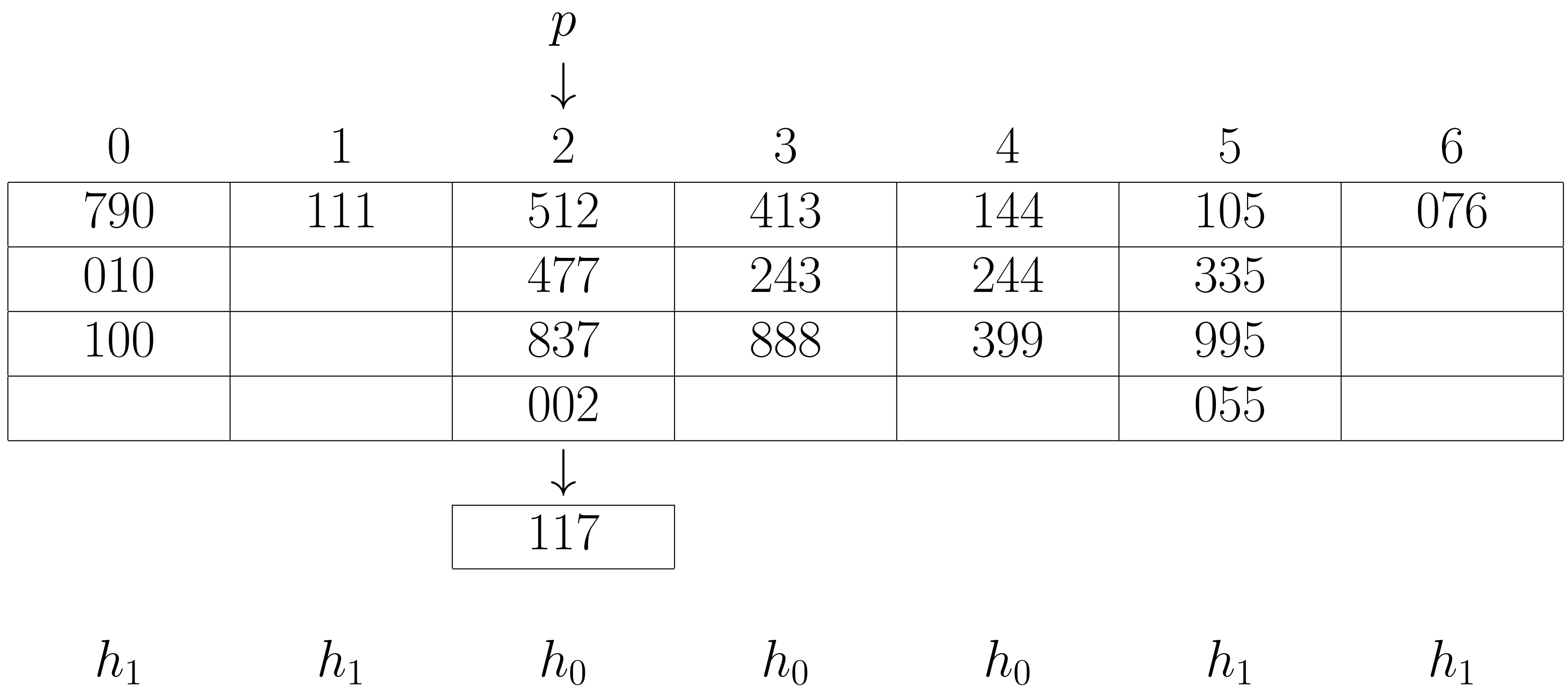


p

0	1	2	3	4	5
790	111	512	413	144	105
010	076	477	243		335
		837	888		995
		002			055
		117			
h_1	h_0	h_0	h_0	h_0	h_1

Example (cont.)

Hash table after the further extensions and splitting bucket 1:



p ↓ 2

0	1	2	3	4	5	6
790	111	512	413	144	105	076
010		477	243	244	335	
100		837	888	399	995	
		002			055	
		↓ 117				
h_1	h_1	h_0	h_0	h_0	h_1	h_1

6.8 Further Remarks on Hashing

Both hashing with chaining and hashing with linear probing have the same complexity: which one is the better choice can only be determined experimentally in a case-by-case way

However, the implementation effort for hashing with chaining is usually larger

One remaining problem is that by means of the theorems we have proven (at least for hashing with chaining) we only have a guarantee for the average case complexity, not for the worst case

Perfect hashing guarantees that the worst case complexity of *find* is in $\Theta(1)$

It can be shown that a perfect hash function with range $\{0, \dots, 2\sqrt{2}n\}$ can be constructed on average in linear time

We omit further details (see e.g. the textbook by Mehlhorn 2008, pp. 92-94)—this is advanced material