Hashing

1. Introduction

Hashing is a method of storing elements in a table in a way that reduces the time for search.

Elements are assumed to be records with several fields. One of the fields is called "key" - the field that we use to search. For example, in a student record the key might be students' ID number. In any database with personal records the key might be the SSN .
In a lexical database (electronic dictionary) the key might be the word itself.

Idea: map the keys to indexes in an array (table)
Array elements are accessed by index. If we can find a mapping between the search keys and indices, than we can store each record in the element with the corresponding index. Thus each element would be found with one operation only.

Example:

To keep information about 1000 students, we can assign identification number to each student – a number between 0 and 999, and then use an array of 1000 elements.

Let us assume that we want to use the SSN of each student. SSN is a 9-digit number. If we want to use directly the number as an index, the table should have much more elements than the number of the students, which is a great waste of space.

However, if we had a mapping between the 1000 SSNs of the students and the numbers between 0-999, than we can use the SSNs as search key and still use an array of 1000 elements to store the records.

Approach: Directly referencing records in a table by doing **arithmetic operations on keys** to map them onto table addresses.

Advantage: the records can be references directly - ideally the search time is a constant , complexity O(1)

Question: how to find such correspondence?
Answers:

- direct access tables
- hash tables

2. Direct-address tables

Direct-address tables – the most elementary form of hashing.
Assumption – direct one-to-one correspondence between the keys and numbers 0, 1, …, m-1., m – not very large.

Searching is fast, but there is
cost – the size of the array we need is determined by the largest key.
Not very useful if only a few keys are widely distributed.

3. Hash Functions

Hash function: function that transforms the **search key into a table address**.

Hash functions transform the keys into numbers within a predetermined interval. These numbers are then used as indices in an array (table, hash table) to store the records (keys and pointers.)

3.1. Keys – numbers

If **M** is the size of the array, then the **hash function h(key)** can be computed in this way:

h(key) = key % M.

This will map all the keys into numbers within the interval [0 , M-1].

3.2. Keys – strings of characters

Treat the binary representation of a key as a number, and then apply 3.1

How keys are treated as numbers:

If each character is represented with **p** bits, then the string can be treated as base-2^p number.

Example:

A K E Y : 00001 01011 00101 11001 = $1 \cdot 32^3 + 11 \cdot 32^2 + 5 \cdot 32^1 + 25 \cdot 32^0 = 44271$

If the keys are very long, an overflow may occur. A solution to this is to apply the Horner's method in computing the hash function.

Horner's method – representation of polynomials:

$a_n x^n + a_{n-1}.x^{n-1} + a_{n-2}.x^{n-2} + \ldots + a_1 x^1 + a_0 x^0 =$

$x(x(\ldots x(x\ (a_n.x + a_{n-1}) + a_{n-2}\ ) + \ldots.\ ) + a_1) + a_0$

Example

$4x^5 + 2x^4 + 3x^3 + x^2 + 7x^1 + 9x^0 =$

$x(x(x(x\ (4.x + 2) + 3) + 1\ ) + 7) + 9$

The polynomial can be computed by alternating the multiplication and addition operations.

V	E	R	Y	L	O	N	G	K	E	Y
10110	00101	10010	11001	01100	01111	01110	00111	01011	00101	11001
22	5	18	25	12	15	14	7	11	5	25

$22.32^{10} + 5.\ 32^9 + 18.\ 32^8 + 25.\ 32^7 + 12.\ 32^6 + 15.32^5 + 14.32^4 + 7.\ 32^3 +$

$+ 11.\ 32^2 + 5.\ 32^1 + 25.\ 32^0 =$

$(((((((((22.32 + 5)32 + 18)32 + 25)32 + 12)32 + 15)32 + 14)32 + 7)32 + 11)32 + 5)32 + 25$

Having this representation we compute the hash function by applying the mod operation at each step, thus avoiding overflowing.

First we compute $h_0 = (22.32 + 5)\%M$

Then we compute $h_1 = (32.h_0 + 18)\%M$

Then $h_2 = (32.h_1 + 25)\%M$

Etc.

Here is the code of a hash function using base = 32. The key is a string of characters, stored in an array **key** of length **Key_Length.**
tbl_size is the size of the table.

```
int hash32 (char key[Key_Length])
{
      int h = 0;
      int i;
            for (i=0; i < Key_Length ; i++)
            {
                  h = (32 * h + key[i]) % tbl_size;
            }
      return h;
}
```

4. Hash Tables

Once we have found the method of mapping keys to indexes, the questions to be solved is how to choose the size of the table (array) to store the records, and how to perform the basic operations

- insert
- search
- delete

4.1. Basic concepts

Let **N** be the number of the records to be stored, and **M** - the size of the array (hash table). The integer, generated by a hash function between 0 and M-1 is used as **an index in a hash table** of M elements.

Initially all slots in the table are *blank.* This is shown either by a sentinel value, or a special field in each slot.
To insert: use the hash function to generate an address for each value to be inserted.
To search for a key in the table**:** the same hash function is used.
To delete a record with a given key - first we apply the search method and when the key is found we delete the record

Size of the table: Ideally we would like to store N records in a table with size N. However, in many cases we don't know in advance the exact number of records. Also, the hash function can map two keys to one and the same index, and some cells in the array will not be used. **Hence we assume that the size of the table can be different from the number of the records.**

A characteristic of the hash table is its **load factor** λ: the ratio between the number of records to be stored and the size of the table: λ=**N/M** .The method to choose the size of the table depends on the chosen method of collision resolution, discussed below.

M must be prime number. It has been proved that if M is a prime number, we obtain better (more even) distribution of the keys over the table.

4.2. Collision resolution

Problem: Obviously, a mapping from a potentially huge set of strings to a small set of integers will not be unique. The hash function maps keys into indices in **many-to-one** fashion. Having a second key into a previously used slot is called **a collision.**

Collision resolution: deals with keys that are mapped to same addresses.
Methods:

- Separate chaining
- Open addressing
  - Linear probing
  - Quadric probing
  - Double hashing

4.2.1. Separate chaining

Invented by H. P. Luhn, an IBM engineer, in January 1953.

Idea: Keys hashing to same address are kept in lists attached to that address

For each table address, a linked list of the records whose keys hash to that address, is built. This method is useful for highly dynamic situations, where the number of the search keys cannot be predicted in advance.

Example

Records: (each character is a key):

Key: A S E A R C H I N G E X A M P L E
Hash (M = 11): 1 8 5 1 7 3 8 9 3 7 5 2 1 2 5 1 5

Separate chaining:

0	1	2	3	4	5	6	7	8	9	10
=	L	M	N	=	E	=	G	H	I	=
	A	X	C		P		R	S	=	
	A	=	=		E		=	=		
	A				E					
	=				=					

Each column is a linked list. Thus we maintain M lists with M list header nodes
We can use the list search and insert procedures for sequential searching

Complexity of separate chaining

The time to compute the index of a given key is a constant. Then we have to search in a list for the record. Therefore the time depends on the length of the lists. It has been shown empirically that on average the list length is N/M (the load factor), provided M is prime and we use a function that gives good distribution.

Unsuccessful searches go to the end of some list, hence we have λ comparisons
Successful searches are expected to go half the way down some list. On average the number of comparisons in successful search is $\lambda/2$. Therefore we can say that runtime complexity of separate chaining is $O(\lambda)$

Note, that what really matters is the lead factor rather than the size of the table or the number of records, taken separately.

How to choose M in separate chaining?

Since the method is used in cases when we cannot predict the number of records in advance, the choice of M basically depends on other factors such as available memory.

Typically M is chosen relatively small so as not to use up a large area of contiguous memory, but enough large so that the lists are short for more efficient sequential search.

Recommendations in the literature vary form M to be about one tenth of N - the number of the records to M to be equal (or close to) N

Other possibilities of chaining:

- Keep the lists ordered: useful if there are much more searches than inserts, and if most of the searches are unsuccessful.
- Represent the chains as binary search tree. Extra effort needed – not efficient.

Advantages of separate chaining– used when memory is of concern, easy to implement
Disadvantages–In case of unevenly distributed keys there may be long lists and many empty spaces in the table.

4.2.2. Open Addressing

Invented by A. P. Ershov and W. W. Peterson in 1957 independently.
Idea: Store collisions in the hash table itself.
Different ways to implement this idea.

If collision occurs, next probes are performed following the formula:

$$\mathbf{h_i(x) = (hash(x) + f(i))mod\ TableSize}$$

where:

$h_i(x)$ is an index in the table to insert **x**
hash(x) is the hash function
f(i) is the collision resolution function.
i - the current attempt to insert an element

Open addressing methods are distinguished by the type of f(i)
Linear probing : f(i) = i.
Quadratic probing: $f(i) = i^2$
Double hashing: $f(i) = i*hash_2(x)$, where $hash_2(x)$ is a second hash function

Problems with delete: a special flag is needed to distinguish deleted from empty positions. Necessary for the search function – if we come to a "deleted" position, the search has to continue as the deletion might have been done after the insertion of the sought key – the sought key might be further in the table.
Total amount of memory space – less, since no pointers are maintained.

4.2.2.1. Linear probing (linear hashing, sequential probing) f(i) = i.

Insert: When there is a collision we just probe the next slot in the table. If it is unoccupied – we store the key there. If it is occupied – we continue probing the next slot.

Search: If the key hashes to a position that is occupied and there is no match, we probe the next position.

a) match – successful search
b) empty position – unsuccessful search
c) occupied and no match – continue probing.

When the end of the table is reached, the probing continues from the beginning, until the original starting position is reached.

Disadvantage: " Primary clustering"

Large clusters tend to build up: if an empty slot is preceded by **i** filled slots, the probability that the empty slot is the next one to be filled is (i+1)/M.

If the preceding slot was empty, the probability is 1/M.

This means that when the table begins to fill up, many other slots are examined.
Linear probing runs slowly for nearly full tables.

Example:

$h_i(x) = (\text{hash}(x) + f(i)) \bmod \text{TableSize} = (\text{hash}(x) + i) \bmod \text{TableSize}$

f(i) = i.

I is the number of the attempt to insert, (starts with 0)

Key:	A	S	E	A	R	C	H	I	N	G	E	X	A	M	P	L	E
Hash (M = 19):	1	0	5	1	18	3	8	9	14	7	5	5	1	13	16	12	5

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	A		A																	
2	S	S																		
3	E						E													
4	A		x	A																
5	R																			R
6	C				C															
7	H									H										
8	I										I									
9	N															14				
10	G								G											
11	E						x	E												
12	X						x	x	x	x	x	X								
13	A		x	x	x	A														
14	M														M					
15	P																	P		
16	L													L						
17	E						x	x	x	x	x	x	E							

4.2.2.2. Quadratic probing f(i) = i^2

Similar to linear probing.
Difference: instead of using linear function to advance in the table in search of an empty slot, we use a quadratic function to compute the next index in the table to be probed.

In linear probing we check the I-th position. If it is occupied, we check the I+1^{st} position, next, I+2^{nd}, etc.

In quadric probing, if the I-th position is occupied we check the I+1^{st}, next we check I+4^{th} next, I + 9^{th} etc.

The idea here is to skip regions in the table with possible clusters.

4.2.2.3. Double hashing f(i) = i*$hash_2$(x)

Purpose – same as in quadratic probing : to overcome the disadvantage of clustering.

Instead of examining each successive entry following a collided position, we use
a second hash function to get a fixed increment for the "probe" sequence.

The second function should be chosen so that the increment and M are relatively prime. Otherwise there will be slots that would remain unexamined.

Example of $hash_2$(x):
$hash_2$(x) = R - (x mod **R),**
R smaller than tableSize, prime

5. Analysis of Open Addressing. Rehashing

The time to hash a key to an address in the table is a constant O(1). If there were no collisions, that would be the search time and the time to insert a new record. However, in case of collisions we will have to count all positions in the hash table that have to be probed in order to find the wanted record.

The run time in the case of collisions is computed to be O(1/(1- λ)) for unsuccessful search, and O((1/ λ)ln(1/(1-λ))) for successful search. Thus the run time is determined by the value of the load factor λ . Recall that in open addressing the load factor λ is less than 1. The smaller the load factor – the faster the search and insertion.

Good strategy: λ < 0.5, i.e. the size of the table is more than twice the number of the records. If the table is close to full, the search time grows and may become equal to the table size.

If the load factor exceeds a certain value (greater than 0.5) we do **rehashing** :

Build a second table twice as large as the original and rehash there all the keys of the original table.

Expensive operation, running time O(N)

However, once done, the new hash table will have good performance.

6. Extendible Hashing

Used when the amount of data is too large to fit in main memory and external storage is used.

N records in total to store, M records in one disk block

The problem: in ordinary hashing several disk blocks may be examined to find an element - a time consuming process.

Extendible hashing: no more than two blocks are examined.

Idea:

Keys are grouped according to the first m bits in their code.

Each group is stored in one disk block.

If the block becomes full and no more records can be inserted, each group is split into two , and m+1 bits are considered to determine the location of a record.

Example: assume we have 4 groups of keys according to the first two bits:

00	01	10	11
00010	01001	10001	11000
00100	01010	10100	11010
	01100		

4 disk blocks, each can contain 3 records

New key to be inserted: 01011.

Block2 is full, so we start considering 3 bits:

000/001 (still on same block)	010	011	100/101	110/111
00010	01001	01100	10001	11000
00100	01010		10100	11010
	01011			

Size of the directory (pointers from the cells in the first row above to the columns below): $2^D = O(N^{(1+1/M)} / M)$

D - the number of bits considered.

N - number of records

M - number of disk blocks

7. Conclusion

Hashing is the best search method (constant running time) if we don't need to have the records sorted.

The **choice of the hash function** remains the most difficult part of the task and depends very much on the nature of the keys.

Separate chaining or open addressing?

If there is enough memory to keep a table twice larger than the number of the records - open addressing is the preferred method.

Separate chaining is used when we don't know in advance the number of the records to be stored. It requires additional time for list processing, however it is simpler to implement.

Some application areas

Dictionaries, on-line spell checkers, compiler symbol tables.