1 Hash Tables

The problem at hand is that of managing a collection of objects. Each object is uniquely identified by a key—this means that there are no two objects that have identical key. For example, the SSN can be a good choice for a key to represent a student in a class, since no two students will have the same SSN.

The two major operations that are of interest are the insertion of new objects and the retrieval of objects. In the retrieval operation, given a key we want to obtain the corresponding object (if present).

The ideal situation is the one where, given a key, we can obtain immediately the object, without any search. Let us assume elements are stored in a fixed size table. Thus we would like to have a function $f$ which, given a key, produces the index of the location in the table where the element with such key has been stored

$$f(\text{key}) = \text{location of object}$$

When an insertion has to be performed, we can apply $f$ to the key to determine which location of the table should be used. This function is what we call the Hash Function. The table that stores the elements is then called Hash Table. See also Fig. 1. The process of using an hash tables proceeds as follows:

- whenever I need to insert a new element, e.g., with data $X$ and key $K$, then I apply the hash function to the key (compute $f(K)$); this will return an index in the table ($I$) and I will proceed in storing the new element ($\langle K, X \rangle$) in entry $I$ of the table. In this case, the hash function is telling me where I should place the element of key $K$ in the table.

- whenever I want do perform a search to determine the data (if any) associated to key $K$, I will simply apply the hash function to the key (i.e., compute $f(K)$), and use the result as index to access the table. If the hash function returns $I$, then I will check entry $I$ in the table to located the desired data. In this case, given a key $K$, the hash function is telling me where that key has been stored in the table.

If $f$ is our hash function, and $key$ is one of the keys, then the table index $f(key)$ is what we call the Home Address of key.

If $f$ can be computed, we would like it to map each key to a different location of the table; such a function is called perfect hashing function.

Let us consider the following simple situation. A company has 100 employees, and upon hiring each employee is assigned a numeric code to identify him/her within the company. For example, the owner is employee 1, the vicepresident is 2, etc.

We can use this numeric code as the key for each employee. The information about employee $i$ will be stored in the entry of index $i$ in the table (a 100 elements table). Each time I need the data about employee $i$ I only need to access table[i] and I will be able to retrieve all data about such employee.

In this case, there is a one-to-one mapping between keys and entries in the table. In a single step I can insert the data of an employee in the table and in a single step, given the key, I can retrieve the data about the employee.

Perfect hashing is in general infeasible. To have perfect hashing, the hash function should assign a separate table entry to each possible input key. Often the space of the input keys is very large (e.g., think about using SSN
as a key) and it is impossible to have a table that has a separate entry for each possible key. These are difficult to compute. It can be that $f$ is a normal hashing function. This means that $f(key)$ produces a tentative location which may not contain the object with the indicated key. This happens because the space of possible keys is MUCH larger than the number of entries in the table, and as result we can have many different keys that are mapped by the hash function to the same location of the table. If the location of index $f(key)$ does not contain the desired object, than further search (called rehashing) is required.

Thus, while sorting arranges elements directly in a proper order, hashing scatters elements in a table (hash table). The distribution of the elements in the table is such to guarantee that a given element can be quickly found given his key. Observe that in this organization we have only elements scattered in a table; no notions of parent, child, precedence, etc. is present in this organization. Thus hash table are meaningful only when insert and retrieval are the only operations required. In particular, deletion of elements is not simple.

For example: consider a table with 7 locations (e.g., `table[0]` to `table[6]`). Consider the keys to be integer positive numbers. We could adopt the following hashing function:

$$f(key) = key \% 7$$

This is a correct function in the sense that for each key it will produce a valid table index (since the only valid table indices are the numbers from 0 to 6, and the function $f$ will always produce values in this range).

If we insert the element with key 374, then this will be placed in the location $f(374) = 374 \% 7 = 3$ of the table. If the successive element inserted is an element of key 1091, then this will be placed in the location $f(1091) = 6$ of the table.

What happens if we insert next the element having key 227 ? The hash function produces value $f(227) = 3$, but location 3 of the table has already been used. This is a collision. In this case we must adopt a strategy for collision-resolution.

The problem occurs because the hash function is not perfect, i.e., it can map different keys to the same table location. On the other hand perfect hashing functions are typically very complex and expensive, thus it is often worth considering non-perfect functions and develop efficient collision-resolution mechanisms.

### 1.1 Hashing Functions

In this section we will study some possible functions which can be used for hashing.
1.1.1 Direct Hashing

This is applicable only in very particular cases. Direct hashing assumes that the key is a number, and the number itself is used as index in the table. This means that the hash function is defined as follows:

$$\text{hash}(\text{key}) = \text{key}$$

In order to be feasible, this solution requires knowledge about the possible keys that can arise. The keys must be all in the range 0 . . . S − 1 where S is the size of the table.

1.1.2 Digit Selection

Let us assume that the size of the table (TableSize) is a power of 10 (e.g., 10, 100, 1000)—i.e., TableSize is $10^k$. Let us also assume that the key is an integer number. If we have a guarantee that the keys follow a good random distribution (i.e., each key has the same probability to appear), then we can obtain a good behavior by taking as hash function the function which extracts the last $k$ digits of the key. Thus if the key is the sequence of digits

$$\text{key} = d_n d_{n-1} d_{n-2} \ldots d_3 d_2 d_1$$

then the hash function applied to key will produce the number represented by the last $k$ digits, i.e., $d_k d_{k-1} \ldots d_2 d_1$. This operation can be easily achieved by simply computing

$$f(\text{key}) = \text{key} \% 10^k$$

where $\%$ denotes the modulo operation (remainder of the division).

In the same way, we can design different hash functions by selecting different digits from the key. For example, another hash function can be obtained by selecting the first $k$ digits of the number ($d_n \ldots d_{n-k+1}$). Is this always a good choice? It depends on the nature of the key. E.g., social security numbers.

1.1.3 Division

The idea of the digit selection can be generalized. Let us assume in general that TableSize is $m$, where $m$ can be an arbitrary chosen number. Then, since the hash function is supposed to generate values in the range 0 to $m − 1$, this can be directly achieved by performing

$$f(\text{key}) = \text{key} \% m$$

Now, having dropped the restriction of $m$ being a power of 10, the result of the hash function is not directly related to the digits of key. Nevertheless, we are guaranteed that the remainder of a division by $m$ is always going to be a number in the range 0 to $m − 1$, which is satisfactory for accessing an hash table.

In practice this approach works very well. The modulo operation is very efficient in most machine (hardware support). The critical thing is the selection of the size $m$ of the table. For example, if we select $m = 25$ then we have the following behavior: all the keys which can be divided by 5 will map to either location 0 or 5 or 10 or 15 or 20 of the table. Thus, a large subset of the keys maps to a subset of the location of the table. This is unpleasant; all keys whose last digit is either 0 or 5 will map to a fixed set of locations in the table. If we do not have good knowledge about the distribution of the keys this can be dangerous.

This behavior can be avoided if we guarantee that the key and $m$ do not have common factors. Thus, chosing $m$ to be a prime number will solve the problem.

1.1.4 Multiplication

In order to create more randomness in the key, we can perform some operations on the key before selecting certain digits from its representation. A typical approach is that of multiplying the key by itself and then performing digit selection on the result (e.g., select the digits in the middle). For example, if we have that the size of
the table is 1000 and the keys have 5 digits, then starting from the key $d_1d_2d_3d_4d_5$ we can multiply by itself
\[
\begin{array}{c}
\times \\
\hline
r_1r_2r_3r_4r_5 \quad r_6r_7r_8r_9r_{10}
\end{array}
\]
and the extract the middle digits to produce the hash value ($r_4r_5r_6$).

The selection of the digits should be done again with care to avoid complications. Consider the key 54321 and let us adopt the policy of selecting the rightmost two digits. The product of 54321 by itself produces 2950771041, thus the last two digits are 41. But the 41 was obtained by $1 \times 21$ and $2 \times 21$. Thus it depends only on the last two digits of the key!! Any key with the same last two digits will produce the same hash value.

### 1.1.5 Folding

Suppose that we have a six-digit key
\[
key = d_1d_2d_3d_4d_5d_6
\]
and let us assume that we have a table that has 100 positions (i.e., the hash function is expected to produce numbers in the range 0…99). Since the result of the hash function is supposed to be a two-digit number, we can break the key into subsequences of length two:
\[
d_1d_2 \quad d_3d_4 \quad d_5d_6
\]
and then add them together
\[
d_1d_2 + d_3d_4 + d_5d_6
\]
(if the result is a number with 3 digits, we can just ignore the leftmost digit).

E.g., if the key is 123456 then we can perform the computation as:
\[
12 + 34 + 56 = 102
\]
if we ignore the extra digit 1 we get the index 02.

### 1.2 Collision Resolution

Collision resolution is required whenever the hash function may return the same table location for two different keys.

Let us call $TableSize$ the number of locations available in the table. Let us denote as $n$ the current number of objects present in the table. We define as load factor the result of
\[
\alpha = \frac{n}{TableSize}
\]
Often we will allow more objects to be stored in the same location of the table; in this case each location of the table will be called a **bucket**.

#### 1.2.1 Open Address Method

The open address method, in case of collisions, keeps looking for an empty location in the table. This means that along with the hash function we also have a strategy for searching for empty locations in the table.

Let us assume that $f$ is the hash function and $k$ is the key that we are trying to insert in the table. The first location that we will try to use is $f(k)$. If this location is already taken, then we will start generating a sequence of alternative table indices to explore in order to determine an empty location—this sequence of indices is called **probe sequence**.

Let us consider the case of a table which uses the hash function
\[
f(key) = key \% 7
\]
Let us consider the following elements to be inserted : 911, 374, 1091. These would be inserted respectively in locations 1, 3, and 6.

Now if we insert 227, since $f(227) = 3$ we will have a collision. Open address keeps searching for a free location to insert 227. We have various possible approaches:
- *Linear Rehashing* where simply a sequential search is started from location \( f(key) \) until a free location is detected. In the example, since location 3 is taken, we will look at location 4, which is free, and the new element 227 will be placed there. The same path is traversed during searching (first compute \( f \) of the key, and, if the element in such location does not have the right key, a sequential search is performed).

Thus the probing sequence is generated by

\[
k = (f(key) + j) \% TableSize
\]

where \( j \) is the number of the current attempt to solve the collision.

For example, if the \( TableSize \) is 11 and the key is 272, then the probe sequence is:

\[ 8, 9, 10, 0, 1, 2, 3, \ldots \]

Note that 8 is obtained by applying the hash function \( (272 \% 11 = 8) \) and all the other elements are obtained by adding 1.

There are two problems with this scheme:

- **Primary Clustering:** all the keys that hash to the same location will follow the same pattern of analysis. Thus any key that hashes to 3 will collide with all the keys that have previously hashed to 3.

- **Secondary Clustering:** If later we insert a key that hashes to 4, we will collide with 227. Thus we have previously taken a spot dedicated to a different hash value. As result, keys that map to different hash values may end up sharing the same pattern of rehashing.

The situation can be improved by having a non-constant step in the sequential search.

**Example 1.1** Try to use a table of size 7, use the division method to compute the hash function, and insert in the order the keys 911, 374, 1091, 227, 421, 624.

Note that the linear rehashing has the advantage of being simple to compute and guarantees that each location of the table can be eventually used.

The overall structure of a linear rehashing insertion:

```c
Insert(key, data):
    index = hash(key);
    for (i = 0; i < TableSize; i++)
        if (table[index] is free)
            {
                table[index].key = key;
                table[index].data = data;
                return;
            }
        else
            index = (index + 1) % TableSize
    printf(''TABLE IS FULL'');
```

The search of an element will follow the same scheme: we use the key to generate the initial index (via hash function) and then we keep scanning the table until we find the desired key or determine that the key is not present:

```c
Search(key):
    index = hash(key);
    for (i=0; i < TableSize; i++)
```
Quadratic Rehashing: a different scheme is to rehash using the following function:

\[ k = (\text{homeaddress} + j^2) \% \text{TableSize} \]

where \( j \) is the number of the current rehashing attempt. Thus at the first collision we will try with \( j = 1 \), at the second with \( j = 2 \), etc. This solves the problem of secondary clustering. The probe sequence generated will look like:

\[ \text{hash}(key), \text{hash}(key) + 1, \text{hash}(key) + 4, \text{hash}(key) + 9, \text{hash}(key) + 16, \ldots \]

**Example 1.2** Generate the probe sequence for table of size 7, with the same order of insertion as seen in the previous example.

The advantage of quadratic rehash is that secondary clustering is not present any longer. But the method has one drawback: at most \( 1/2 \) of the locations of the table will be accessed starting from a given home address.

Double Hashing: in this case we have two hash functions, one to compute the initial location, and the other to compute the step of the search for a free location:

\[ f(key, j) = (h(key) + jh'(key)) \% \text{TableSize} \]

This should hopefully avoid primary clustering, as now keys with the same initial location may follow different chain of searches.

To have a good behavior, \( h' \) should produce values which are relatively prime to \( \text{TableSize} \). If \( \text{TableSize} \) and \( h'(key) \) have a common divisor \( d \), then only \( (1/d) \)th of the table is searched.

\[ h(key) = \text{key} \% m \]
\[ h'(key) = 1 + (\text{key} \%(m - 2)) \]

For example, if we have a table of size 7, then we could use the rehash function \( h'(k) = 1 + (k \% 5) \).

### 1.2.2 Bucket Hashing

An alternative to open address methods is represented by the use of bucket hashing. In this case, each entry in the table is capable of storing not just one element, but a collection of elements. For example, each entry in the table could be an array that can store 5 elements (the entry in the table that can store more than one element is what we call a bucket). Using this scheme, when a collision occur, we may still be able to insert the element at its home address, by using one of the available positions in the bucket. This will work fine as long as the number of collisions on the same home address is smaller than the size of the bucket.
1.2.3 External Chaining

A second approach is to store elements that are colliding in the same table location, by creating a linked list which originates from the table cell. Thus, in the previous example, when 227 is inserted, this is added as a new element to the linked list originating from the location 3 of the table.

When searching an element, the linked list has to be traversed. We can maintain the sequence of elements however we prefer (e.g., binary search tree).

This method has the following advantages:

- easy deletion;
- no limits to number of elements inserted;
- good performance;

1.3 An Implementation

We define an ADT which is used to implement an hash table. The operations allowed are described in the following header file (hash.h):

```c
#define TableSize 7

struct element /* one element of the table */
{
    int key;
    int flag;
    ....
};

typedef struct element Element;
typedef struct element *Table;

/* Operations to manipulate the table */

Table CreateTable();
void DeleteTable(Table);
void InsertElement(Table,Element);
Element FindElement(Table,int);
......

Let us assume that we intend to implement a table which uses division hashing and adopts a linear policy to perform collision resolution (file hash.c).

#include "hash.h"

/* Prototypes of additional functions */
int hash(int);
int rehash(int,int);

/* Function Definitions */

/*****************************/
/* Hash Function: given a */
/* key computes initial */
/* location */
/*--------------------------*/

int hash(int key)
{
    return (key % TableSize);
}

/*--------------------------*/
/* Rehash Function: given a */
/* an key and an attempt */
/* number computes the new*/
/* location to probe */
/*--------------------------*/

int rehash(int key, int attempt)
{
    return ( (hash(key) + attempt) % TableSize );
}

/*----------------------------*/
/* Create a new Hash table */
/*----------------------------*/

Table CreateTable()
{
    Table temp;
    int i;

    temp = malloc(sizeof(Element)*TableSize);

    for (i=0; i < TableSize; i++)
        temp[i].flag = 0;

    return (temp);
}

/*-------------------------------*/
/* Destroy an existing table */
/*-------------------------------*/

void DeleteTable(Table t)
{
    free(t);
}

/*--------------------------------*/
/* Insert a new element in table */
/*--------------------------------*/
void InsertElement(Table t, Element e)
{
    int i;
    int visited = 0;

    e.flag = 1; /* valid element */

    i = 0;
    while ( (visited < TableSize) &&
            (t[rehash(e.key,i)].flag == 1) )
    {
        visited++;     
        i++;          
    }
    if (visited == TableSize)
        printf(''Table is full!
'');
    else
        t[rehash(e.key,i)] = e;
}

Element FindElement(Table t, int key)
{
    int i;
    int visited;

    i=0; visited=0;
    while ( (visited < TableSize) &&
            (t[rehash(key,i)].key != key) &&
            (t[rehash(key,i)].flag == 1) )
    {
        i++;     
        visited++;          
    }
    if (t[rehash(key,i)].key == key)
        return (t[rehash(key,i)]);
    else
        printf(''Not found'');
}

1.4 Application: Sets