**Hashing**

A **hash function** is any [well-defined procedure](http://myzain.gh.zain.com.t9space.com/power/m/http/en.wikipedia.org/wiki/Algorithm) or [mathematical function](http://myzain.gh.zain.com.t9space.com/power/m/http/en.wikipedia.org/wiki/Function_%28mathematics%29) that converts a large, possibly variable-sized amount of data into a small datum, usually a single [integer](http://myzain.gh.zain.com.t9space.com/power/m/http/en.wikipedia.org/wiki/Integer) that may serve as an [index](http://myzain.gh.zain.com.t9space.com/power/m/http/en.wikipedia.org/wiki/Array_index) to an [array](http://myzain.gh.zain.com.t9space.com/power/m/http/en.wikipedia.org/wiki/Array_data_type). The values returned by a hash function are called **hash values**, **hash codes**, **hash sums**, or simply **hashes**.

Hash functions are mostly used to speed up table lookup or data comparison tasks such as finding items in a [database](http://myzain.gh.zain.com.t9space.com/power/m/http/en.wikipedia.org/wiki/Database).

Thereforea hash function acts as a mapping, h(K), that maps from key K, onto the index of an entry. A black-box into which we insert a key (e.g. Voters ID number) and out pops an array index.

As an example lets use an array of size 11 to store some airport codes, e.g. PHL, DCA, FRA.

In a three letter string *X*2*X*1*X*0 the letter ’A’ has the value 0, ’B’ has the value 1 etc.

One hash function is:

*h*(*K*) = (*X*2 *\** 262 + *X*1 *\** 26 + *X*0 )%11

Applying this to ”DCA”:

*h*("*DCA*") = (3 *\** 262 +2 *\** 26+ 0)%11

*h*("*DCA*") = (2080)%11

*h*("*DCA*") = 1

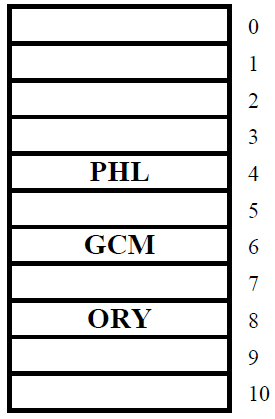
|  |
| --- |
|  |
| **DCA** |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

An example of a hash function for inserting integer keys into a hash table

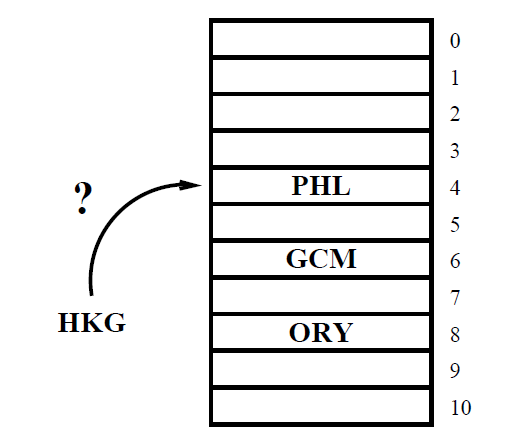
*h****(K) = K mod SizeOfArray***

**Collisions**

Inserting ”PHL”, ”ORY” and ”GCM”:

****

However, inserting ”HKG” causes a collision.

****

**Collisions**

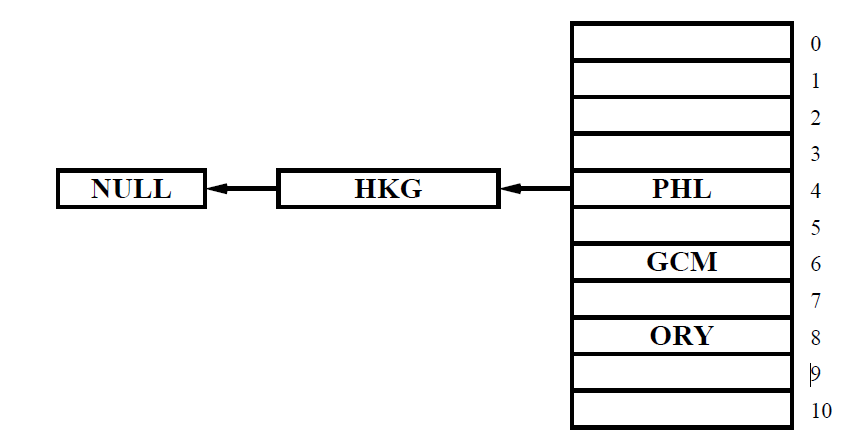
An ideal hashing function maps keys into the array in a *uniform* and *random* manner.

*Collisions* occur when a hash function maps two different keys onto the same address.

The policy of finding another free location if a collision occurs is called **open-addressing**.

The simplest method of open-addressing is *linear probing.* When a collision occur, keep going down until a free location is encountered. Once the bottom of the table is reached, the probe sequence wraps around to the beginning of the table.

Open-addressing is not the only method of collision reduction. Another way of resolving collision is by the use of **separate chaining**. In separate chaining, all nodes that map to an array location are placed in a linked list.

****

**Algorithm complexity** of hashing is **O(1)**

**SORTING**

In Computer Science, a **sorting algorithm** is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms (such as search algorithms) that require sorted lists to work correctly.

**Bubble Sort**

*Bubble sort* is a straightforward and simplistic method of sorting data. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, then it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. This algorithm is highly inefficient, and is rarely used. For example, if we have 100 elements then the total number of comparisons will be 10000.

**Algorithm**

do

{

swapped = 0;

for (x = 0; x < array.size -1 ; x++)

{

if (array[x] > array[x+1])

{

Swapped = 1;

tmp = array[j];

array[j] = array[j + 1];

array[j + 1] = tmp;

}

}

} while (swapped);

**An example:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 4 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 1 | 4 | 5 | 2 | 6 | 5 | 4 | 9 |

b

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 3 | 4 | 2 | 5 | 5 | 4 | 6 | 9 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 3 | 2 | 4 | 5 | 4 | 5 | 6 | 9 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 2 | 3 | 4 | 4 | 5 | 5 | 6 | 9 |

**Complexity:** Bubble sort *average case* and *worst case* are both **O(*n*²)**

**Selection Sort**

Selection sort is a simple sorting algorithm that improves on the performance of bubble sort. It works by first finding the smallest element using a linear search and swapping it into the first position in the list, then finding the second smallest element by scanning the remaining elements, and so on.

**Algorithm :**

int x,y,min;

for (x = 0; x < array.size-1; x++)

{

min = x;

for (y=x+1; y<array.size; y++)

{

if (array[y] < array[min])

{

min = y;

}

}

/\* swap the places \*/

tmp = array[x];

array[x] = array[min];

array[min] = tmp;

}

**Example:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 4 | 7 | 5 | 9 | 10 | 6 | 8 | 2 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **3** | 4 | 7 | 5 | 9 | 10 | 6 | 8 | 2 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | **2** | 4 | 7 | 5 | 9 | 10 | 6 | 8 | **3** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | **3** | 7 | 5 | 9 | 10 | 6 | 8 | **4** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | **4** | 5 | 9 | 10 | 6 | 8 | **7** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | **5** | 9 | 10 | 6 | 8 | 7 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | **6** | 10 | **9** | 8 | 7 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | **7** | 9 | 8 | **10** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | **7** | **8** | **9** | 10 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | **7** | 8 | **9** | 10 |

**```````````````````````````````**

**Complexity: Selection Sort** *average case* and *worst case* are both **O(*n*²)**