

**Hash Function types:**

**1, Division** *hD*(*x*) = *x* % *M* :

* + Using the modulus (%) operator.
  + We divide the key *x* by some number *M* and use the remainder as the hash index for *x*.
    - This gives indexes that range from 0 to *M* - 1,

where *M* = that table size (hash table).

* The choice of *M* is critical.
  + If *M* is divisible by 2, then odd keys to odd indexes and even keys to even ones. (biased!!)
  + If M is a power of 2, i.e. m = 2^p , then h(k) is just the p lowest-order bits of k. (biased!!)
  + If M = pH, then keys in the set {H, 2H, 3H, …, (p-1)H, pH, (p+1)H,…, kH,…} map to p positions {H, 2H, 3H, …, (p-1)H, 0} only (biased!!)
  + A good choice for *M* would be : *M* a prime number such that *M* does not divide *rk*±*a* for small *k* and *a*.

**2, Folding**

* + Partition key *x* into several parts.
  + All parts except for the last one have the same length.
  + Add the parts together to obtain the value y, the hash index then is *h*(*x*) = y % *M.*
* Two possibilities (divide *x* into several parts)
  + ***Shift folding***:   
    Shift all parts except for the last one, so that the least significant bit of each part lines up with corresponding bit of the last part. Suppose x = 72320354121324
    - *x*1=723, *x*2=203, *x*3=541, *x*4=213, *x*5=24,

index= (*x*1 + *x*2 + *x*3 + *x*4 + *x*5) % 1000 = 1704%1000 = 704

* + ***Boundary folding*** (folding at the boundaries):   
    reverses every other partition before adding
    - *x*1=723, *x*2=302, *x*3=541, *x*4=312, *x*5=24, index=1902%1000 = 902

**3,** In the **mid-square** method, the key is *squared* and the middle or *mid* part of the result is used as the address

Exp: key = 3121² = 9 740 641 ->index = 406 mod Tsize

Since the middle bits of the square usually depend upon all the characters in a key, there is high probability that different keys will produce different hash indexes.

**4,**  In the **extraction** method, only a part of the key is used to compute the address

Exp: 123-45-6789->1234-5-6789 -> index = 1289 mod TSize

**5,** Using the **radix transformation**, the key *K* is transformed into another number base; *K* is expressed in a numerical system using a different radix.

Exp: 34510 = 4239 -> index = 423 mod TSize

**Độ phức tạp**

Gọi:

* n là số phần tử ta cần lưu trong Hash table
* k là số bucket

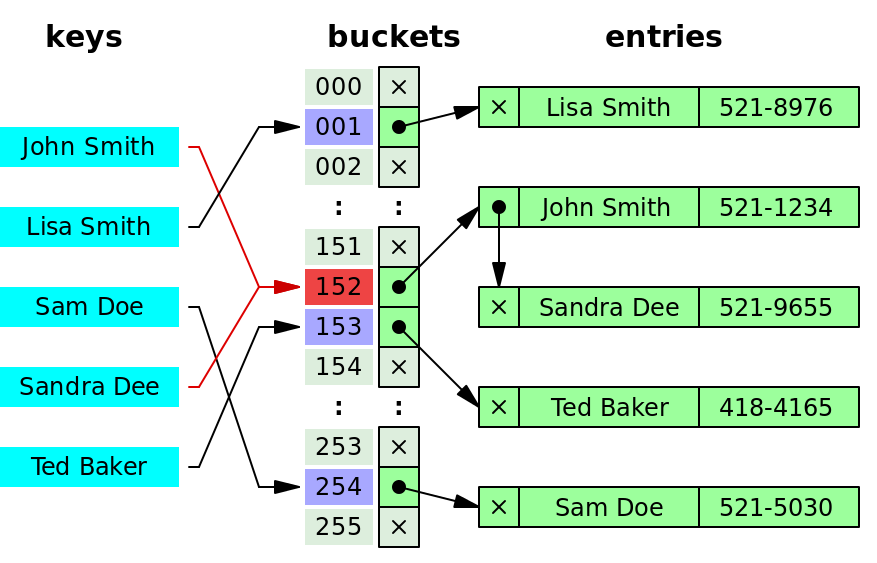
Giá trị n/k được gọi là **load factor**. Khi load factor nhỏ (xấp xỉ 1), và giá trị của hàm Hash phân bố đều, độ phức tạp của các thao tác trên Hash table là O(1)

**Hash collision gồm 3 cái:**

**Chaining, open addressing, bucket addressing**

**\*Separate chaining ( một thằng nữa là Coalesced chaining)**

Trường hợp một hash bucket chứa nhiều hơn một giá trị ta gọi đó là **Hash collision** (va chạm). Việc xử lý hash collision rất quan trọng đối với độ hiệu quả của bảng băm. Một trong những phương pháp đơn giản nhất là cài đặt các [danh sách liên kết](https://vnoi.info/wiki/algo/data-structures/array-vs-linked-lists) ở các bucket. Kĩ thuật này được gọi là **Separate chaining**:

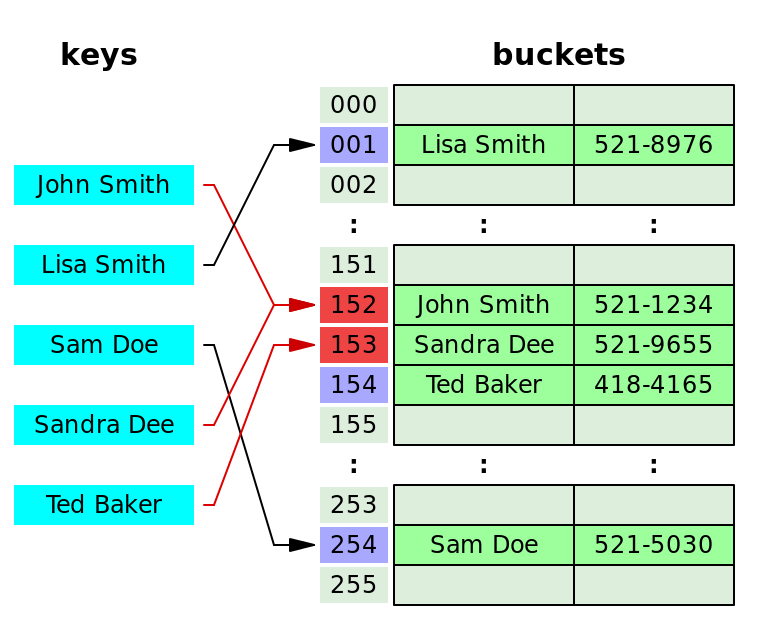


Giải thích hình minh họa:

* Mỗi bucket là 1 danh sách liên kết
* John Smith và Sandra Dee cùng có giá trị hàm hash là 152, nên ở bucket 152, ta có 1 danh sách liên kết chứa 2 phần tử.

**\*Open Addressing**

Tư tưởng của **Open Addressing** là, khi xảy ra Hash collision, ta lưu vào một vị trí tiếp theo trong bảng băm. Ví dụ:



Trong hình minh họa:

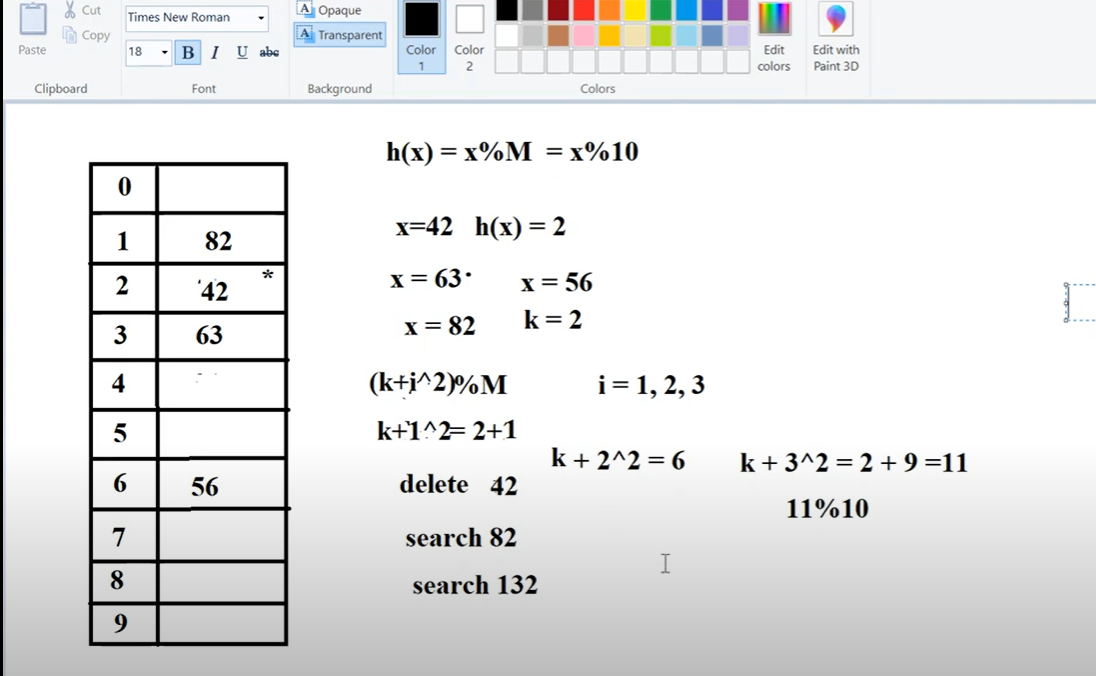
* John Smith và Sandra Dee đều có giá trị Hash là 152. Vì ta đã lưu John Smith vào bucket 152, nên ta buộc phải lưu Sandra Dee vào bucket 153.
* Ted Baker có giá trị Hash là 153, nhưng lúc này bucket 153 đã lưu thông tin của Sandra Dee, nên ta phải lưu giá trị của Ted Baker vào bucket 154.

**Chú ý**:

* Để cài đặt được cách này, Load factor phải nhỏ hơn 1.
* Khi tìm kiếm 1 phần tử, ta phải kiểm tra tất cả các bucket bắt đầu từ bucket của giá trị Hash, đến khi bucket trống (ví dụ ta tìm Sandra Dee thì phải tìm bucket 152, 153; tìm Ted Baker thì phải tìm bucket 152, 153, 154. Nếu ta tìm 1 người khác có giá trị Hash là 152, thì phải tìm cả bucket 152, 153, 154 và 155 (chỉ khi bucket 155 trống, ta mới chắc chắn người đó không có trong Hash table).

**Open addressing methods:**

1. The simplest method is **linear probing***,* for which *p*(*i*) = *i,* and for the *ith* probe, the position to be tried is (*h*(x) + *i*) mod M*, i = 1,2,…*
2. **Quadratic**: p(i) = i2 thus the position to be tried is (*h*(x) + i2) mod M*, i = 1,2,…*



**Factors affecting Search perfomance**

1. Quality of hash function
   1. how uniform?
   2. depends on actual data
2. Collision resolution strategy used
3. Load factor of the HashTable