How fast can we search?

Arrays		Linked Lists		Binary Trees		Hash Tbl	Graphs	
Unsorted	Sorted	Unsorted	Sorted	UnBal.	Bal.		Dense	Sparse
O(n)	O($\log_2 n$)	O(n)	O(n)	O(n)	O($\log_2 n$)	O(1)	O(n^2)	O(n)

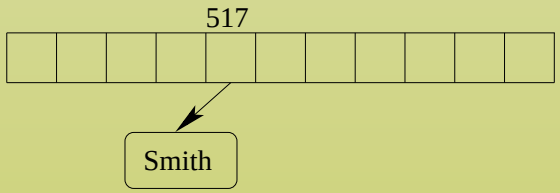
A dictionary is a data structure that supports the operations: insert, delete and search. The efficiency of the operations are given in the table above. Under reasonable assumptions expected time for the search operation in a hash table is constant $O(1)$ but it uses more memory than a regular array.

Hashing

A hash table is an example of a dynamic set which can change with time by adding or deleting items. Each item contains at least one identifying key member.

A hashing is a dictionary supported by the following operations: insert, search and delete based on the value of a key member. The key may be of any type but it is usually an integer or string.

A hash code transforms a key into an integer and a hash function maps it to a corresponding index in the hash table where the item is placed.



Hash codes for strings

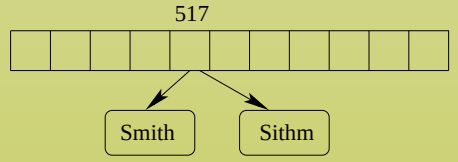
A string is often the search key. Therefore, it is important to have a method generating good hash codes for strings.

The first method is based on a character Unicode/ASCII value. It transforms a string into an integer suitable for further hashing action.

Example:

A last name used as a search key. As an example, say that "Smith" is the key.

The ASCII values of the string characters are: $S = 83$, $m = 109$, $i = 105$, $t = 116$, $h = 104$. The natural hash code for a string is the sum of ASCII values of its characters. That is, the hash code for "Smith" is $83 + 109 + 105 + 116 + 104 = 517$. However, this method can return the same hash code for any permutation of characters in the string (for example, try with "Sithm").



A better approach to obtain hash code for a string is to represent the string as a number written in a special notation.

To explain this, consider the following example. Assume that our strings consist only of capital letters from "A" to "Z". A function *Index* maps each character to a different integer in the range from 1 to 26 ($r = 26$ is the radix). That is,

$Index(A) = 1, Index(B) = 2, Index(C) = 3, \ldots, Index(Z) = 26.$

The characters of any such a string are viewed as digits of a number in the base 26 number system.

If $S = S_1 S_2 \ldots S_N$, then the corresponding number is:

$$Index(S_1) * r^{N-1} + Index(S_2) * r^{N-2} + \ldots + Index(S_{N-1}) * r^1 + Index(S_N) * r^0$$

The above transformation of string to integer is unique. If we have a hash table of infinite size, then this is the perfect hash function for strings.

To compute the value of the hash code more efficiently we will use the Horner's method. This is a perfect hash algorithm which computes the hash code of a string S.

```
/* perfect hash for strings */
   int pHash = 0; r = 26;
   for(int k = 0; k < S.length(S); k++)
      pHash = pHash*r + S.charAt(k);
   return pHash;
```

Example: This is a hash code for the string "ALFA".

$$hashCode(\text{"}ALFA\text{"}) = 1 * 26^3 + 12 * 26^2 + 6 * 26 + 1 * 26^0 = 367909.$$

But in practice we have a limited hash table size, so we need to use the mod function. Let *tableSize* be the size of a hash table. This is how we can compute an index to a hash table.

$$hashFun(\text{"}String\text{"}) = hashCode(\text{"}String\text{"}) \bmod tableSize$$

To find the hash index we use the following modification of the above algorithm (`r` = 26).

```
int hashFun(String S) {
   int pHash = 0;
   for(int k = 0; k < S.length(S); k++)
      pHash = (pHash*r + S.charAt(k)) % tableSize;
   return pHash;
}
```

Example:

Let `TableSize` = 31. Then `hashFun("ALFA")` = 28, since:

```
pHash = 0;
k = 0: pHash = (1 % 31) = 1
k = 1: pHash = (1 * 26 + 12) % 31 = 7
k = 2: pHash = (6 * 26 + 6) % 31 = 7
k = 3: pHash = (7 * 26 + 1) % 31 = 28
```

The hash code for primitive types

If a search key is of type `int` you can use the key itself as the hash code. For other primitive types: `byte, short, char` hash code can be obtained by casting into `int` type.

If a search key is of type `long` (64 bits) then casting to `int` type (32 bits), which corresponds to division by 2^{32}, causes the first 32 bits of the key to be lost.

The simplest hash function which satisfies the condition (*) is:

$$h(k) = k \bmod m$$

where k is the key, m is a size of the hash table and the function mod returns the remainder of division k by m.

Example: Let $m = 6$ and key values be $3, 5, 6, 15$.

index	key
0	6
1	
2	
3	3
4	
5	5

$h(3) = 3 \bmod 6 = 3$

$h(5) = 5 \bmod 6 = 5$

$h(6) = 6 \bmod 6 = 0$

But if we want to add the key 15 there is a collision:

$h(15) = 15 \bmod 6 = 3$

Collisions

In fact, any hash function will generate collisions, when two different keys hash to the same place in the hash table:

$h(k_1) = h(k_2)$ although $k_1 \neq k_2$.

The hash function is not unique because it maps the large domain of key values into a small range of a hash table.

The collisions are resolved using two techniques: open address and chaining.

In the previous example, we can minimize the number of collisions when the size of a hash table is a prime number.

0	
1	15
2	
3	3
4	
5	5
6	6

Let $m = 7$. Then

$h(3) = 3 \bmod 7 = 3$

$h(5) = 5 \bmod 7 = 5$

$h(6) = 6 \bmod 7 = 6$

$h(15) = 15 \bmod 7 = 1$

Hash Function

A good hash function should:

- minimize the number of collisions
- distribute keys uniformly throughout the hash table
- be fast to compute

However, no perfect hash function can be created so it is impossible to get rid of collisions entirely. Examples of hash functions:

- Random Number Generator (RNG).
- the folding method.
- digit or character extraction.
- the division remainder technique (the mod function).

Method 1: RNG

Random Number Generator (RNG) produces random sequences of real or integer values based on a seed parameter. When the seed is fixed, then RNG produces the same sequence of numbers. Here is a C code for RNG.

```
int rand(void); //integer RNG
   //maximum range of numbers is from 0 to RAND_MAX
void srand(unsigned seed); //sets a seed
   //This example produces 10 integer random numbers
int main()
{
   int i;
   srand( time(NULL) );
   for(i = 0; i < 10; i++)
      printf("%d ", rand() );
   return 0;
}
```

We can use RNG as a hash function for the following key values: $3, 5, 8, 10$. Assume that the size of a hash table is 7.

```
int main()
{
   int i;
   int a[4] = {3,5,8,10}; /* key values */
   for(i = 0; i < 4; i++) {
      srand(a[i]);
      printf("%d ", (int)(rand() / (RAND_MAX+1.0) *7);
      printf("\n");
   }
}
```

These indices are generated: $2, 0, 1, 3,$

But if we take the keys: 3, 4, 8, 10, then we get the indices: 2, 1, 1, 3 (collision).

Method 2: The Folding Method

Assume that the key is the Social Security Number (SSN): 158-26-3806.

- Boundary folding: the index is equal to $158 + 26 + 3806 = 3990$
- Shift folding: the index is equal to $158 + 62 + 3806 = 4026$ (26 is reversed).

Method 3: The Extraction Method

- Only part of the key is used for computing the index.
- The key is SSN: 158-26-3068. Then the index = 3068 (the last part of SSN).

Method 4: The Division Remainder Technique

A hash function should produce values between 0 and the size of the hash table so the mod function is well defined for any distinct key.

```
HashFun(key) = key mod tableSize
```

Notice that the mod function is biased when `tableSize` is even. In order to minimize collisions we choose `tableSize` as a prime number and keep the table not too dense. If the table becomes full the probability that collisions will occur increases. Therefore, we need to keep some positions in the table empty; otherwise search efficiency will deteriorate.

Example: The key is the string "Smith" with radix $128 = 2^7$. Then the index = $(83 \cdot 2^{7 \cdot 4} + 109 \cdot 2^{7 \cdot 3} + 105 \cdot 2^{7 \cdot 2} + 116 \cdot 2^7 + 104 \cdot 2^0)$ mod m,

where m is a prime number corresponding to the size of the hash table.

Open Addressing and Linear Collision Processing

- When a collision has occurred during insertion of an object, the essential problem is to develop an algorithm that will find a unoccupied position in a hashing table.
- Such an algorithm should minimize the possibility of future collisions.
- The same algorithm which was used for insertion also should be used for a search and delete operations on the hash table.

Linear collision processing

When a collision occurs, proceed down the array in sequential order until a vacant position is found. The key is then placed at this first vacant position. The hash function is defined by the formula:

If we come to the last element of the array, we wrap around to the first element of the array and continue looking for a vacant position.

Example. Insert the following key values: $4, 7, 8, 15, 11$ in the hash table of size 11using the division method.

index	0	1	2	3	4	5	6	7	8	9	10
key	11				4	15		7	8		

$h(4) = 4 \bmod 11 = 4$, $h(7) = 7 \bmod 11 = 7$

$h(8) = 8 \bmod 11 = 8$, $h(15) = 15 \bmod 11 = 4$ collision – try the next position

$h(11) = 11 \bmod 11 = 0$ the hash function can wrap around

C and Java code for the search operation with resolving collisions by linear probing.

```
void Search(Record
      HashTable[TabSize],
      KeyType Target, Record *Item,
Boolean *success){
   int k, j;
   Boolean prob = false;
   *success = false;
   j = k = Hash(Target);
   while(HashTable[j].key != Empty
         && !(prob || *success))
      if(Target==HashTable[j].key){
         *Item = HashTable[j];
         *success = true;
      } else {
         j = (j+1)%TableSize;
         prob = (j == k);
      }

}
```

```
KC_Record Search(Comparable Target){
   int k, j;
   Boolean prob = false;
   j = k = Hash(Target);
   while(HashTable[j].key != Empty
         && !prob)
      if (HashTable[j].key.
            compareTo(Target) == 0)
         return HashTable[j];
      else {
         j = (j+1)%TableSize;
         prob = (j == k);
      }
   return null; // target not found

}
```

Drawbacks of linear collision processing:

- clustering (primary clustering) grouping keys that have collision one after another in a hashing table.
- expanding clusters and grouping adjacent clusters.

The hash function check the simplest pattern described by the following formula:
$h(k,i) = (h_1(k) + i) \bmod m$ where $h_1(k)$ is the hash function and
$i = 0, 1, 2, 3..$ is called the prob number for key k.

Example. Insert the following key values: $23, 124, 225, 24, 327$ in the hash table of size 101 using the division method.

index	..	..	..	23	24	25	26	27	..	..	..
key	..	..	..	23	124	225	24	327	..	..	..

Efficiency for linear hashing

Let denote density by D = $\frac{Number\ of\ filled\ positions}{TableSize}$. Searching is a density-dependent technique and the average number of array accesses for a successful and unsuccessful searches, respectively:

$$(*) \quad \frac{1}{2}(1+\frac{1}{1-D})$$

$$(**) \quad \frac{1}{2}(1+\frac{1}{(1-D)^2})$$

D	*	**
0.1	1.06	1.18
0.5	1.5	2.5
0.75	2.5	8.5
0.9	5.5	50.5

Open Addressing and Quadratic Collision Processing

To correct the problems with primary clustering caused by linear collision resolution, the quadratic collision processing is proposed. Suppose that

```
hashFun(KeyValue) = j
and a collision results.
```

The hash function for the quadratic probing can be described by the following formula:

$h(k,i) = (h_1(k) + c_2 i^2 + c_1 i + c_0) \text{mod} m$ where $h_1(k)$ is the hash function, in the of case of collision also the quadratic function in i is used and $i = 0, 1, 2, 3..$ is called the prob number for key k.

Assume that the constants are equal $c_2 = 1$, $c_1 = c_0 = 0$. To resolve the collision the following positions are probed in the hash table:

$j+1$, $j+2^2$, $j+3^2$, $j+4^2$, ... , $j+R^2$, ...

Example. Positions of hash table checked by quadratic probing Insert the following key values: $3, 10, 17, 11, 26$ in the hash table of size 7 using the division method.

index	..	j	$j+1$	..	..	$j+2^2$					$j+3^2$	..	..	..
key	..	..	..	..	..						..	..	..	..

If any of these positions is vacant then the key is placed there. If $J = j + R^2$ is greater than `tableSize`, then we wrap it around, that is, J mod `tableSize`.

Example. Insert the following key values: $3, 10, 17, 11, 24$ in the hash table of size 7 using the division method. In case of collisions apply the quadratic probing.

index	0	1	2	3	4	5	6
key	17	..	..	3	10	11	..

$h(3,0) = 3,$

$h(10,0) = 3, h(10,1) = 4$

$h(17,0) = 3, h(17,1) = 4, h(17,2) = 0$

$h(24,0) = 3, h(24,1) = 4, h(24,2) = 0, h(24,3) = 5,$

$h(24,4) = 5$

In the case, the quadratic probing is checking only certain positions in the hash table and there is impossible to insert 24 although the hash table is not full..

Efficiency for quadratic method

Assume that the density D = $\frac{Number-of-filled-positions}{TableSize}$ is less than 1, it means that some slots of a table are empty.

Average search efficiency for the successful and for unsuccessful cases, respectively:

$(*)$ $1 - log_e(1-D) - \frac{D}{2}$

$(**)$ $\frac{1}{1-D} - D - log_e(1-D)$

D	*	**
0.1	1.05	1.11
0.5	1.44	2.19
0.75	2.01	4.64
0.9	2.85	11.4

Drawbacks of quadratic method

- More complex computation of a position to be probed when a collision occurs

  $J = (j + R^2) \text{mod } \texttt{tableSize}$

- No guarantee that we will try every position in the array before finding that a given key cannot be inserted (which was guaranteed by the linear method).
  - Linear and quadratic hashing can face a complication with the search operation after deleting some elements from a hash table.

Example: Consider the following sequence of operations:

1. insert keys k_1, k_2 and k_3 in the positions of the hash table $6, 7$ and 8 respectively
2. delete k_2
3. search for k_3

The search operation cannot find k_3 because the slot 7 is empty and it terminates.

The open addressing should mark positions if the hash table in two different ways:

empty slot – allows to insert a new element and terminates search

available slot – indicates that element used to be there but was deleted and search should be continued.

Double hashing

Similar to quadratic hashing a first hash function h_1 is used to determine where to start probing and in the case of a collision a second function $h_2$is used to determine the prob sequence. The second function can be any function that generates different prob sequences for collide elements.

The search pattern is given by the formula:

$h(k,i) = (h_1(k) + ih_2(k)) \bmod TableSize$

Bucket Hashing

Bucket = contiguous region of locations within the hash array.

Hashing function transforms a given key to a bucket number (not an index to the array).

```
hashFun(k) = k mod number_buckets
```

When `J = hashFun(k)` is computed, then the `Target` is compared sequentially to all the keys in that bucket `J`. It is similar to the linked hash method.

Chaining method

This method does not have the drawbacks of the previous methods, but needs more memory. The hash table is defined as a table of head pointers to linked lists. Initially, the hash function translates keys into the linked list number and all elements that hash to the same table location are placed in the same linked list.

- The insertion operation usually puts a new element at the beginning of the linked list.
- Search operation selects the linked list and search for an element in a given linked list.
- Delete operation is preceded by the search and removing the element from the linked list.

A good hash function should spread all elements evenly among the linked lists and keeps the linked list short with respect to the size of the table.

Efficiency for chaining

Let denote load factor by $D = \frac{Number\ of\ elements}{TableSize}$. Assume that the average length of a linked list is equal to D or wording differently, any key is equally likely to hash to any of the hash table positions.

Average search efficiency for the successful and unsuccessful cases, respectively:

$(*) \quad 1 + \frac{D}{2}$

$(**) \quad 1 + D$

D	*	**
2	2	3
5	3.5	6
10	6	11
20	11	21