DATA STRUCTURES AND ALGORITHMS

Searching algorithms: Hashing

Summary of the previous lecture

Searching algorithms of sorted lists:

- Divide and conquer principle
- Jump search
  - **T(n) = O($\sqrt{n}$.)**
- Binary search
  - **T(n) = O(log n)**
- Interpolation (dictionary) search
  - **T(n) = O(log log n)**

Hashing

One of the possible way to search for element is to use **searching tables** where element is directly accessed based on key value.

Definition
The process of finding a record using some computation to map its key value to a position in the table is called **hashing**.

Most hashing schemes place records in the table in whatever order satisfies the needs of the address calculation, thus the records are not ordered by value or frequency.

Definitions

The function that maps key values to positions is called a **hash function** and is usually denoted by **h**.

The array that holds the records is called the **hash table** and will be denoted by **HT**.

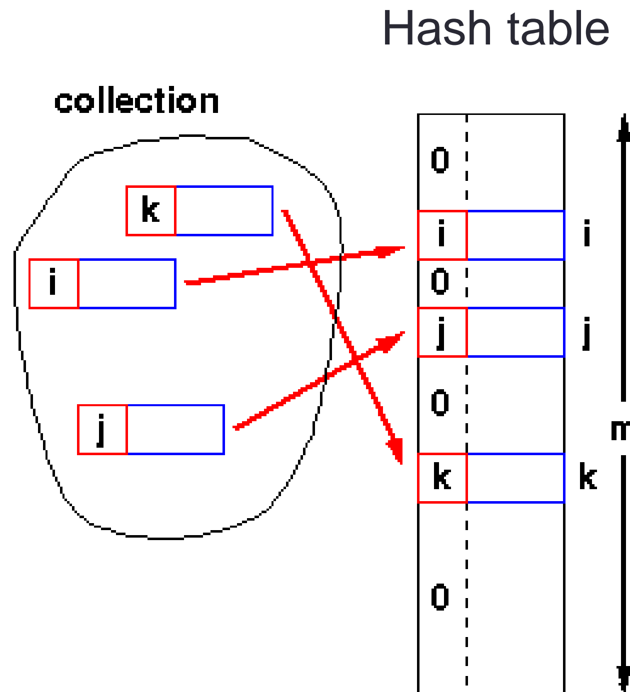
A position in the hash table is also known as a **slot**.

The number of slots in hash table HT will be denoted by the variable **M**, with slots numbered from 0 to M - 1.

The goal for a hashing system is to arrange things such that, for any key value **K** and some hash function **h**, $i = h(K)$ is a slot in the table such that $0 <= h(K) < M$, and we have the key of the record stored at **HT[i]** equal to **K**.

Example

Consider storing collection of **m** records, each with a unique key value in the range [0 ; m – 1].
A record with key **k** can be stored in **HT[k],** and the hash function is simply **h(k) = k**. To find the record with key value **k**, simply look in **HT[k].**



Hashing

- Hashing cannot be used for applications where multiple records with the same key value are permitted.
- Hashing is not a good method for answering range searches. In other words, we cannot easily find all records (if any) whose key values fall within a certain range.

- Hashing is suitable for both in-memory and disk-based searching
- Hashing is one of the two most widely used methods for organizing large databases stored on disk

Hash tables

Typically, there are many more values in the key range than there are slots in the hash table.

We must devise a hash function that allows us to store the records in a much smaller table. Because the possible key range is larger than the size of the table, at least some of the slots must be mapped to from multiple key values.

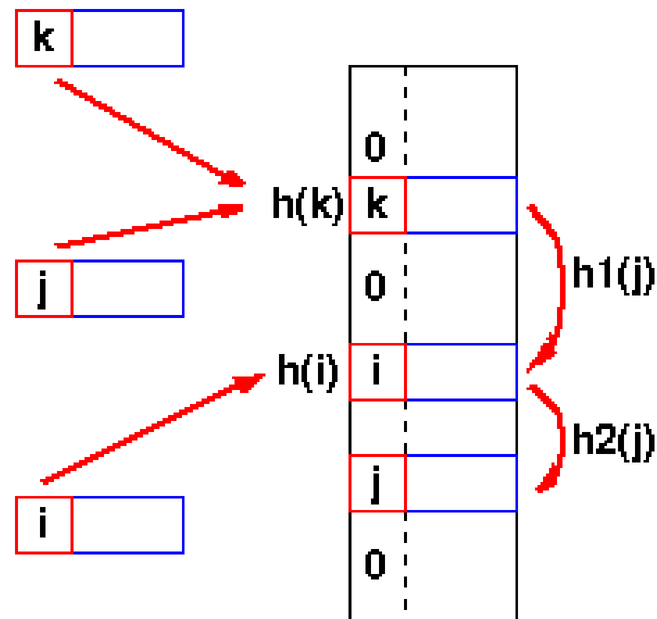
Hash function **h(k)** must allow:

$$h(k1) = h(k2) = x$$

The ratio $\alpha = n/m$ is called a **load factor**, that is, the average number of elements stored in a **HT**, where set size is *n*, **HT** size is *m*.

Collision

A hash function **h** maps the keys *k* and *j* to the same slot, so a **collision** apears.



Example

Keys: 5, 28, 19, 15, 20, 33, 12, 17, 10

HT slots: 9

hash function = h(k) = k % 9

h(5) = 5 % 9 = 5
h(28) = 28 % 9 = 1
h(19) = 19 % 9 = 1
h(15) = 15 % 9 = 6
h(20) = 20 % 9 = 2
h(33) = 33 % 9 = 6
h(12) = 12 % 9 = 3
h(17) = 17 % 9 = 8
h(10) = 10 % 9 = 1

Collision resolution

The goal of a hash function is to minimize collisions (collisions are normally unavoidable in practice).

Thus, hashing implementations must include some form of collision resolution policy.

Collision resolution techniques can be broken into two classes:

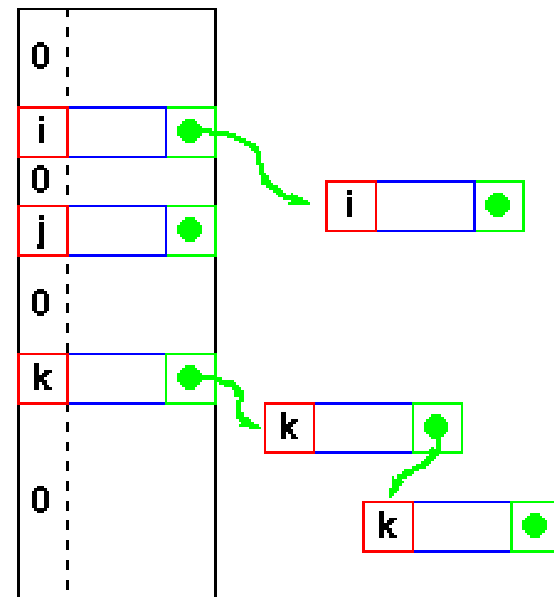
- **open hashing** (also called **separate chaining**)
- **closed hashing** (also called **open addressing**).

The difference between techiques is :

- whether collisions are stored **outside the table** (open hashing),
- whether collisions result in storing one of the records **at another slot in the table** (closed hashing).

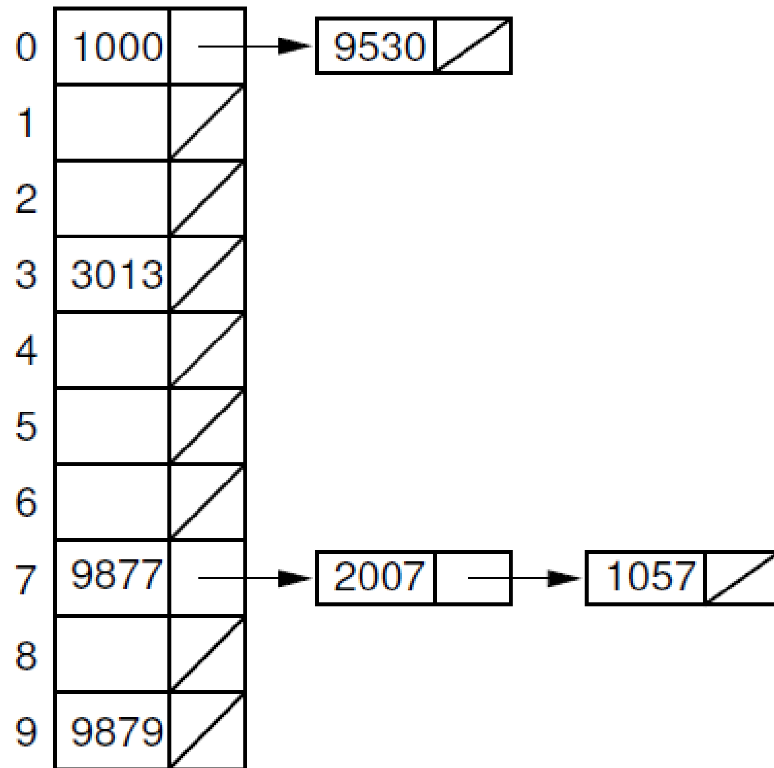
Collision Resolution by Chaining

When there is a collision, the incoming keys is stored in an overflow area and the corresponding record is appeared at the end of the **linked list**. All records that hash to a particular slot are placed on that slot's linked list



Each slot HT[j] contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_n)$ and $h(k_5) = h(k_2) = h(k_7)$.

Example of open hashing



An illustration of open hashing for seven numbers stored in a ten-slot hash table using the hash function h(K) = K % 10

Running time

Running time in **open hashing case**:

- The worst case running time for insertion is O(1).
- The worst case behavior of chain-hashing, all **n** keys hash to the same slot, creating a list of length **n**, so runing time for search is **O(n).**

Hash function

A good hash function satisfies the assumption of simple uniform hashing, each element is equally likely to hash into any of the **m** slots, independently of where any other element has hash to.

When designing hash functions, we are generally faced with one of two situations:

1. We know nothing about the distribution of the incoming keys. In this case, we wish to select a hash function that evenly distributes the key range across the hash table
2. We know something about the distribution of the incoming keys. In this case, we should use a distribution-dependent hash function that avoids assigning clusters of related key values to the same hash table slot. For example, if hashing English words, we should not hash on the value of the first character because this is likely to be unevenly distributed.

Example

Here is a hash function for strings of characters:

```
int h(char* x) {
int i, sum;
for (sum=0, i=0; x[i] != '\0'; i++)
    sum += (int) x[i];
return sum % M;
}
```

This function sums the ASCII values of the letters in a string. If the hash table size **M** is small, this hash function should do a good job of distributing strings evenly among the hash table slots, because it gives equal weight to all characters.

Example

A good hash function for numerical values is the **mid-square** method. The mid-square method squares the key value, and then takes the middle **r** bits of the result, giving a value in the range 0 to 2^r-1.

This works well because most or all bits of the key value contribute to the result.

For example, consider records whose keys are 4-digit numbers in base 10. The goal is to hash these key values to a table of size 100 (i.e., a range of 0 to 99).

If the input is the number 4567, squaring yields an 8-digit number, 20857489. The middle two digits of this result are 57.

Hash function

Usually it is not possible to check this condition because **one rarely knows the probability distribution according to which the keys are drawn**.

In practice, heuristic techniques are used to create a hash function that perform well.

Methods for Creating Hash Function

- The division method
- The multiplication method
- Closed hashing

Division Method

Map a key **K** into one of **m** slots by taking the remainder of **K** divided by **m**.

$$h(k) = k \% m.$$

Example:

m = 12;
key **k** = 100
h(100) = 100 % 12 = 4

Good choice of *m - a* prime number not too close to 2.

Multiplication Method

Two step process:

Step 1:

Multiply the key **k** by a constant $0 < A < 1$ and extract the fraction part of *k*A.

Step 2:

Multiply kA by **m** and take the **floor** of the result (largest integer not greater than *kA)*.

Advantage of this method is that the value of **m** is not critical and can be implemented on most computers.

A reasonable value of constant **A** is ~ (sqrt(5) - 1) /2 suggested by Knuth's Art of Programming.

Closed hashing

In this technique all records are stored in the hash table itself.

That is, each table entry contains either an element or NULL.

Searching for the element (or empty slot):

- systematically examine slots until we found an element (or empty slot). There are no lists and no elements stored outside the table. That implies that table can completely "fill up“ and the load factor α can never exceed 1.

Closed hashing

Insertion algorithm:

- Each record ***R*** with key value ***kR*** has a home position that is **h(kR)**, the slot computed by the hash function.
- If **R** is to be inserted and another record already occupies R's home position, then **R** will be stored at some other slot in the table.

The same principle is used for searching.

Example

	Hash Table
0	1000
	9530
1	
2	9877
	2007
3	3013
4	9879

Overflow
1057

An illustration of bucket hashing for seven numbers stored in a five bucket hash table using the hash function h(K) = K% 5.

Each bucket contains two slots.

Closed hashing

Advantage of closed hashing: avoids pointers (pointers need space too).

Instead of hashing pointers, we compute the sequence of slots to be examined.

To perform insertion, we successively examine or probe, the hash table until we find an empty slot. The sequence of slots probed "depends upon the key being inserted."

To determine which slots to probe, the hash function includes the probe number as on of the input argument of the function.

Linear probing

Turn to the most commonly used form of hashing - **closed hashing** with a collision resolution policy that can potentially use any slot in the hash table.

During insertion, the goal of **collision resolution** is to find a free slot in the hash table when the home position for the record is already occupied.

We can view any collision resolution method as generating a sequence of hash table slots that can potentially hold the record.

The first slot in the sequence will be the **home position** for the key. If the home position is occupied, then the collision resolution policy goes to the next slot in the sequence and so on.

Linear probing

Sequence of slots is known as the probe sequence, and it is generated by some **probe function** that we will call **p**.

Probe function **p** allows us many options for how to do collision resolution. In fact, linear probing is one of the worst collision resolution methods.

Linear probing example

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	1059

Example of problems with linear probing.
Four values are inserted in the order 1001, 9050, 9877, and 2037 using hash function h(K) = K % 10

Example

```
int collision_LinearProbing( int hashIndex, int count, int data)
{
    if (count == hashSize)
    return ( fail );

if ( *(head + hashIndex) == NULL)
{
      *(head+hashIndex) = data;
        return (success);
}
else
{    collision_LinearProbing((++hashIndex % hashSize), ++count, data);
}
}
```

Improved Collision Resolution Methods

One possible improvement might be to use linear probing, but to skip slots by a constant c other than 1. This would make the probe function

$$p(K; i) = ci;$$

and so the ith slot in the probe sequence will be

$(h(K)+ic) \% M$.

In this way, records with adjacent home positions will not follow the same probe sequence.

For example, if we were to skip by twos, then our offsets from the home slot would be 2, then 4, then 6, and so on.

Improved Collision Resolution Methods

Another probe function that eliminates primary clustering is called quadratic probing. Here the probe function is some quadratic function

$$p(K; i) = c_1 i^2 + c_2 i + c_3$$

for some choice of constants $c_1 + c_2 + c_3$.

The simplest variation is $p(K; i) = i^2$. Then the ith value in the probe sequence would be $(h(K) + i^2) \% M$.

Under quadratic probing, two keys with different home positions will have diverging probe sequences.

Exercise

Assume that you have a seven-slot closed hash table (the slots are numbered 0 through 6).

Show the final hash table that would result if you used the hash function h(k) = k % 7 and linear probing on this list of numbers: 3, 12, 9, 2.

After inserting the record with key value 2, list for each empty slot the probability that it will be the next one filled.

The END of theory course!!!