Data Structures and Algorithms (10)

Instructor: Ming Zhang
Textbook Authors: Ming Zhang, Tengjiao Wang and Haiyan Zhao

Higher Education Press, 2008.6 (the "Eleventh Five-Year" national planning textbook)

https://courses.edx.org/courses/PekingX/04830050x/2T2014/

Chapter 10 Retrieval

- 10.1 Retrieval in a linear list
- 10.2 Retrieval in a set
- 10.3 Retrieval in a hash table
- Summary

Basis Concepts

- Retrieval

The process of finding a record with its key value equal to a given value in a set of records, or the records whose keys meet some specific criteria.

- The efficiency of retrieval is very important
  - Especially for big data
  - Need special storage processing for data

Methods of Improving Retrieval Efficiency

- Sorting
- Indexing
- Hashing
- When hashing is not suitable for disk-oriented applications, we can use B trees.

- Take much time
- preprocessing (finished before retrieval)
- disadvantages :
  - Unsuitable for range searches
  - Generally, duplicate keys are not allowed

- Organize the data into a table
- Get the position of records in the table according to the key values

- Make the most of auxiliary index information
- Sacrifice space
- To improve retrieval efficiency

Average Search Length (ASL)

- Comparison of keys: main operation of retrieval
- Average Search Length
  - Average number of comparisons during retrieval
  - The time metric for evaluating retrieval algorithms

$$ASL = \sum_{i=1}^{n} P_i C_i$$

- P_i is probability of searching the *i-th element*

- C_i is the number of comparisons needed to find the i-th element

Other Metrics for Evaluating Retrieval Algorithms

- Considerations when evaluating retrieval algorithms
  - The storage needed
  - Implementation difficulties
  - ...

Thinking

- Assume that a linear list is (a, b, c), and the probabilities of searching a, b, c are 0.4, 0.1, 0.5 respectively
  - What is the ASL of sequential search algorithms? (which means how many times of comparisons of key values are needed to find the specific element on the average)

Chapter 10. Retrieval

Retrieval in a Linear List

- 10.1.1 Sequential search
- 10.1.2 Binary search
- 10.1.3 Blocking search

Sequential Search

- Compare the key values of records in a linear list with the given value one by one
  - If the key value of a record is equal to the given value, the search hits;
  - Otherwise the search misses (cannot find the given value in the end)
- Storage: sequential or linked
- Sorting requirements: none

Sequential Search with Sentinel

```
// Return position of the element if hit; otherwise return 0
template  <class Type>
class Item {
private:
    Type  key;                              // key field
                                            // other fields
public:
    Item(Type value):key(value) {}
    Type getKey() {return key;}             // get the key
    void setKey(Type k){ key=k;}            // set the key
};
vector<Item<Type>*>  dataList;
template  <class Type>  int SeqSearch(vector<Item<Type>*>& dataList, int
length, Type k) {
    int i=length;
    dataList[0]->setKey (k);                // set the 0th element as the element
                                            to be searched, set the lookout
    while(dataList[i]->getKey()!=k) i--;
    return i;                               // return the position of the element
}
```

Performance Analysis of the Sequential Search

- Search hits: assume the probability of searching any key value is uniform: $P_i = 1/n$

$$\sum_{i=0}^{n-1} P_i \cdot (n - i) = \frac{1}{n}\sum_{i=0}^{n-1}(n - i)$$

$$= \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$$

- Search misses: assume that n+1 times of comparisons are needed when the search misses (with a sentinel)

Average Search Length of Sequential Search

- Assume the probability of search hit is p, and the probability of search miss is q=(1-p)

$$\begin{aligned}\mathrm{ASL} &= p\cdot\frac{n+1}{2}+q\cdot(n+1)\\ &= p\cdot\frac{n+1}{2}+(1-p)(n+1)\\ &= (n+1)(1-p/2)\end{aligned}$$

- $(n+1)/2 < ASL < (n+1)$

Pros and Cons of Sequential Search

- Pros: insertion in $\Theta(1)$ time
  - We can insert a new element into the tail of list
- Cons: search in $\Theta(n)$ time
  - Too time-consuming

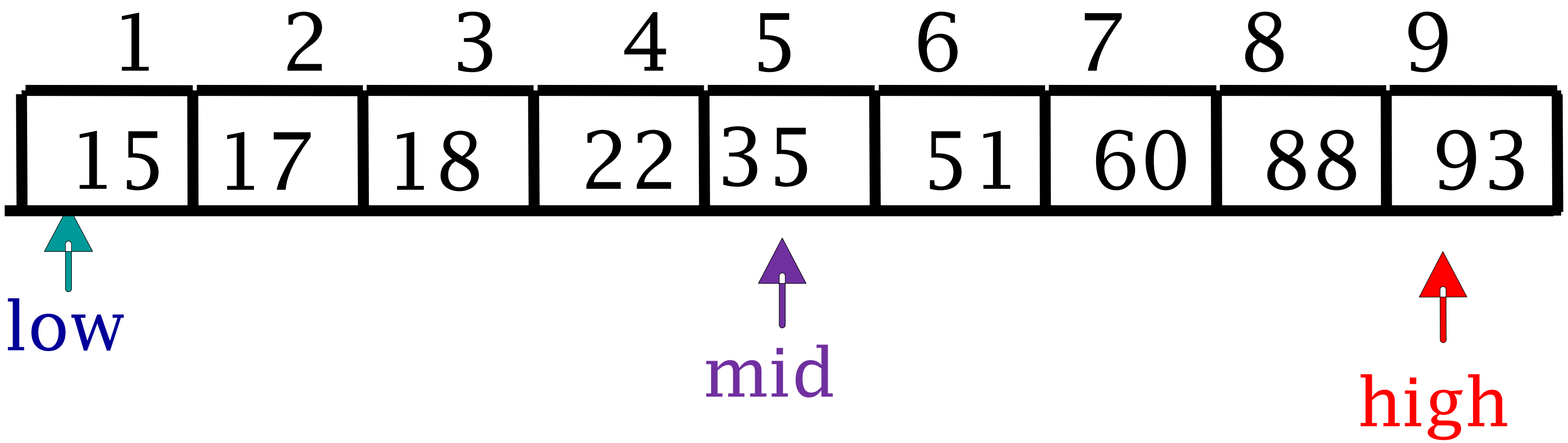
Binary Search

- Compare any element dataList[i].Key with the given value K, there are three situations:
  - (1) Key = K, succeed, return dataList[i]
  - (2) Key > K, the element to find must be before dataList[i] if exists
  - (3) Key < K, the element to find must be after dataList[i] if exists
- Reduce the range of latter search

Binary Search Algorithm

```
template  <class Type> int BinSearch (vector<Item<Type>*>& dataList, int
length, Type k){
    int low=1, high=length, mid;
    while (low<=high)  {
        mid=(low+high)/2;
        if (k<dataList[mid]->getKey())
            high = mid-1;                  // drop the right half of the search range
        else if (k>dataList[mid]->getKey())
            low = mid+1;                   // drop the left half of the search range
       else return mid;                    // return if succeeds
    }
    return 0;                              // if fails, return 0
}
```

Key value 18 low=1 high=9



The first time: l=1, h=9, mid=5; array[5]=35>18

The second time: l=1, h=4, mid=2; array[2]=17<18

The third time: l=3, h=4, mid=3; array[3]=18 = 18

Performance Analysis of the Binary Search

- Maximum search length is

$$\lceil \log_2(n+1) \rceil$$

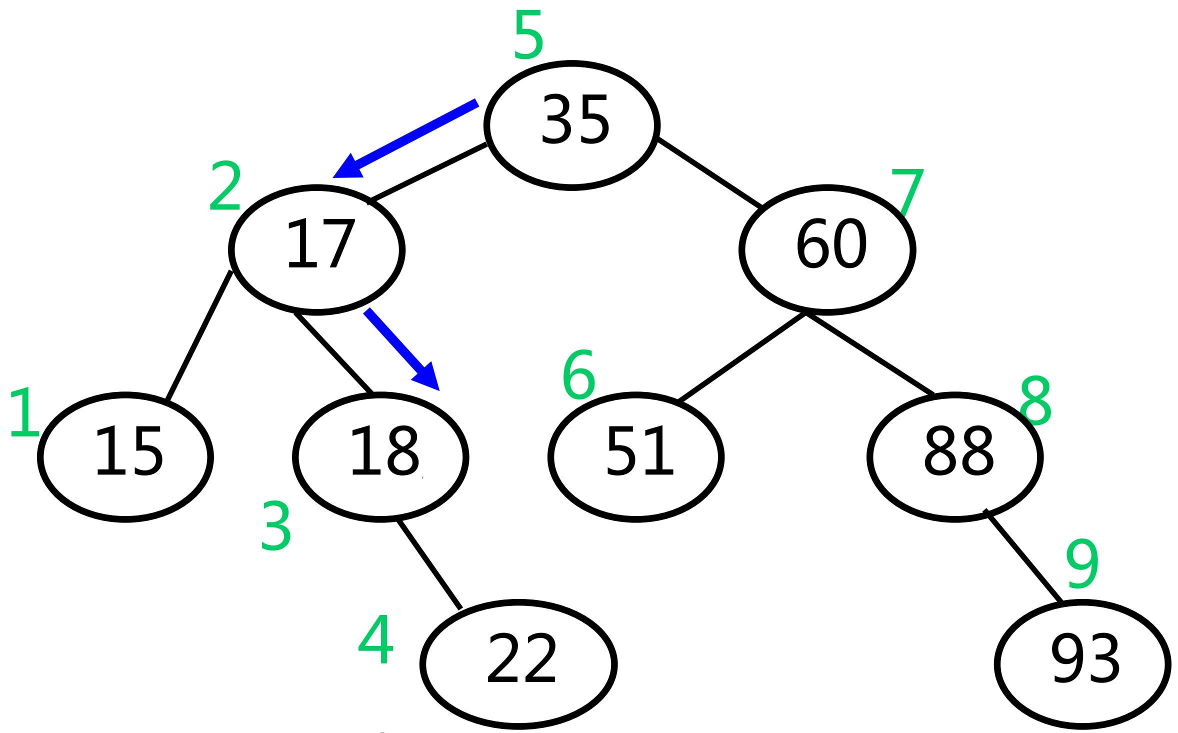
- Failed search length is

$$\lceil \log_2(n+1) \rceil$$

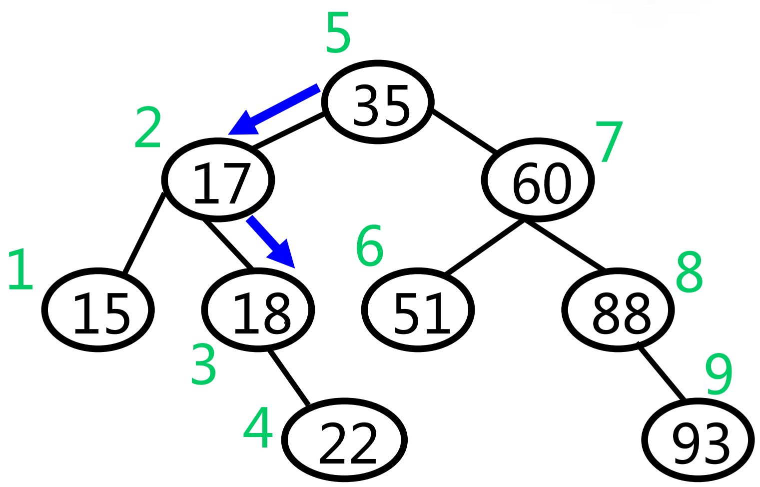
Or

$$\lfloor \log_2(n+1) \rfloor$$

- In the complexity analysis
  - The logarithm base is 2
  - When the log base changes, the order of complexity will not change



Performance Analysis of the Binary Search



- ASL of successful search is:

$$\mathrm{ASL} = \frac{1}{n}\left(\sum_{i=1}^{j} i \cdot 2^{i-1}\right)$$

$$= \frac{n+1}{n}\log_2(n+1) - 1$$

$$\approx \log_2(n+1) - 1 \qquad (n > 50)$$

- Pros: the average and maximum search length is in the same order, and the retrieval is very fast
- Cons: need sorting, sequential storage, difficult to update (insertion/deletion)

Ideas of the Blocking Search

- "Ordering between blocks"
  - Assume that the linear list contains n data element, split it into b blocks
  - The maximum element in any block must be smaller than the minimum element in the next block
  - Keys of elements are not always ordered in one block
- Tradeoff between sequential and binary searches
  - Not only fast
  - But also enables flexible update

Blocking Search – Index Sequential Structure

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
22	12	13	9	8		33	42	44	24	48		60	80	74	49	86	53

- Link: starting position of a block
- Key: Maximum key value in the block
- Count: #elements in a block

link:	**0**	**6**	**12**
Key:	**22**	**48**	**86**
count:	**5**	**5**	**6**

Performance Analysis of Blocking Search

- Blocking search is a two-level search
  - First, find the block where the specific element stays at, with ASL_b
  - Second, find the specific element inside that block, with ASL_w

$$ASL = ASL_b + ASL_w$$
$$\approx \log_2 (b+1)-1 + (s+1)/2$$
$$\approx \log_2(1+n/s) + s/2$$

Performance Analysis of Blocking Search

- If we use sequential search in both the index table and the blocks

$$\mathrm{ASL}_b = \frac{b+1}{2} \qquad \mathrm{ASL}_w = \frac{s+1}{2}$$

$$\mathrm{ASL} = \frac{b+1}{2} + \frac{s+1}{2} = \frac{b+s}{2} + 1$$

$$= \frac{n+s^2}{2s} + 1$$

- When s = $\sqrt{n}$, we obtain the minimum ASL:

$$\mathrm{ASL} = \sqrt{n} + 1 \approx \sqrt{n}$$

Performance Analysis of Blocking Search

- When n=10,000,
  - Sequential search takes 5,000 comparisons
  - Binary search takes 14 comparisons
  - Block search takes 100 comparisons

Pros & Cons of Blocking Search

- Pros:
  - Easy to insert and delete
  - Few movement of records
- Cons:
  - Space of a auxiliary array is needed
  - The blocks need to be sorted at the beginning
  - When a large number of insertion/deletion are done, or nodes are distributed unevenly, the efficiency will decrease.

Thinking

- Try comparing the sequential search witch binary search in terms of advantages and disadvantages.

- What are the application scenes of these retrieval methods respectively?

Chapter 10 Retrieval

- 10.1 Retrieval in a linear list
- 10.2 Retrieval in a set
- 10.3 Retrieval in a hash table
- Summary

Set

- Set: a collection of well defined and distinct objects
- Retrieval in a set: confirm whether a specific element belongs to the set

10.2 Retrieval in a Set

	Names	Math Symbols	Computer symbols
Arithmetic operations	union	$\cup$	+, |, *OR*
	intersection	$\cap$	*, &, *AND*
	complement	$-$	$-$
	equality	$=$	==
	inequality	$\neq$	!=
Logic operations	subset	$\subseteq$	<=
	superset	$\supseteq$	>=
	proper subset	$\subset$	<
	proper superset	$\supset$	>
	element of	$\in$	IN, at

Abstract Data Type of Sets

```
template<size_t N>                      // N is the number of elements of the set
class mySet {
public:
        mySet() ;                       // constructor
        mySet(ulong X);
        mySet<N>& set();                // set attributes of the set
        mySet<N>& set(size_t P, bool X = true);
        mySet<N>& reset();              // clear the set
        mySet<N>& reset(size_t P);      // delete the element p
        bool at(size_t P) const;        // belong operation
        size_t count() const;           // get the count of elements of the set
        bool none() const;              // check whether the set is empty
```

Abstract Data Type of Sets

```
bool operator==(const mySet<N>& R) const;          // equal
bool operator!=(const mySet<N>& R) const;          // not equal
bool operator<=(const mySet<N>& R) const;          // be subset of
bool operator< (const mySet<N>& R) const;          // be proper subset of
bool operator>=(const mySet<N>& R) const;          // be superset of
bool operator>(const mySet<N>& R) const;           // be proper superset of

friend mySet<N> operator&(const mySet<N>& L, const mySet<N>& R);  // union
friend mySet<N> operator|(const mySet<N>& L, const mySet<N>& R);  // intersection
friend mySet<N> operator-(const mySet<N>& L, const mySet<N>& R);  // complement
friend mySet<N> operator^(const mySet<N>& L, const mySet<N>& R);  // xor
};
```

Retrieval in a Set

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0

- Bitmap representation
  - Suitable when the number of valid elements is close to all the possible elements

Example: Find the Odd Primes between 0 and 15

Odd:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

&

Prime :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0

||

Odd prime :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	0

Example: Represent a set by an UnsignedIinteger

- The complete set is a set with 40 elements
- The set {35, 9, 7, 5, 3, 1} can be represented with 2 ulongs.

0000 0000 0000 0000 0000 0000 0000 1000
0000 0000 0000 0000 0000 0010 1010 1010

Since 40 < 64, the size of 2 ulongs, we pad 0's on the left

10.2 Retrieval in a Set

```
typedef unsigned long ulong;
enum {
    // number of bits of a unsigned long
    NB = 8 * sizeof (ulong),
    // The subscript of the last element of the array
    LI = N == 0 ? 0 : (N - 1) / NB
};
    // the array used for saving the bit vector
ulong A[LI + 1];
```

Set the Elements of the Set

```
template<size_t N>
mySet<N>& mySet<N>::set(size_t P, bool X) {
        if (X)          // If X is true , the corresponding value of the bit vector should be set to 1
                A[P / NB] |= (ulong)1 << (P % NB);
                // a Union operation is operated for the element that  corresponds to p
        else    A[P / NB] &= ~((ulong)1 << (P % NB));
                //If X is false , the corresponding value of the bit vector should be set to 0
        return (*this);
}
```

Intersection Operations of a set “&”

```
template<size_t N>
mySet<N>& mySet<N>::operator&=(const mySet<N>& R)
{  // assignment of intersection
    for (int i = LI; i >= 0; i--)       // from low bits to high bits
        A[i] &= R.A[i];                 // intersect bit by bit in the unit
of ulongs
    return (*this);
}
template<size_t N>
mySet<N> operator&(const mySet<N>& L, const mySet<N>& R)
{   //intersection
    return (mySet<N>(L) &= R);
}
```

Thinking

- What else can we use to implement a set?
- Survey various implementations of set in the STL library.

Chapter 10. Retrieval

- 10.1 Retrieval in a linear list
- 10.2 Retrieval in a set
- 10.3 Retrieval in a hash table
- Summary

Retrieval in a Hash Table

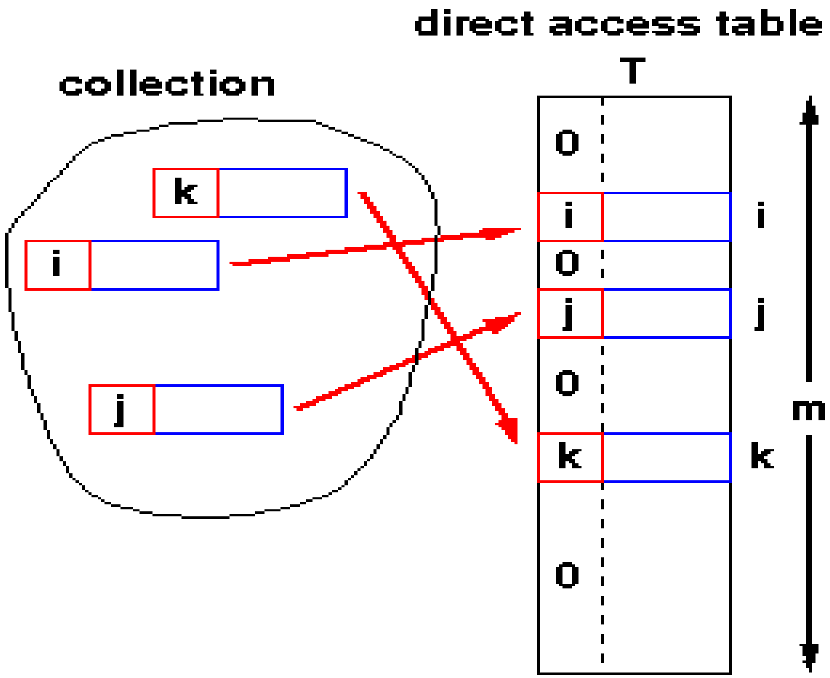
- 10.3.0 Basic problems in hash tables
- 10.3.1 Collision resolution
- 10.3.2 Open hashing
- 10.3.3 Closed hashing
- 10.3.4 Implementation of closed hashing
- 10.3.5 Efficiency analysis of hash methods

Basic problems in Hash Tables

- Retrieval based on comparison of keys
  - Sequential search: ==, !=
  - Binary search, tree based: >, == , <
- Retrieval is the operation interfaced with users
- When the problem size is large, the time efficiency of retrieval methods mentioned above may become intolerable for users
- In the best case
  - Find the storage address of the record according to the key
  - No need to compare the key with candidate records one by one.

Think of Hash from Direct Access

- For example, we can get the element in an array with a specific subscript
  - Inspired by this, computer scientists invented hash method.
- A certain function relation h()
  - Keys of nodes k are used as independent variables
  - Function value h(K) is used as the storage address of the node
- Retrieval uses this function to calculate the storage address
  - Generally, a hash table is stored in a one-dimensional array
  - The hash address is the array index



Example 1

Example 10.1: you already know the set of the key of a linear list: S = {and, array, begin, do, else, end, for, go, if, repeat, then, until, while, with}

We can let the hash table be: char HT2[26][8];

The value of hash function H(key), is the sequence number of the first letter of key in the alphabet {a, b, c, ..., z}, which means H(key) = key[0] – 'a'

Example 1 (continued)

Hash address	key
0	(and, array)
1	begin
2	
3	do
4	(end, else)
5	for
6	go
7	
8	if
9	
10	
11	

Hash address	key
13	
14	
15	
16	
17	repeat
18	
19	then
20	until
21	
22	(while, with)
23	
24	

Example 2

```
// the value of hash function is the average of the sequence numbers of
the first and the last letters of key in the alphabet. Which means:
int H3(char key[])
{
    int i = 0;
    while ((i<8) && (key[i]!='\0'))  i++;
    return((key[0] + key(i-1) - 2*'a') /2 )
}
```

Example 2 (continued)

Hash address	key
0	
1	and
2	
3	end
4	else
5	
6	if
7	begin
8	do
9	
10	go
11	for

Hash address	key
13	while
14	with
15	until
16	then
17	
18	repeat
19	
20	
21	
22	
23	
24	

Several Important Concepts

- The load factor $\alpha = N/\mathrm{M}$
  - M is the size of the hash table
  - N is the number of the elements in the table
- Collision
  - Some hash function return the same value for 2 or more distinct keys
  - In practical application, there are hardly any hash functions without collision
- Synonym
  - The two keys that collides with each other

Hash Function

- Hash function: the function mapping keys to storage addresses, generally denoted by h
- $Address = Hash(key)$
- Principles to select hash functions
  - Be easy to compute
  - The range of the function must be inside the range of the hash table
  - Try to map two distinct keys to different addresses as good as possible.

Various Factors Needed to be Consider

- Lengths of keys
- Size of hash tables
- Distribution of keys
- Frequency rate of searching for records
- …

Commonly-Used Hash Functions

- 1. Division method
- 2. Multiplication method
- 3. Middle square method
- 4. Digit analysis method
- 5. Radix conversion method
- 6. Folding method
- 7. ELF hash function

1. Division method

- Division method: divide M by key x, and take the remainder as the hash address, the hash function is:

$$h(x) = x \bmod M$$

- Usually choose a prime as M
  - The value of function relies on all the bits of independent variable x, not only right-most k bits.
  - Increase the probability of evenly distribution
  - For example, 4093

Why isn't M an even integer?

- If set M as an even integer?
  - If x is an even integer, h(x) is even too.
  - If x is an odd integer, h(x) is odd too;
- Disadvantages: unevenly distribution
  - If even integers occur more often than odd integers, the function values would not be evenly distributed
  - Vice versa

M shouldn't be a Power of Integers

$x \bmod 2^8$ choose right-most 8 bits

0110010111000011010

- If set M as a power of 2
  - Then, $h(x) = x \bmod 2^k$ is merely right-most k bits of x (represented in binary form)
- If set M to a power of 10
  - Then, $h(x) = x \bmod 10^k$ is merely right-most k bits of x (represented in decimal)
- Disadvantages: hashed values don't rely on the total bits of x

Problems of Division Method

- The potential disadvantages of division method
  - Map contiguous keys to contiguous values
- Although ensure no collision between contiguous keys
- Also means they must occupy contiguous cells
- May decrease the performance of hash table

2. Multiplication method

- Firstly multiply *key* by a constant A (0 < A < 1), extract the fraction part
- Then multiply it by an integer n, then round it down, and take it as the hash address
- The hash function is:
  - $hash(key) = \lfloor n * (A * key \% 1) \rfloor$
  - *"A * key % 1" denotes extracting the fraction part of A * key*
  - $A * key \% 1 = A * key - \lfloor A * key \rfloor$

Example

- let key = 123456, n = 10000 and let A = = 0.6180339 ,
- Therefore,

$$hash(123456) = (\sqrt{5}-1)/2$$
$$= \lfloor 10000*(0.6180339*123456 \% 1) \rfloor =$$
$$= \lfloor 10000 * (76300.0041151... \% 1) \rfloor =$$
$$= \lfloor 10000 * 0.0041151... \rfloor = 41$$

Consideration about the Parameter Chosen in Multiplication Method

- If the size of the address space is p-digit then choose $n = 2^p$
  - The hash address is exactly the left-most p bits of the computed value
  - $A * key \% 1 = A * key - \lfloor A * key \rfloor$
  - Advantages: not related to choose of n
- Knuth thinks: A can be any value, it's related to the features of data waited to be sort. Usually golden section is the best

3. Middle Square Method

- Can use middle square method this moment: firstly amplify the distinction by squaring keys, then choose several bits or their combination as hash addresses.
- For example
  - A group of binary key: (00000100 , 00000110 , 000001010 , 000001001 , 000000111)
  - Result of squaring: (00010000 , 00100100 , 01100010 , 01010001 , 00110001)
  - If the size of the table is 4-digit binary number, we can choose the middle 4 bits as hash addresses: (0100, 1001, 1000, 0100, 1100)

4. Digit Analysis Method

- If there are n numbers, each with d digits and each digit can be one of r different symbols
- The occurring probabilities of these r symbols may are different
  - Distribution on some digits may be the same for the probabilities of all the symbols
  - Uneven on some digits, only some symbols occur frequently.
- Based on the size of the hash table, pick evenly distributed digits to form a hash address

Digit Analysis Method (2/4)

- The evenness of distribution of each digit λ_k

$$\lambda_k = \sum_{i=1}^{r} (\alpha_i^k - n/r)^2$$

  - α_i^k denotes the occurring number of ith symbols
  - n/r denotes expected value of all the symbols occurring on n digits evenly
- The smaller λ_k get, the more even the distribution of symbols on this digit is

Digit Analysis Method (3/4)

- If the range of hash table address is 3 digits, then pick the ④ ⑤ ⑥ digits of each key to form the hash address of the record
- We can add ①, ②, ③ digits to ⑤ digit, get rid of the carry digit, to become a 1-digit number. Then combine it with ④, ⑥ digits, to form a hash address. Some other methods also work

9	9	2	1	4	8
9	9	1	2	6	9
9	9	0	5	2	7
9	9	1	6	3	0
9	9	1	8	0	5
9	9	1	5	5	8
9	9	2	0	4	7
9	9	0	0	0	1
①	②	③	④	⑤	⑥

①digit, $\lambda_1 = 57.60$
②digit, $\lambda_2 = 57.60$
③digit, $\lambda_3 = 17.60$
④digit, $\lambda_4 = 5.60$
⑤digit, $\lambda_5 = 5.60$
⑥digit, $\lambda_6 = 5.60$

Digit Analysis Method (4/4)

- Digit analysis method is only applied to the situation that you know the distribution of digits on each key previously
  - It totally relies on the set of keys
- If the set of keys changes, we need to choose again

5. Radix Conversion Method

- Regard keys as numbers using another radix.
- Then convert it to the number using the original radix
- Pick some digits of it as a hash address
- Usually choose a bigger radix as converted radix, and ensure that they are inter-prime.

Example: Radix Conversion Method

- For instance, give you a key $(210485)_{10}$ in base-10 system, treat it as a number in base-13 system, then convert it back into base-10 system
- $(210485)_{13}$

$$= 2\times13^5 + 1\times13^4 + 4\times13^2 + 8\times13 + 5$$

$$= (771932)_{10}$$

- If the length of hash table is 10000, we can pick the lowest 4 digits 1932 as a hash address

6. Folding Method

- The computation becomes slow if we use the middle square method on a long number
- Folding method
  - Divide the key into several parts with same length (except the last part)
  - Then sum up these parts (drop the carries) to get the hash address
- Two method of folding:
  - Shift folding — add up the last digit of all the parts with alignment
  - Boundary folding — each part doesn't break off, fold to and fro along the boundary of parts, then add up these with alignment, the result is a hash address

Example: Folding Method

- [example 10.6] If the number of a book is 04-42-20586-4

(a) shift holding	(b) Boundary holding
5 8 6 4	0 4 4 2 2 0 5 8 6 4
4 2 2 0	0 2 2 4 4 0
+ 0 4	+ 0 4
[1] 0 0 8 8	6 0 9 2
h(key)=0088	h(key)=6092

- (a) shift holding (b) Boundary holding

7. ELF hash function

- Used in the UNIX System V4.0 “Executable and Linking Format(ELF for short)
-

```
int ELFhash(char* key) {
    unsigned long h = 0;
    while(*key) {
        h = (h << 4) + *key++;
        unsigned long g = h & 0xF0000000L;
        if (g) h ^= g >> 24;
        h &= ~g;
    }
    return h % M;
}
```

Features of ELF hash function

- Work well for both long strings and short strings
- Chars of a string have the same effect
- The distribution of positions in the hash table is even.

Application of Hash Functions

- Choose appropriate hash functions according to features of keys in practical applications
- Someone have used statistical analysis method of "roulette" to analyze them by simulation, and it turns out that the middle square is closest to "random"
  - If the key is not a integer but a string, we can convert it to a integer, then apply the middle square method

Thinking

- Consider when using hash methods:

  (1) how to construct (choose) hash functions to make nodes distributed evenly

  (2) Once collision occurs, how to solve it?

- The organization methods of the hash table itself

Chapter 10. Retrieval

- 10.1 Retrieval in a linear list
- 10.2 Retrieval in a set
- 10.3 Retrieval in a hash table
- Summary

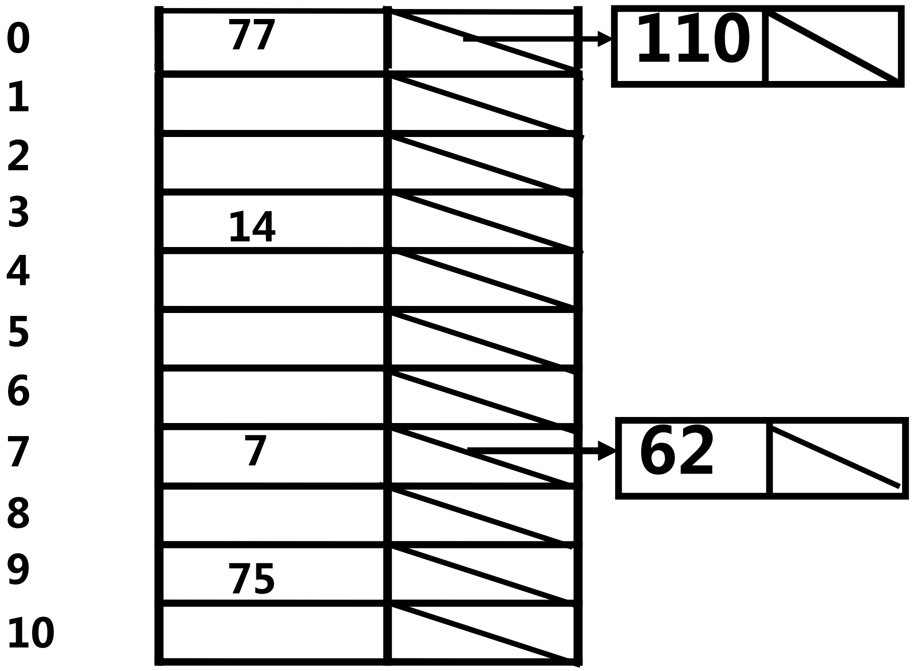
Retrieval in a Hash Table

- 10.3.0 Basic problems in hash tables
- 10.3.1 Collisions resolution
- 10.3.2 Open hashing
- 10.3.3 Closed hashing
- 10.3.4 Implementation of closed hashing
- 10.3.5 Efficiency analysis of hash methods

Open Hashing

{77、14、75、 7、110、62 、95 }

■h(Key) = Key % 11



- The empty cells in the table should be marked by special values
  - like -1 or INFINITY
  - Or make the contents of hash table to be pointers, and the contents of empty cells are null pointers

Performance Analysis of Chaining Method

- Give you a table of size M which contains n records. The hash function (in the best case) put records evenly into the M positions of the table which makes each chain contains n/M records on the average
  - When M>n, the average cost of hash method is $\Theta(1)$

10.3.3 Closed Hashing

- d_0=h(K) is called the base address of K.
- When a collision occurs, use some method to generate a sequence of hash addresses for key K

  d_1, d_2, … d_i , …, d_{m-1}
  - All the d_i (0<i<m) are the successive hash addresses
- With different way of probing, we get different ways to resolve collisions.
- Insertion and retrieval function both assume that the probing sequence for each key has at least one empty cell
  - Otherwise it may get into a endless loop
- We can also limit the length of probing sequence

Problem may Arise - Clustering

- Clustering
  - Nodes with different hash addresses compete for the same successive hash address
  - Small clustering may merge into large clustering
  - Which leads to a very long probing sequence

Several General Closed Hashing Methods

- 1. Linear probing
- 2. Quadratic probing
- 3. Pseudo-random probing
- 4. Double hashing

1. Linear probing

- Basic idea:
  - If the base address of a record is occupied, check the next address until an empty cell is found
    - Probe the following cells in turn: d+1, d+2, ……, M-1, 0, 1, ……, d-1
  - A simple function used for the linear probing:

    p(K,i) = I
- Advantages:
  - All the cell of the table can be candidate cells for the new record inserted

Instance of Hash Table

- M = 15, h(key) = key%13
- In the ideal case, all the empty cells in the table should have a chance to accept the record to be inserted
  - The probability of the next record to be inserted at the 11th cell is 2/15
  - The probability to be inserted at the 7th cell is 11/15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
26	25	41	15	68	44	6				36		38	12	51

Enhanced Linear Probing

- Every time skip constant c cells rather than 1
  - The ith cell of probing sequence is
    $(\mathrm{h}(K) + ic) \bmod M$
  - Records with adjacent base address would not get the same probing sequence
- Probing function is $\mathrm{p}(K,i) = i*c$
  - *Constant c and M must be co-prime*

Example: Enhance Linear Probing

- For instance, c = 2, The keys to be inserted, k_1 and k_2. h(k_1) = 3, h(k_2) = 5
- Probing sequences
  - The probing sequence of k_1: 3, 5, 7, 9, ...
  - The probing sequence of k_2: 5, 7, 9, ...
- The probing sequences of k_1 and k_2 are still intertwine with each other, which leads to clustering.

2. Quadratic probing

- Probing increment sequence: 1^2, -1^2, 2^2, -2^2, ..., The address formula is

  $d_{2i-1} = (d + i^2) \% M$

  $d_{2i} = (d - i^2) \% M$

- A function for simple linear probing :

  $p(K, 2i-1) = i*i$

  $p(K, 2i) = - i*i$

Example: Quadratic Probing

- Example: use a table of size M = 13
  - Assume for k_1 and k_2, $h(k_1)=3$, $h(k_2)=2$
- Probing sequences
  - The probing sequence of k_1: 3, 4, 2, 7, ...
  - The probing sequence of k_2: 2, 3, 1, 6, ...
- Although k_2 would take the base address of k_1 as the second address to probe, but their probing sequence will separate from each other just after then

3. Pseudo-Random Probing

- Probing function p(K,i) = perm[i - 1]
  - here perm is an array of length M – 1
  - It contains a random permutation of numbers between 1 and M

```
// generate a pseudo-random permutation of n numbers
void permute(int *array, int n) {
    for (int i = 1;  i <= n;  i ++)
        swap(array[i-1], array[Random(i)]);
}
```

Example: Pseudo-Random Probing

- Example: consider a table of size M = 13, perm[0] = 2, perm[1] = 3, perm[2] = 7.
  - Assume 2 keys k_1 and k_2, $h(k_1)=4$, $h(k_2)=2$
- Probing sequences
  - The probing sequence of k_1: 4, 6, 7, 11, ...
  - The probing sequence of k_2: 2, 4, 5, 9, ...
- Although k_2 would take the base address of k_1 as the second address to probe, but their probing sequence will separate from each other just after then

Secondary Clustering

- Eliminate the primary clustering
  - Probing sequences of keys with different base address overlap
  - Pseudo-random probing and quadratic probing can eliminate it
- Secondary clustering
  - The clustering is caused by two keys which are hashed to one base address, and have the same probing sequence
  - Because the probing sequence is merely a function that depends on the base address but not the original key.
  - Example: pseudo-random probing and quadratic probing

4. Double Probing

- Avoid secondary clustering
  - The probing sequence is a function that depends on the original key
  - Not only depends on the base address
- Double probing
  - Use the second hash function as a constant
    - $p(K, i) = i * h_2(key)$
  - Probing sequence function
    - $d = h_1(key)$
    - $d_i = (d + i\, h_2(key)) \% M$

Basic ideas of Double Probing

- The double probing uses two hash functions h_1 and h_2
- If collision occurs at address h_1(key) = d, then compute h_2(key), the probing sequence we get is :

  $(d+h_2(key))$ % M , $(d+2h_2$ (key)) %M , $(d+3h_2$ (key)) % M , ...
- It would be better if h_2 (key) and M are co-prime
  - Makes synonyms that cause collision distributed evenly in the table
  - Or it may cause circulation computation of addresses of synonyms
- Advantages: hard to produce “clustering”
- Disadvantages: more computation

Method of choosing M and h2(k)

- Method1: choose a prime M, the return values of h_2 is in the range of

  $1 \le h2(K) \le M - 1$
- Method2: set $M = 2^m$,let h_2 returns an odd number between 1 and 2^m
- Method3: If M is a prime, $h_1(K)$ = K mod M
  - h_2 (K) = K mod(M-2) + 1
  - or h_2(K) = [K / M] mod (M-2) + 1
- Method4: If M is a arbitrary integer, h_1(K) = K mod p (p is the maximum prime smaller than M)
  - h_2 (K) = K mod q + 1 (q is the maximum prime smaller than p)

Thinking

- When inserting synonyms, how to organize synonyms chain?
- What kind of relationship do the function of double hashing h_2 (key) and h_1 (key) have?

Chapter 10. Retrieval

- 10.1 Retrieval in a list
- 10.2 Retrieval in a set
- 10.3 Retrieval in a hash table
- Summary

Retrieval in a Hash Table

- 10.3.0 Basic problems in hash tables
- 10.3.1 Collision resolution
- 10.3.2 open hashing
- 10.3.3 closed hashing
- 10.3.4 Implementation of closed hashing
- 10.3.5 Efficiency analysis of hash methods

Implementation of Closed Hashing

Dictionary

- A special set consisting of elements which are two-tuples (key, value)
  - The keys should be different from each other (in a dictionary)
- Major operations are insertions and searches according to keys
  - **bool hashInsert(const Elem&);**
    // insert(key, value)
  - **bool hashSearch(const Key& , Elem&) const;**
    // lookup(key)

ADT of Hash Dictionaries (attributes)

```
template <class Key , class Elem , class KEComp , class
EEComp> class hashdict
{
private:
    Elem* HT;                    // hash table
    int M;                       // size of hash table
    int currcnt;                 // current count of elements
    Elem EMPTY;                  // empty cell
    int h(int x) const ;         // hash function
    int h(char* x)const ;        // hash function for strings
    int p(Key K , int i)         // probing function
```

ADT of Hash Dictionaries (methods)

```
public:
hashdict(int sz , Elem e) {                    // constructor
    M=sz;  EMPTY=e;
    currcnt=0;  HT=new Elem[sz];
    for (int i=0; i<M; i++) HT[i]=EMPTY;
}
~hashdict() { delete [] HT; }
bool hashSearch(const Key& , Elem&) const;
bool hashInsert(const Elem&);
Elem hashDelete(const Key& K);
int size() { return currcnt; }                 // count of elements
};
```

Insertion Algorithm

hash function h, assume k is the given value

- If this address hasn't been occupied in the table, insert the record waiting for insertion into this address
- If the value of this address is equal to K, report "hash table already have this record"
- Otherwise, you can probe the next address of probing sequence according to how to handle collision, and keep doing this.
  - Until some cell is empty (can be inserted into)
  - Or find the same key (no need of insertion)

Code of Hash Table Insertion

```
// insert the element e into hash table HT
template <class Key, class Elem, class KEComp, class EEComp>
bool hashdict<Key, Elem, KEComp, EEComp>::hashInsert(const Elem& e) {
    int home= h(getkey(e));                  // home save the base address
    int i=0;
    int pos = home;                          // Start position of the probing sequence
    while (!EEComp::eq(EMPTY, HT[pos])) {
        if (EEComp::eq(e, HT[pos])) return false;
        i++;
        pos = (home+p(getkey(e), i)) % M;    // probe
    }
    HT[pos] = e;                             // insert the element e
    return true;
}
```

Search Algorithm

- Similar to the process of insertion
  - Use the same probing sequence
- Let the hash function be h, assume the given value is K
  - If the space corresponding to this address is not occupied, then search fails
  - If not, compare the value of this address with K, if they are equal, then search succeeds
  - Otherwise, probe the next address of the probing sequence according to how to handle collision, and keep doing this.
    - Find the equal key, search succeeds
    - Haven’t found when arrive at the end of probing sequence, then search fails

10.3 Retrieval in a Hash Table

```
template <class Key, class Elem, class KEComp, class EEComp>
bool hashdict<Key, Elem, KEComp, EEComp>::
hashSearch(const Key& K, Elem& e) const {
    int i=0, pos= home= h(K);                    // initial position
    while (!EEComp::eq(EMPTY, HT[pos])) {
        if (KEComp::eq(K, HT[pos])) {            // have found
            e = HT[pos];
            return true;
        }
        i++;
        pos = (home + p(K, i)) % M;
    } // while
    return false;
}
```

Deletion

- Something to consider when delete records:
  - (1) The deletion of a record mustn't affect the search later
  - (2) The storage space released could be used for the future insertion
- Only open hashing (separated synonyms lists) methods can actually delete records
- Closed hashing methods can only make marks (tombstones), can't delete records actually
  - The probing sequence would break off if records are deleted. Search algorithm "until an empty cell is found (search fails)"
  - Marking tombstones increases the average search length

Problems Caused by Deletions

0	1	2	3	4	5	6	7	8	9	10	11	12
	K1	K2	K1		K2	K2	K2			K2		

- For example, a hash table of length M = 13, let keys be $k1$ and $k2$, h($k1$) = 2 , h($k2$) = 6。
- Quadratic probing
  - The quadratic probing sequence of $k1$: 2、3、1、6、11、11、6、5、12、...
  - The quadratic probing sequence of $k2$: 6、7、5、10、2、2、10、9、3、...
- Delete the record at the position 6, put the element in the last position 2 of $k2$ sequence instead, set position 2 to empty
- search $k1$, but fails (may be put at position 3 or 1 in fact)

Tombstones

- Set a special mark bit to record the cell status of the hash table
  - Be occupied
  - Empty
  - Has been deleted
- The mark to record the status of has been deleted is called tombstone
  - Which means it was occupied by some record ever
  - But it isn't occupied now

Deletion Algorithms with Tombstones

```
template <class Key, class Elem, class KEComp, class EEComp>Elem
hashdict<Key,Elem,KEComp,EEComp>::hashDelete(const Key& K)
{   int i=0,  pos = home= h(K);                   // initial position
    while (!EEComp::eq(EMPTY, HT[pos]))  {
        if (KEComp::eq(K, HT[pos])){
            temp = HT[pos];
            HT[pos] = TOMB;                       // set up tombstones
             return temp;                         // return the target
        }
        i++;
        pos = (home + p(K, i)) % M;
    }
    return EMPTY;
}
```

Insertion Operation with Tombstones

- If a cell marked as a tombstone is met at the time of insertion, can we insert the new record into this cell?
  - In order to avoid inserting two same keys
  - The process of search should carry on along the probing sequence, until find a real empty cell

An Improved Version of Insertion Operation with Tombstones

```
template <class Key, class Elem, class KEComp, class EEComp>
bool hashdict<Key, Elem, KEComp, EEComp>::hashInsert(const
Elem &e)  {
    int insplace, i = 0, pos = home = h(getkey(e));
    bool tomb_pos = false;
    while (!EEComp::eq(EMPTY, HT[pos])) {
        if (EEComp::eq(e, HT[pos])) return false;
        if (EEComp::eq(TOMB, HT[pos]) &&  !tomb_pos)
        {insplace = pos;  tomb_pos = true;}            // The first
        pos = (home + p(getkey(e), ++ i)) % M;
    }
    if (!tomb_pos) insplace=pos;                       // no tombstone
      HT[insplace] = e;    return true;
}
```

Efficiency Analysis of Hash Methods

- Evaluation standard: the number of record visits needed for insertion, deletion, search
- Insertion and deletion operation of hash tables are both based on search
  - Deletion: must find the record at first
  - Insertion: must find until t the tail of the probing sequences, which means need a failed search for the record
    - For the situation without consideration about deletion, it is the tail cell.
    - For the situation with consideration about deletion, also need to arrive at the tail to confirm whether there are repetitive records

Important Factors Affecting Performance of Retrieval

- Expected cost of hash methods is related to the load factor
- $\alpha = N/M$
  - When α is small, the hash table is pretty empty, it's easy for records to be inserted into empty base addresses.
  - When α is big, inserting records may need collision resolution strategies to find other appropriate cells
- With the increase of α, more and more records may be put further away from their base addresses

Analysis of Hash Table Algorithms (1)

- The probability of base addresses being occupied is α
- The probability of the i-th collision occurring is

$$\frac{N(N-1)\cdots(N-i+1)}{M(M-1)\cdots(M-i+1)}$$

- If N and M are both very large, then it can be expressed approximately as

$$(N/M)^i$$

- The expected value of the number of probing is 1, plus occurring probability of each the i-th ($i \geq 1$) collision, which is cost of inserting, :

$$1+\sum_{i=1}^{\infty}(N/M)^i = 1/(1-a)$$

Analysis of Hash Table Algorithms (2)

- A cost of successful search (or deletion) is the same as the cost of insertion
- With the increase of the number of records of hash tables, α also get larger and larger
- We can get the average cost of insertion (the average of the cost of all the insertion) by computing the integral from 0 to current value of α

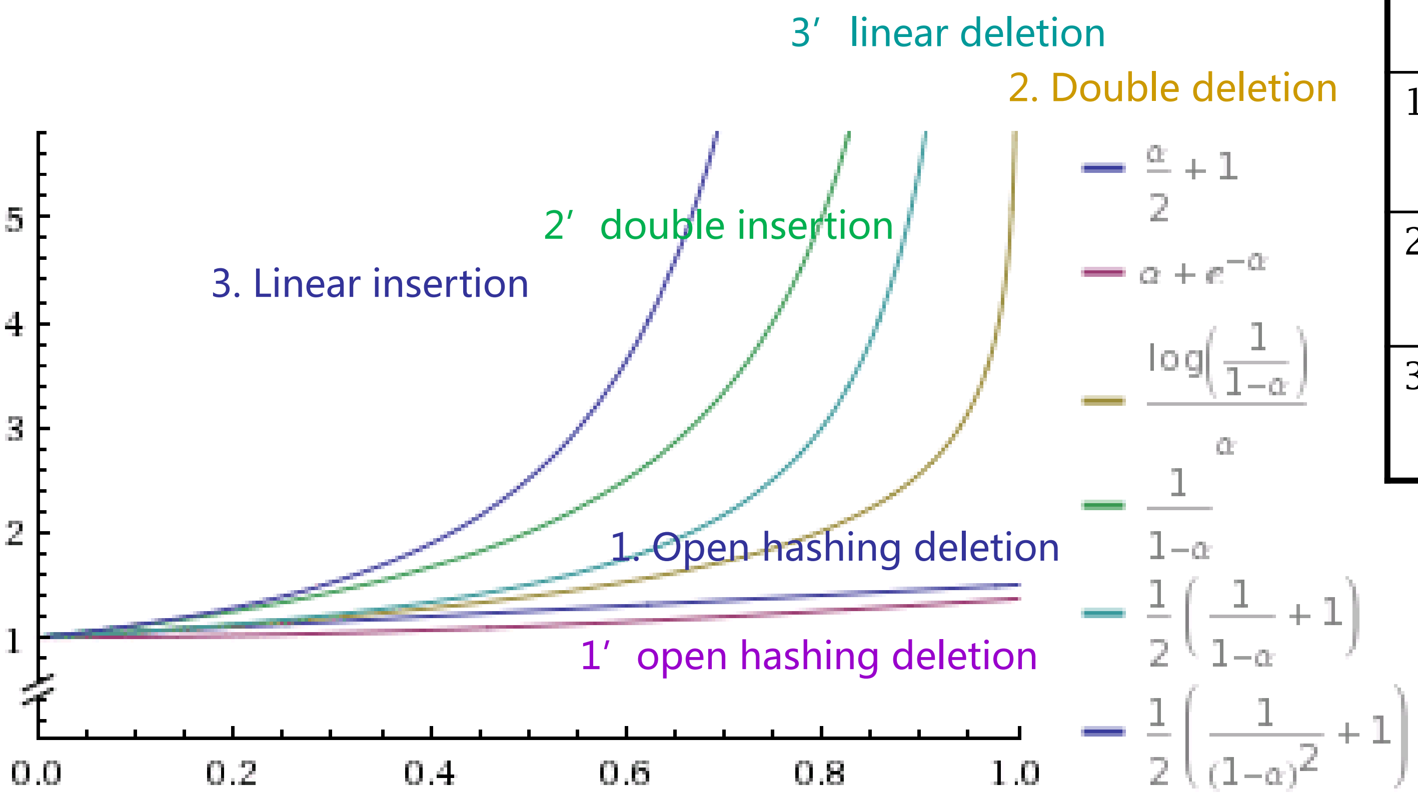
$$\frac{1}{a}\int_0^a \frac{1}{1-x}dx = \frac{1}{a}\ln\frac{1}{1-a}$$

Hash Table Algorithms Analysis (table)

No.	Collision resolution strategy	Successful search (deletion)	Failed search (insertion)
1	Open hashing	$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}$
2	Double hashing	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$
3	Linear probing	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$

Hash Table Algorithms Analysis (diagram)

- ASLs of using different way to resolve collision in hash tables



No.	Collision resolution strategy	Successful search (deletion)	Failed search (insertion)
1	Open hashing	$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}$
2	Double hashing	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$
3	Linear probing	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$

Conclusion of Hash Table Algorithms Analysis

- Normally the cost of hash methods is close to the time of visiting a record. It is very effective , greatly better than binary search which need log n times of record visit
    - Not depend on n, only depend on the load factor $\alpha = n/M$
    - With the increase of α, expected cost would increase too
    - When $\alpha \le 0.5$, The excepted cost of most operations is less than 2 (someone say 1.5)
- The practical experience indicates that the critical value of the load factor α is 0.5 (close to half full)
    - When the load factor is bigger than this critical value, the performance would degrade rapidly

Conclusion of Hash Table Algorithms Analysis (2)

- If the insertion or deletion of hash tables is complicated, then efficiency degrades
  - A mass of insertion operation would make the load factor increases.
    - Which also increase the length of synonyms linked chains, and also increase ASL
  - A mass of deletion would increase the number of tombstones.
    - Which increase the average length from records to their base addresses
- In the practical application, for hash tables with frequent insertion or deletion, we can perform rehashing for hash tables regularly
  - Insert all the records to another new table
    - Clear tombstones
    - Put the record visited most frequently on its base address

Thinking

- Can we mark the status of empty cell and having been deleted as a special value, to distinguish them from "occupied" status?
- Survey implementation of dictionary other than hash tables.

Data Structures and Algorithms

Thanks

the National Elaborate Course (Only available for IPs in China)
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

Ming Zhang, Tengjiao Wang and Haiyan Zhao

Higher Education Press, 2008.6 (awarded as the "Eleventh Five-Year" national planning textbook)