Data Structures

Hush Table

Teacher : Wang Wei

1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. 殷人昆, 数据结构
3. 金远平, 数据结构
4. http://inside.mines.edu/~dmehta/

Hashing

- **Hash Table**
  - The dictionary pairs are stored in a table **HT[*m*]**
  - **HT** is partitioned into ***m*** **position**
  - Each position of this array is **a bucket**
  - A bucket is said to consist of ***s*** **slots**
    - usually s=1, each bucket hold only one dictionary pair
  - Each slot being large enough to hold one dictionary pair
- **Hash function *hash***
  - Converts each **key *k*** into an index in the range **[0, m-1]**
  - ***hash(key)*** is the *home bucket* for **key *k***
- Every dictionary pair ***(key, element)*** is stored in its home bucket HT***[hash[key]]***

Hashing

- Consequently
  - The number of buckets ***m*** is usually of the same magnitude as the number of keys
  - The number of keys ***n*** is also much less than the total number of possible keys ***N*** in the hash table
  - **The hash function *hash* maps several different keys into the same home bucket**
    - **Synonyms (同义词)**
- Example
  - **Keys are 12361, 07251, 03309, 30976**
  - **Hash function :** ***hash*(*key*) = *key* % 73 + 13420**
  - **Then *hash*(12361) = *hash*(07250) = *hash*(03309) = *hash*(30976) = 13444**

Overflow and Collision

- **if s>1**
  - Since many keys typically have the same home bucket
  - An **overflow has occurred**
    - There is full and no space in the home bucket for a new dictionary pair
  - A **collision occurs**
    - When the home bucket for the new pair is not empty and occupied by a pair with a different key
- **if s=1**
  - **collisions and overflows occur together**
    - each bucket has 1 slot
      - when a bucket can hold only one pair

Hash Table Issues

- Overflow necessarily occur!
- It is desirable issues:
  - 1 Choice of hash function
    - A hash function is both easy to compute and minimizes the number of collisions
    - *uniform hash function*
  - 2 Overflow handling method
  - 3 Size (number of buckets) of hash table

Hash Function

- Two parts :
  - Convert key into a nonnegative integer in case the key is not an integer
  - Map an integer into a home bucket
- Desired properties
  - Random key has an equal chance of hashing into any of the buckets
    - *uniform hash function*
  - ***homeBucket = hash(key)*** is an integer in the *range [0, m-1]*

Division

- Most common method
  - the most widely used in practice
- Keys
  - assumed : Keys are non-negative integers
  - using the modulo (%) operator
- Hash function

  $homeBucket = hash(key) = key \% p \qquad p \leq m$

  $0 \leq homeBucket < p \leq m$

  - key : a pair(*key*,*element*)
  - p : a prime number
  - m : the number buckets of the hash table
  - *homeBucket* : the remainder is used as the home bucket for key

- Example:
  - *key* = 962148
  - *m* = 25 or *HT*[25]
  - *p* = 23
- *homeBucket* = *hash*(962148) = 962148 % 23 = 12

Mid-Square

- Key
  - The home bucket for a key by **squaring** the key
  - assumed : key = integer
  - *r* bits : an appropriate number of bits from the middle of the square to obtain the bucket address
- Hash function

  homeBucket = *r* bits

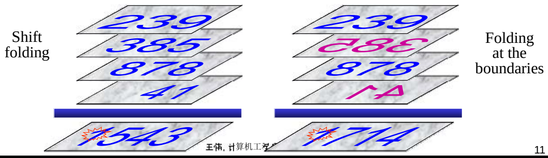
- The size of hash tables is chosen to be a power of 2 or 8
  - HT[*homeBucket*]
    - such as $0 \leq homeBucket \leq 2^r\text{-}1$ or $0 \leq homeBucket \leq 8^r\text{-}1$
- The middle bits of the square usually depend on all bits of the key

- Example
  - $m = 8^r$
  - $r = 3$

Element (Identifier)	Key (**Octal** codes)	Key^2	*homeBucket*
A	01	**01**	001
*A*1	0134	2**042**0	042
*A*9	0144	2**342**0	342
B	02	**04**	004
DMAX	04150130	21526**443**617100	443
*DMAX*1	0415013034	5264473**352**2151420	352
AMAX	01150130	1354**236**17100	236
*AMAX*1	0115013034	3454246**652**2151420	652

Folding

- Key
  - is partitioned into several parts
  - possibly the last being of the same length
- Hash function
  - *homeBucket* = $hash(key) = \sum_{i=1}^{l} pi$
- Example
  - *key* = 23938587841
  - A part = 3, decimal digits long
  - Partitions： p1=239 p2=385 p3=878 p4=41



Digit Analysis

- Key
  - Keys are known in advance
  - Each key is interpreted as a number using some radix ***r***
  - The digits of each key are examined
- Same radix is used for all the keys
  - λ_k : distribution uniformity
    - The smaller the value, the more evenly the radix ***r*** are distributed in the *k*th bit of the key
  - n : the number of keys
  - k : bits of each key
  - r : an radix
- Hash function
  - *homeBucket* = the number of bits which are distributed evenly for these keys

- Example
  - n = 8
  - r = 10
  - k = 6

Expected value of uniform appearance of ***r*** in ***n***

The number of times the ***ith*** digit appears on the ***kth*** bit

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

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

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

Overflow Handling

- An overflow occurs
  - when the home bucket for a new pair ***(key, element)*** is full
- Eliminate overflows by permitting each bucket to keep a list of all pairs for which it is the home bucket
  - Open addressing： array linear list
    - Search the hash table in some systematic fashion for a bucket that is not full
      - Linear probing (linear open addressing)
      - Quadratic probing
      - Random probing
  - Chaining： single linked list

Open addressing： array linear list

(1) Linear Probing

- s=1, search or insert a key
  - Computed $H_0 = hash\,(key)$
  - Examined $H_i = (H_{i-1}+1)\ \%\ m, \quad i = 1, 2, \ldots, m-1$

    $H_0+1, H_0+2, \ldots, m-1, 0, 1, 2, \ldots, H_0-1$

    or

    $H_i = (H_0 + i)\ \%\ m, \quad i = 1, 2, \ldots, m-1$
  - Until one of the following happens
    - 1 the bucket `HT[(hash(key)+j)%m]` `== key`
      - *key* has been found
    - 2 `HT[(hash(key)+j)%m]` is empty, *key* is not in the table
    - 3 return to the starting position `HT[(hash(key)+j)%m]`
      - The table is full and key is not in the table

- Keys：

  37, 25, 14, 36, 49, 68, 57, 11
- *HT*[12], *m* = 12
- Hash function：

  Hash (*key*) = key%11

Hash (37) = 4
Hash (25) = 3
Hash (14) = 3
Hash (36) = 3
Hash (49) = 5
Hash (68) = 2
Hash (57) = 2
Hash (11) = 0

Linear Probing：

0	1	2	3	4	5	6	7	8	9	10	11
11		68	25	37	14	36	49	57			
(1)		(1)	(1)	(1)	(3)	(4)	(5)	(7)			

ASL (Average Search Length)

- Successful：
  - The average number of comparisons
  - The average number of buckets examined in a successful search

$$ASL_{succ} = \frac{1}{8}\sum_{i=1}^{8} Ci = \frac{1}{8}(1+1+3+4+3+1+7+1) = \frac{21}{8}$$

- Unsuccessful：

$$ASL_{unsucc} = \frac{2+1+8+7+6+5+4+3+2+1+1}{11} = \frac{40}{11}$$

Class Definition

using Linear Probing

```
const int DefaultSize = 100;
enum KindOfStatus {Active, Empty, Deleted};
                                        //元素分类 (活动/空/删)
template <class E, class K>
class HashTable {                        //散列表类定义
public:
    HashTable (const int d, int sz = DefaultSize);
                                        //构造函数
    ～HashTable() { delete []ht;  delete []info; }
                                        //析构函数
```



```
    HashTable<E, K>& operator =
        (const HashTable<E, K>& ht2);   //表赋值
    bool Search (K k1, E& e1) const;     //搜索k1
    bool Insert (const E& e1);           //插入e1
    bool Remove (const E& e1);           //删除e1
    void makeEmpty ();                   //置表空

private:
    int divitor;                   //散列函数的除数
    int n, TableSize;              //当前桶数及最大桶数
    E *ht;                         //散列表存储数组
    KindOfStatus *info;            //状态数组
    int FindPos (K k1) const;      //散列函数
```



```
    int operator == (E& e1) { return *this == e1; }
                                   //重载函数：元素判等
    int operator != (E& e1) { return *this != e1; }
                                   //重载函数：元素判不等
};
```

```
template<class E, class K>              //构造函数
HashTable<E, K>::HashTable (int d, int sz)
{
    divitor = d;                        //除数
    TableSize = sz;  n = 0;             //表长
    ht = new E[TableSize];              //表存储空间
    info = new KindOfstatus[TableSize];
 for (int i = 0; i < TableSize; i++) info[i] = empty;
};
```


Search Function

```
//搜索在一个散列表中关键码与k1匹配的元素,
//搜索成功，则函数返回该元素的位置,
//否则返回，插入点（如果有足够的空间）
template <class E, class K>
int HashTable<E, K>::FindPos (K k1) const
{
    int i = k1 % divitor;                    //计算初始桶号
    int j = i;                               //j是检测下一空桶下标
   do {
        if (info[j] == Empty || info[j] == Active &&
            ht[j] == k1) return j;           //找到初始桶号
        j = (j+1) % TableSize;               //找下一个空桶
    } while (j != i);
    return j;          //转一圈回到开始点, 表已满, 失败
}
```



```
//使用线性探查法在散列表ht(每个桶容纳一个元素)中搜索k1

bool HashTable<E, K>::Search (K k1, E& e1)
{
    int i = FindPos (k1) ;             //搜索
    if (info[i] != Active || ht[i] != k1) return false;
    e1 = ht[i];
    return true;
}
```

Insertion Function

```
//在ht表中搜索k1。若找到则不再插入，若未找到，
//但找到位置的标志是Empty或Deleted, x插入

template <class E, class K>
bool HashTable<E, K>::Insert (K k1, const E& e1)
{
    int i = FindPos (k1);          //用散列函数计算桶号
    if (info[i] != Active)
   {                                //该桶空,存放新元素
        ht[i] = e1;   info[i] = Active;
        n++;   return true;
    }
    if (info[i] == Active && ht[i] == e1)
        cout << "表中已有此元素，不能插入！ \n";
    else cout << "表已满，不能插入！ \n";
    return false;
};
```


Deletion Function

```
//在ht表中删除元素key, 并在引用参数e1中得到它

template <class E, classK>
bool HashTable<E, K>::Remove (K k1, E& e1)
{
    int i = FindPos (k1);
    if (info[i] == Active)
    {                      //找到要删元素, 且是活动元素
        info[i] = Deleted;  n--;
                           //做逻辑删除标志, 并不真正物理删除
        return true;
    }
    else return false;
};
```


Problem

- **Tend to cluster together**
- **Increasing the search time**
  - The search for a key involves comparison with keys that have different hash values
- Improvement :
  - **Quadratic Probing**
  - Rehashing
  - Random Probing

(2) quadratic probing

- Hash function

  $H_0 = hash(key)$

- Search is carried out by examining buckets :

  $H_i = (H_0+i^2)$ % m $H_i = (H_0-i^2)$% m

  $i = 1, 2, 3, \ldots, (m-1)/2$

  - when $H_0-i^2<0$ then (j = ((H_0-i^2)% m))<0 -> j += m

- So : H_0, H_0+1, H_0-1, H_0+4, H_0-4, …

- m is a prime number of the form $4k+3$, for k is a integer
  - such as 3, 7, 11, 19, 23, 31, 43, 59, 127, 251, 503, …

- Keys：

  37, 25, 14, 36, 49, 68, 57, 11

- HT[19], m = 19
- Hash function：

  $Hash\ (key) = key\%19$

$Hash$ (37) = 18
$Hash$ (25) = 6
$Hash$ (14) = 14
$Hash$ (36) = 17
$Hash$ (49) = 11
$Hash$ (68) = 11
$Hash$ (57) = 0
$Hash$ (11) = 11

Quadratic Probing ：

0	1	2	3	4	5	6	7	8	9	10	11
57						25				11	49
(1)						(1)				(3)	(1)

12	13	14	15	16	17	18
68		14			36	37
(2)		(1)			(1)	(1)

ASL (Average Search Length)

- Successful：

$$ASL_{succ} = \frac{1}{8}\sum_{i=1}^{8} Ci = \frac{1}{8}(1+1+1+1+1+2+1+3) = \frac{11}{8}$$

- Unsuccessful：

$$ASL_{unsucc} = \frac{2+2+3+4+2+2+3+4+11*1}{19} = \frac{33}{19}$$

Example 2

- Keys:

  Burke, Ekers, Broad, Blum, Attlee, Alton, Hecht, Ederly

- Hash function:

  Hash (*key*) = *ord* (*key*) - *ord* ('*A*') //*ord*()

  Hash (Burke) = 1 *Hash* (Ekers) = 4
  Hash (Broad) = 1 *Hash* (Blum) = 1
  Hash (Attlee) = 0 *Hash* (Hecht) = 7
  Hash (Alton) = 0 *Hash* (Ederly) = 4

homeBucket : 0～25 , non-negative integer

HT[26], *m* = 26

- *HT*[26], *m* = 26 *HT*[31], *m* = 31 , Linear Probing :

0	1	2	3	4
Attlee	Burke	Broad	Blum	Ekers
(1)	(1)	(2)	(3)	(1)

5	6	7	8	9	
Alton	Ederly	Hecht			
(6)	(3)	(1)			

- Successful:

$$ASL_{succ} = \frac{1}{8}\sum_{i=1}^{8} Ci = \frac{1}{8}(1+1+2+3+1+6+3+1) = \frac{18}{8}$$

- Unsuccessful:

$$ASL_{unsucc} = \frac{9+8+7+6+5+4+3+2+18}{26} = \frac{62}{26}$$

- HT[31], m = 31, quadratic probing :

0	1	2	3	4	5
Blum	Burke	Broad		Ekers	Ederly
(3)	(1)	(2)		(1)	(2)

6	7	8	9	10	11
	Hecht				
	(1)				

25	26	27	28	29	30
		Alton			Attlee
		(5)			(3)

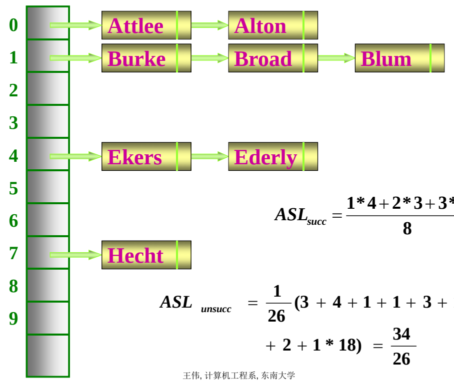
- Successful: $ASL_{succ} = \frac{1}{8}\sum_{i=1}^{8} C_i = \frac{1}{8}(3+1+2+1+2+1+5+3) = \frac{18}{8}$

- Unsuccessful: $ASL_{unsucc} = \frac{1}{26}(6+5+2+3+2+2+20) = \frac{40}{26}$

Chaining : single linked list

王伟, 计算机工程系, 东南大学 34

HT[0..25]，m = 26



$$ASL_{succ} = \frac{1*4+2*3+3*1}{8} = \frac{13}{8}$$

$$ASL_{unsucc} = \frac{1}{26}(3+4+1+1+3+1+1+2+1*18) = \frac{34}{26}$$

王伟, 计算机工程系, 东南大学 35

Class Definition

using Chaining Probing

```
//各桶中同义词子表的链结点定义

#include <assert.h>
const int defaultSize = 100;
template <class E, class K>
struct ChainNode {
    E data;                         //元素
    ChainNode<E, K> *link;          //链指针
};
```

王伟, 计算机工程系, 东南大学 36

```
template <class E, class K>
class HashTable
{        //散列表(表头指针向量)定义
public:
    HashTable (int d, int sz = defaultSize);
                                    //散列表的构造函数
    ～HashTable() { delete [] ht; }  //析构函数
    bool Search (K k1, E& e1);      //搜索
    bool Insert (K k1, E& e1);      //插入
    bool Remove (K k1, E& e1);      //删除

private:
    int divisor;                    //除数（必须是质数）
    int TableSize;                  //容量(桶的个数)
    ChainNode<E, K> **ht;           //散列表定义
    ChainNode<E, K> *FindPos (K k1);  //散列
};
```


Constructor

```
template <class E, class K>          //构造函数
HashTable<E, K>::HashTable (int d, int sz)
{
    divisor = d;  TableSize = sz;
    ht = new ChainNode<E, K>*[sz];   //创建头结点
    assert (ht != NULL);             //判断存储分配成功否
};
```


Verify Position

```
//在散列表ht中搜索关键码为k1的元素。函数返回
//一个指向散列表中某位置的指针

template <class E, class K>
ChainNode<E, K> *HashTable<E, K>::FindPos (K k1)
{
    int j = k1 % divisor;            //计算散列地址
    ChainNode<E, K> *p = ht[j];      //扫描第j链的同义词子表
    while (p != NULL && p->data != k1) p = p->link;
    return p;                        //返回
};
```

Analysis

- **Linear List Of Synonyms**
  - Each bucket keeps a linear list
    - it is the home bucket
  - The linear list
    - may or may not be sorted by key
    - may be an array linear list or a chain

Definition of α

- The key density od a hash table is the ratio **n/T**
- The *loading density* or *loading factor* of a hash table is
  - **α = n/m = n/(s*b)**

$$\alpha = \frac{n}{m}$$

- Where
  - n : the number of pair in the table
  - m : the total number of possible keys
  - s : the number of slots
  - b : the number of buckets

Expected Performance

- S_n
  - expected number of buckets examined in a successful search when n is large
  - Assume : random search key x_i $(1 \leqslant i \leqslant n)$
  - When $\alpha = n / m$, ASLsucc = S_n
- U_n
  - expected number of buckets examined in a unsuccessful search when n is large
  - When $\alpha = n / m$, ASLunsucc = U_n

- **Time to put and remove governed by U_n**

ASL and α

Overflow Techniques		ASL	
		Sn	Un
Open Addressing	Linear probing	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
	Random Quadratic robing Rehashing	$-\left(\frac{1}{\alpha}\right)\log_e(1-\alpha)$	$\frac{1}{1-\alpha}$
Chaining		$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}\approx\alpha$