Data Structures

Hush Table

Teacher : Wang Wei

1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. ,
3. ,
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 erflow

Division

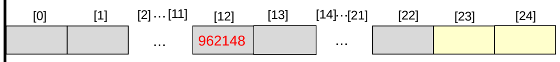
- Most common method
  - the most widely used in practice
- Keys
  - assumed : Keys are non-negative integers
  - using the modulo (%) operator
- Hash function

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

  $0 \leq homeBucket < p \leq m$

  - 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 $0 \leq homeBucket \leq 2^r-1$ or $0 \leq homeBucket \leq 8^r-1$
- The middle bits of the square usually depend on all bits of the key

- Example
  - $m = \mathbf{8}^r$
  - $r = 3$

Element (Identifier)	Key (**Octal** codes)	Key^2	*HomeBucket*
A	01	<u>**01**</u>	001
*A*1	0134	2<u>**042**</u>0	042
*A*9	0144	2<u>**342**</u>0	342
B	02	<u>**04**</u>	004
[illegible]	04150130	21526<u>**443**</u>617100	443
[illegible]	[illegible]5013034	526447<u>**352**</u>2151420	352
[illegible]	[illegible]	1354<u>**236**</u>17100	236
[illegible]	[illegible]	[illegible]<u>**652**</u>2151420	652

- 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

$$_{k} = \sum_{i=1}^{r} \left({}^{k}_{i} - n / r \right)^{2}$$

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 $H_i = (H_{i-1}+1)\ \%\ m,\quad i = 1, 2, \dots, m-1$
    $H_0+1, H_0+2, \dots, m-1, 0, 1, 2, \dots, H_0-1$

    or

    $H_i = (H_0 + i)\ \%\ m,\quad i = 1, 2, \dots, m-1$
  - 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

16

- 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)			

17

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：

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

18

Class Definition

using Linear Probing

```
const int DefaultSize = 100;
enum KindOfStatus {Active, Empty, Deleted};
                                        //          (     /   /   )
template <class E, class K>
class HashTable {                       //
public:
    HashTable (const int d, int sz = DefaultSize);
                                        //
       HashTable() { delete []ht;  delete []info; }
                                        //
```

```
     HashTable<E, K>& operator =
           (const HashTable<E, K>& ht2);   //
     bool Search (K k1, E& e1) const;      //      k1
      bool Insert (const E& e1);           //      e1
     bool Remove (const E& e1);            //      e1
     void makeEmpty ();                    //

 private:
     int divitor;                  //
     int n, TableSize;             //
      E *ht;                       //
     KindOfStatus *info;           //
     int FindPos (K k1) const;     //
```

```
     int operator == (E& e1) { return *this == e1; }
                                    //
     int operator != (E& e1) { return *this != e1; }
                                   //
 };
```

```
template<class E, class K>                //
HashTable<E, K>::HashTable (int d, int sz)
{
    divitor = d;                          //
    TableSize = sz;  n = 0;               //
    ht = new E[TableSize];                //
    info = new KindOfstatus[TableSize];
 for (int i = 0; i < TableSize; i++) info[i] = empty;
};
```

Search Function

```
//                                   k1
//
//
template <class E, class K>
int HashTable<E, K>::FindPos (K k1) const
{
    int i = k1 % divitor;                  //
    int j = i;                             //j
   do {
        if (info[j] == Empty || info[j] == Active &&
             ht[j] == k1) return j;          //
        j = (j+1) % TableSize;               //
    } while (j != i);
    return j;          //                    ,        ,
}
```

```
//                              ht(                   )      k1

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

Insertion Function

```
// ht k1 , ,
// Empty Deleted, x

template <class E, class K>
bool HashTable<E, K>::Insert (K k1, const E& e1)
{
    int i = FindPos (k1);       //
    if (info[i] != Active)
   {                            //     ,
        ht[i] = e1;   info[i] = Active;
        n++;   return true;
    }
    if (info[i] == Active && ht[i] == e1)
        cout << "                          \n";
    else cout << "                  \n";
    return false;
};
```

25

Deletion Function

```
// ht key, e1

template <class E, classK>
bool HashTable<E, K>::Remove (K k1, E& e1)
{
    int i = FindPos (k1);
    if (info[i] == Active)
    {            //          ,
        info[i] = Deleted;  n--;
                 //               ,
        return true;
    }
    else return false;
};
```

26

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

27

- Hash function

  $H_0 = hash(key)$
- Search is carried out by examining buckets :

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

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

  - when H_0-i^2

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

31

- *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：

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

32

- 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：

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

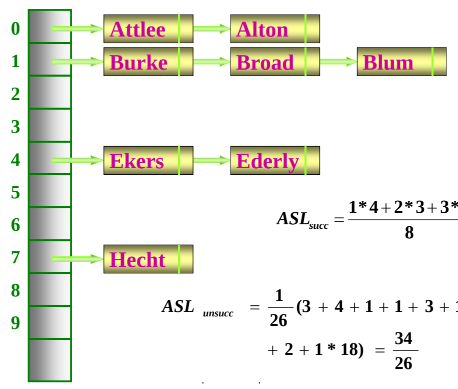
- Unsuccessful：

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

33

Chaining : single linked list

HT[0..25]，m = 26

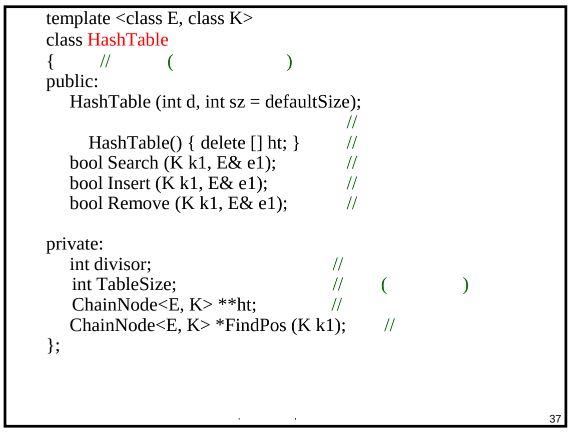


$$ASL_{succ} = \frac{1*4+2*3+3*1}{8} = \frac{13}{8}$$

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

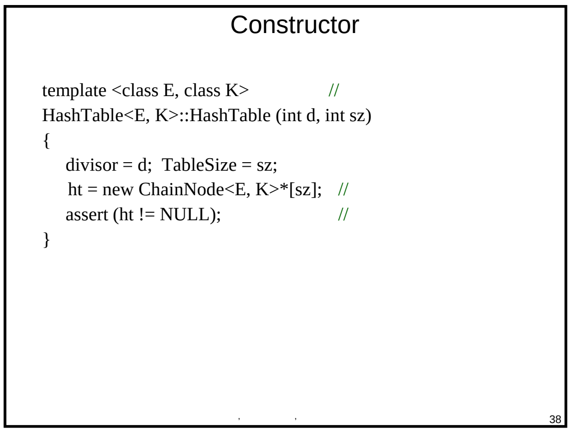
Class Definition
using Chaining Probing

```
//

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



```
template <class E, class K>
class HashTable
{        //          (                      )
public:
    HashTable (int d, int sz = defaultSize);
                                                  //
        HashTable() { delete [] ht; }             //
    bool Search (K k1, E& e1);                    //
    bool Insert (K k1, E& e1);                    //
    bool Remove (K k1, E& e1);                    //

private:
    int divisor;                                //
     int TableSize;                             //       (               )
     ChainNode<E, K> **ht;                      //
    ChainNode<E, K> *FindPos (K k1);        //
};
```

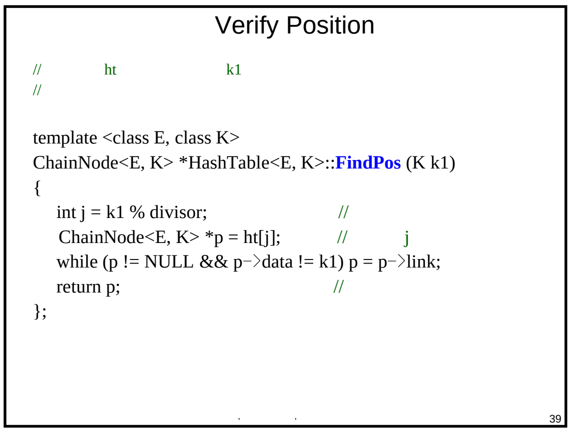
37



Constructor

```
template <class E, class K>              //
HashTable<E, K>::HashTable (int d, int sz)
{
    divisor = d;  TableSize = sz;
     ht = new ChainNode<E, K>*[sz];   //
    assert (ht != NULL);                 //
}
```

38



Verify Position

```
//            ht                        k1
//

template <class E, class K>
ChainNode<E, K> *HashTable<E, K>::FindPos (K k1)
{
    int j = k1 % divisor;                       //
     ChainNode<E, K> *p = ht[j];           //          j
    while (p != NULL && p−>data != k1) p = p−>link;
    return p;                                  //
};
```

39

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

40

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)**

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

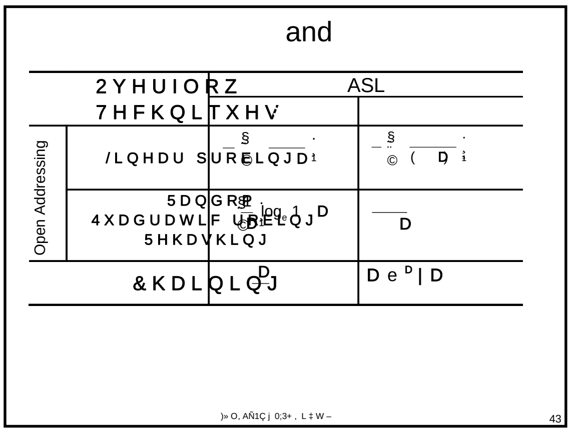
- Where
  - n : the number of pair in the table
  - m : the total number of possible keys
  - s : the number of slots
  - b : the number of buckets

41

Expected Performance

- S_n
  - expected number of buckets examined in a successful search when n is large
  - Assume : random search key x_i $(1 \le i \le n)$
  - 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**

42



and

2YHUIORZ 7HFKQLTXHV		ASL	
Open Addressing	/LQHDU SURELQJ	_ § ____ . © ¹	_ § ______ . © (D) ¹
	5DQGRP 4XDGUDWLF 5HKDVKLQJ	§ _ log 1 D © ¹ e	____ D
&KDLQLQJ		D	D e D | D

)» O, AÑ1Ç j 0;3+ , L ‡ W –

43