Chapter 8
Hashing

8.1 Introduction

Let look back to the ADT Dictionary, which is used in many applications:

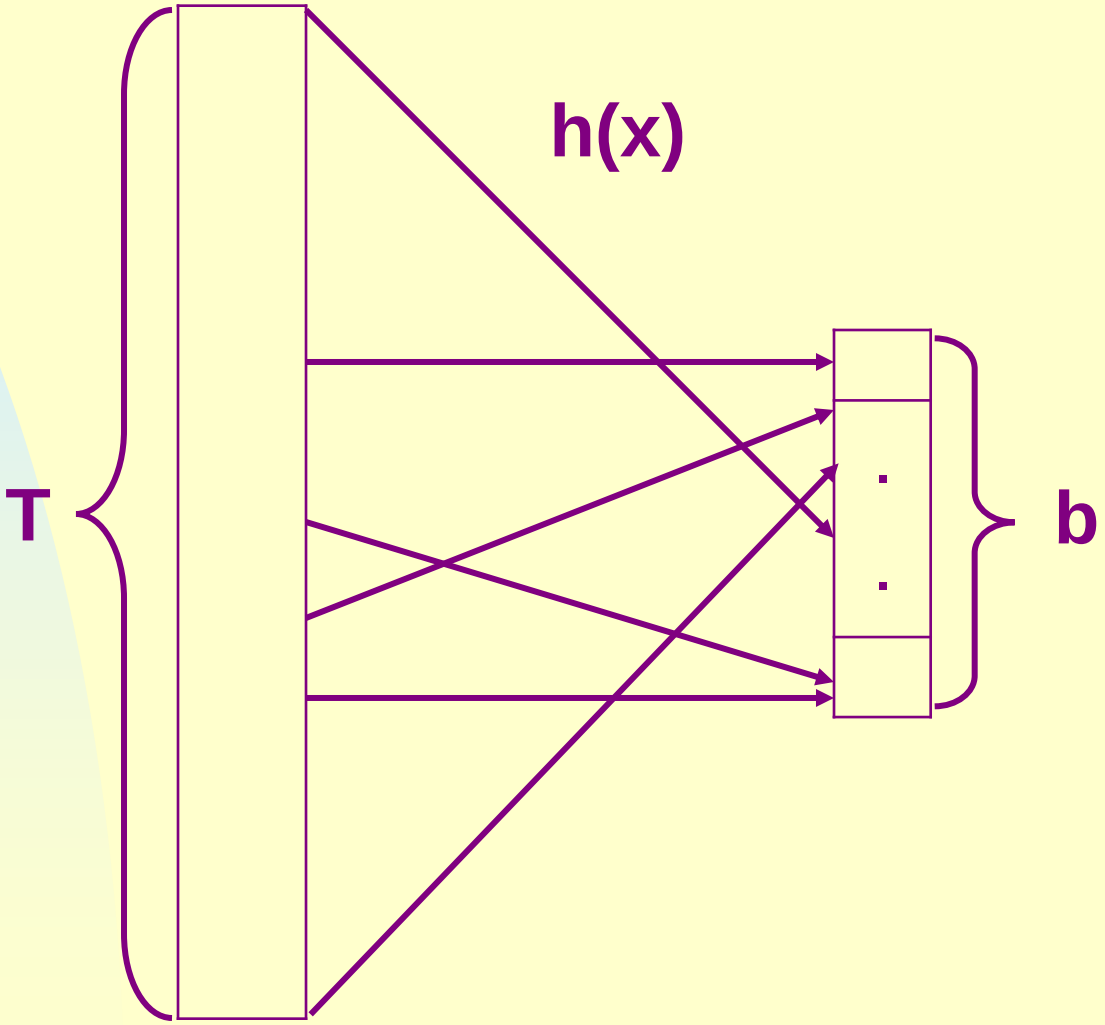
spelling checker, the thesaurus, the index for a database, the symbol tables generated by loaders, assemblers, and compilers, …

ADT 5.3

```
template <class K, class E>
class Dictionary {
public:
   virtual bool IsEmpty () const = 0;
      // return true iff the dictionary is empty
   virtual pair<K,E>* Get(const K&) const = 0;
      // return pointer to the pair with specified key;
      // return 0 if no such pair
   virtual void Insert(const pair<K,E>&) = 0;
      // insert the given pair; if key is a duplicate
      // update associated element
   virtual void Delete(const K&) = 0;
      // delete pair with specified key
};
```

- **The 3 basic operations on Dictionary are Get, Insert and Delete.**
- **In key comparison based search tree methods, these operations take O(h) time, where h is the height of the tree.**
- **If we directly use the key as an address, then we can perform these operations in O(1).**
- **However, the space for key definition is too big, we need to map the big space to a space of practical size.**

This leads to the technique of hashing, in which a hash function is used to map a big space to a small one:



8.2 Static Hashing

8.2.1 Hash Tables

In static hashing the dictionary pairs are stored in a table, ht, called hash table.

- **ht is partitioned into b buckets: ht[0:b-1], and maintained in sequential memory.**
- **each bucket holds s slots, each slot holds one pair, usually, s = 1.**
- **the address of a pair with key k is determined by a hash function h, h(k) is the hash or home address of k, h(k)∈{0, 1,…,b-1}.**

T --- the total number of possible keys.

n --- the number of pairs in the hash table.

Definition:

The key density of a hash table is the ratio n/T.

The loading density (or factor) of a hash table is $\alpha=n/(sb)$.

Usually, n<<T, and b<<T.

- 2 keys k_1 and k_2 are said to be **synonyms** with respect to h if $h(k_1) = h(k_2)$.
- a **collision** occurs when the home bucket for the new pair is not empty.
- an **overflow** occurs when a new pair is hashed into a full bucket.
- when s=1, collisions and overflows occur simultaneously.

Example 8.1:

Let b=26, s=2, n=10, hence α = 10/52 = 0.19

Assume the internal representation for A to Z corresponds to 0 to 25, then

h(k) = the first character of k

Keys: GA, D, A, G, L, A2, A1, A3, A4, and E.

The following shows the keys GA, D, A, G, L and A2 entered into the hash table.

ht	Slot 1	Slot 2
0	A	A2
1		
2		
3	D	
4		
5		
6	GA	G
.	.	.
.	.	.
25		

Next, A1 hashes into ht[0] --- overflow.

If no overflow, the time required to insert, delete or search using hash depends only on the time for computing the hash function and searching one bucket .

--- independent of n.

Since T>>b, it is impossible to avoid overflow, a mechanism to handle overflows is necessary.

8.2.2 Hash Functions

A hash function maps a key into a bucket address in the hash table.

It should be easy to compute and minimize the number of collisions.

If k is a key chosen at random from the key space, then we want the probability that h(k)=i to be 1/b for all buckets i.

A hash function satisfying this property is a uniform hash function.

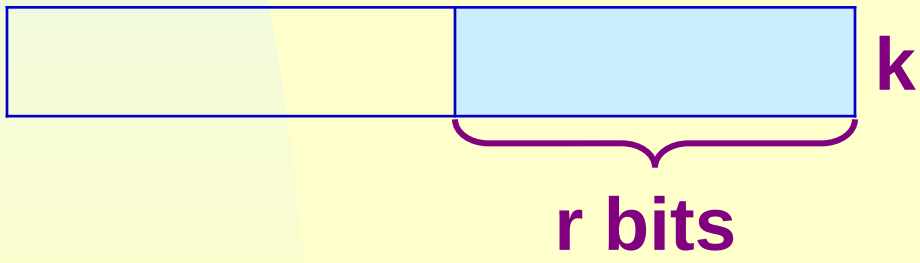
8.2.2.1 Division

h(k) = k % D b = D

The choice of D is critical.

For instance:

(1) If $D=2^r$, then h(k) depends only on the last r bits of k:



(2) If D is divisible by 2, then odd keys are mapped to odd buckets, even keys to even buckets.

In practice, it is sufficient to choose D such that it has no prime divisors less than 20.

8.2.2.2 Mid-Square

h(k) is computed by using an appropriate number of bits from the middle of k^2 to obtain the bucket address.

If r bits used, $b= 2^r$.

8.2.2.3 Folding

k is partitioned into several parts, all but the last being of the same length. These partitions are then added together to obtain the hash address for k.

Example 8.2:

k=12320324111220 is partitioned into parts that are 3 decimal digits long.

P_1=123, P_2=203, P_3=241, P_4=112, P_5=20.

- shift folding

  h(k)=123+203+241+112+20=699

- folding at the boundaries

  h(x)=123+302+241+211+20=897

8.2.2.4 Digit Analysis

Useful in the case of a static file.

- **each k is interpreted as a number using some radix r.**
- **the digits of each k are examined, digits having the most skewed distribution are deleted --- until the number of digits left is small enough to give an address.**

For example, consider students numbers:

7	1	1	1	4	1	0	1
7	1	1	1	4	1	0	2
7	1	1	1	4	1	0	3
					.		
					.		
7	1	1	1	4	4	2	7
7	1	1	1	4	4	2	8
7	1	1	1	4	4	2	9

Let b=1000, we use the last 3 digits.

8.2.4 Overflow Handling

8.2.4.1 Open Addressing

Assume:

- **the hash table, ht, is an array, and stores pointers to dictionary pairs.**
- **for simplicity, s=1.**

The following are the class definitions:

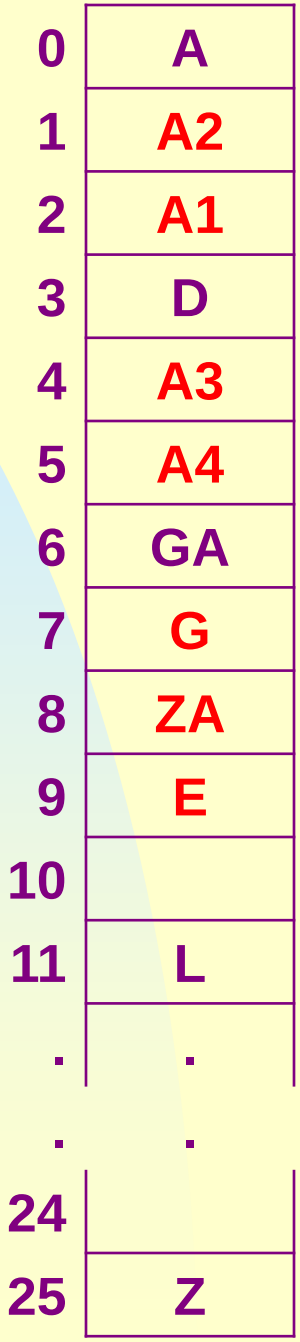
```
template<class K, class E>
typedef pair<K, E> * pairPtr;
class LinearProbing {
public:
    LinearProbing (int size) {
        if (size <3) throw “the size must be >= 3.”;
        b=size;
        ht=new pairPtr[b];
        fill(ht, ht+b,0);  // initialize the hash table to empty.
    };
private:
    int b;
    pairPtr *ht;
};
```

When overflow occurs, we need to find another bucket for the new identifier, the simplest solution is to find the closest bucket.

This is called linear probing or linear open addressing.

Example 8.6:

- b=26, ht is used circularly.
- keys: GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E.
- h(x)=first character of x, e.g., h(GA)=6.
- Initially, all entries in ht are null.



After entering

GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E,

we get the table shown left.

When linear open addressing is used to handle overflows, a hash table search for key k proceeds as follows:

(1) Compute h(k)

(2) Examine ht[h(k)], ht[h(k)+1%b], …, ht[h(k)+j%b] until one of the following happens:

(a) ht[h(k)+j%b] has a pair whose key is k---found.

(b) ht[h(x)+j%b] is empty---not in.

(c) return to the starting position h(k)---the table is full and k is not in.

```
template <class K, class E>
pair<K, E>* LinearProbing<K, E>::Get(const K& k)
{ // search the linear probing hash table ht (s=1) for k.
  // If found return a pointer to the pair, else return 0.
    int i=h(k);  // home bucket
    int j;
    for (j=i; ht[j] && ht[j]→first !=k;) {
       j = (j+1) % b;    // treat the table as circular
       if ( j == i ) return 0;  // back to the start point
    }
    if (ht[j]→first==k) return ht[j];
    return 0;
}
```

Analysis shows that the expected average number of key comparisons to look up a key is approximately

$$(2- \alpha) / (2- 2\alpha)$$

where α is the loading density.

In Example 8.6, $\alpha = 12 / 26 = 0.47$ and the average key comparisons is 1.5.

Although the average number of comparisons is small, the worst case can be quite large.

Some other open addressing methods include:

- **quadratic probing**
- **rehashing**
- **random probing**

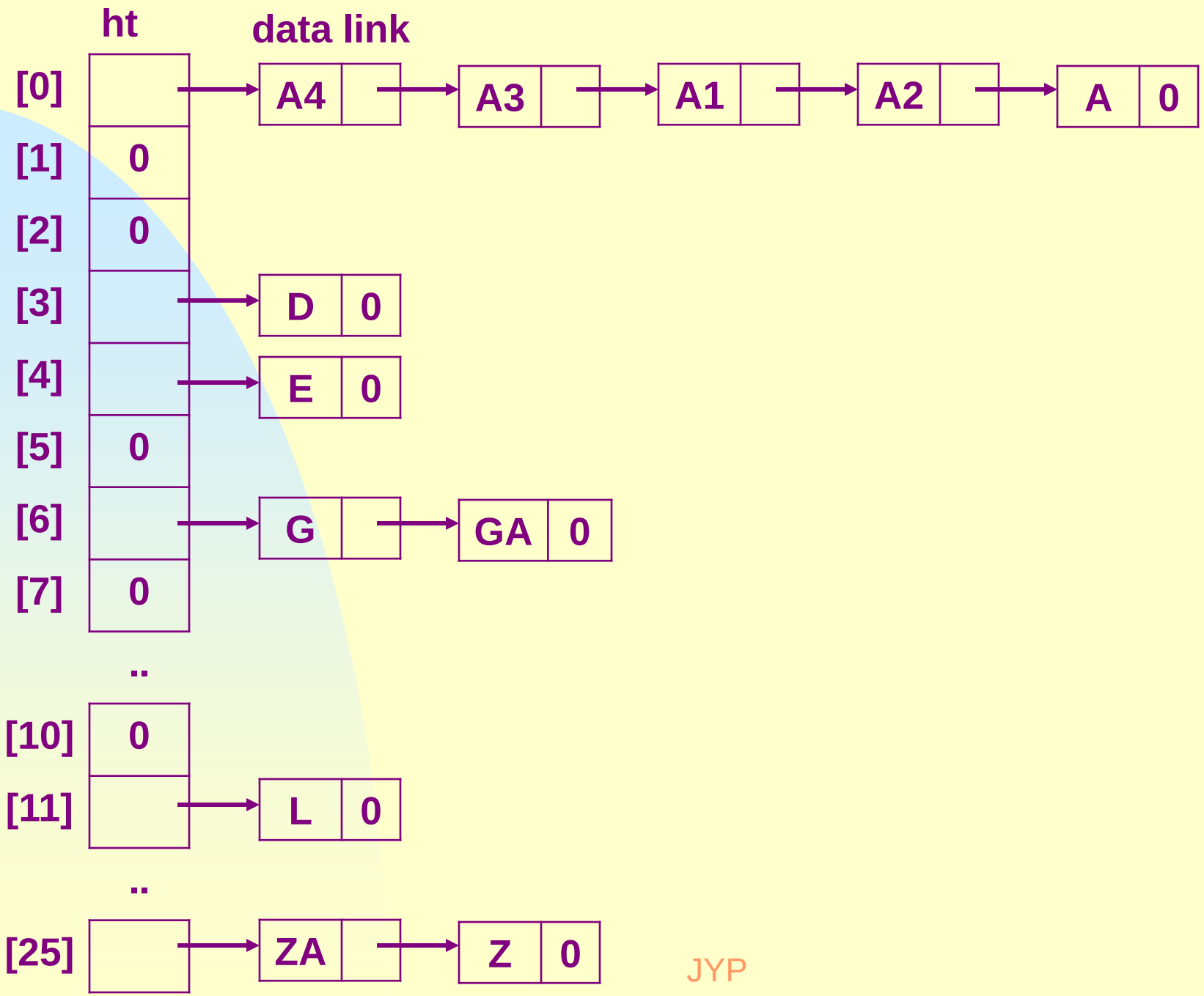
8.2.4.2 Chaining

Linear probing performs poorly because the search for a key involves comparison with keys having different hash values, making a local collision a global one .

Many of the comparisons can be saved if we maintain lists of keys, one list per bucket, each list containing all the synonyms for that bucket.

The lists are most frequently maintained as chains.

When chaining is used for the data of Example 8.6, we obtain the hash chains as in the next slide.


ht
data link
[0]
A4
A3
A1
A2
A
0
[1]
0
[2]
0
[3]
D
0
[4]
E
0
[5]
0
[6]
G
GA
0
[7]
0
..
[10]
0
[11]
L
0
..
[25]
ZA
Z
0

```
template <class K, class E>
typedef ChianNode<pair<K, E>>* ChainNodePtr;
class Chaining {   // assume Chaining is friend of ChainNode
public:
    Chaining(int size) {
        if (size <3) throw “the size must be >= 3.”;
        b=size;
        ht=new ChainNodePtr[b];
        fill(ht, ht+b,0);  // initialize the hash table to empty.
    };
private:
    int b;
    ChainNodePtr *ht;
};
```

```
template <class K, class E>
Pair<K, E>* Chaining<K, E>::Get(const K& k)
{ // Search the chained hash table ht for k. If a pair with key k
  // is found return a pointer to this pair, else return 0.
    int i = h(k);   // home bucket
    // search the chain starting at ht[i]
    for (ChainNode<pair<K, E>>* current=ht[i]; current;
                                          current=current→link)
       if (current→data.first==k) return &current→data;
    return 0;
}
```

The expected average number of key comparisons of a successful search is

$$\approx 1 + \alpha / 2$$

where α is the loading density.

The table in the next slide presents the results of empirical study. The values in each column give the average number of key comparisons.

$\alpha = n/b$	0.50		0.75		0.90		0.95	
Hash function	Chain	Open	Chain	Open	Chain	Open	Chain	Open
mid square	1.26	1.73	1.40	9.75	1.45	37.14	1.47	37.53
division	1.19	4.52	1.31	7.20	1.38	22.42	1.41	25.79
shift fold	1.33	21.75	1.48	65.10	1.40	77.01	1.51	118.57
bound fold	1.39	22.97	1.57	48.70	1.55	69.63	1.51	97.56
digit analysis	1.35	4.55	1.49	30.62	1.52	89.20	1.52	125.59
theoretical	1.25	1.50	1.37	2.50	1.45	5.50	1.48	10.50

As expected, Chaining outperforms Linear open addressing for overflow handling and Division is superior to other hash functions.

Exercises: P475-3, 6