Chap 8. Hashing (1)

Contents

8.1 Introduction

ADT *Dictionary* is
objects: a collection of $n > 0$ pairs, each pair has a key and an associated item
functions:
for all $d \in$ *Dictionary*, *item* $\in$ *Item*, $k \in$ *Key*, $n \in$ integer

Dictionary Create(*max_size*)	::=	create an empty dictionary.
Boolean IsEmpty(d, n)	::=	**if** ($n > 0$) **return** *FALSE* **else return** *TRUE*
Element Search(d, k)	::=	**return** item with key k, **return** NULL if no such element.
Element Delete(d, k)	::=	delete and return item (if any) with key k;
void Insert(d, *item*, k)	::=	insert *item* with key k into d.

ADT 5.3: Abstract data type *dictionary*

- Examples of dictionaries
  - Spelling checker
  - Thesaurus
  - Data dictionary
  - Symbol table

- In chapter 5, we learned how to build a BST for dictionaries
  - The worst-case time complexity for a BST is O(n)
  - The best-case time complexity for a balanced BST can be O(log n)

- In this chapter, We will learn a technique called *hashing* that enables us to perform *search*, *insert*, and *delete* in O(1) expected time.
  - Static hashing
  - Dynamic hashing

8.2 Static Hashing

8.2.1 Hash Tables

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

- Hash table with *b* ***buckets*** and *s* ***slots***
  - buckets : *ht*[0], ···, *ht*[b−1]
  - Each bucket is capable of holding *s* dictionary pairs.

- Hash function *h*(*k*)
  - determines the address of a pair whose key is *k*
  - maps keys into buckets
  - is an integer in the range 0 through *b*−1

- $h(k)$ is the hash or home address of k
- Under the ideal condition, dictionary pairs are stored in their home buckets.

- **Definition:**
  - The *key density* of a hash table is the ratio n/T, where n is the number of pairs in the table and T is the total number of possible keys.
  - The *loading density* or *loading factor* of a hash table is $\alpha = n / (sb)$.

- Suppose
  - keys are at most six characters long
  - The first character must be a letter
  - The remaining characters can be letters or digits

- Then, the number of possible keys
  - $T = \Sigma_{i=0}^{5} 26 \times 36^{i} > 1.6 \times 10^{9}$
  - But most applications use only very small fraction of it

- Hash table also uses only small number of buckets
  - The hash function *h* maps several different keys into the same bucket

Terminologies

- Two keys k_1 and k_2 are said to be *synonyms* with respect to h if $h(k_1) = h(k_2)$
- *Collision* : occurs when the home bucket for a new pair is not empty at the time of insertion
- *Overflow* : hash a new identifier into a full bucket
- Collision and overflow occur at the same time if each bucket has 1 slot.

Example 8.1

- Suppose b=26, s=2, n=10
  - α = 10 / 52 = 0.19
- identifiers: ‘acos’, ‘define’, ‘float’, ‘exp’, ‘char’, ‘atan’, ‘ceil’, ‘floor’, ‘clock’, ‘ctime’
- $h(x)$: the first character of x

- ‘acos’ and ‘atan’ are *synonym*
- ‘clock’: *overflow*

	Slot 0	Slot 1
0	**acos**	**atan**
1		
2	**char**	**ceil**
3	**define**	
4	**exp**	
5	**float**	**floor**
6		
. . .		
25		

Figure 8.1: Hash table with 26 buckets and two slots per bucket

- Insert, delete, find
  - O(s) if no overflow occurs
  - O(1) if no collision occurs
- But, collision occurs for most cases, since the ratio b/T is usually very small.

- Hash Table Issues
  - Choice of hash function
  - Overflow handling method
  - Size (number of buckets) of hash table

8.2.2 Hash Functions

- Requirements for hash function
  - Easy to compute
  - Minimize the number of collisions
  - Unbiased
    - uniform hashing function
    - the probability of $h(k) = i$ is $1/b$ for all buckets i

8.2.2.1 Division

- The most widely used hash function in practice
- Assume that the keys are nonnegative
- The key is divided by some number D
  - $h(k) = k \% D$
  - bucket addresses : 0 ~ D–1
- At least b = D buckets

- *The number of overflows* on real-world dictionaries *is critically dependent on the choice of D*.
  - distribution of home buckets is biased whenever D has small prime factors such as 2,3,5,7
  - degree of bias decreases as the smallest prime factor of D increases

- The relaxed requirement on D
  - use odd D & *set* b = D
  - *Array doubling* results in increasing the number of buckets(and hence the divisor D)from b to $2b+1$.

8.2.2.2 Mid-Square

- Square the key
- Take an appropriate number of bits from the middle of square → use it as a key
- If r bits are used, then the table size = 2^r

ex)

```
      10100
      10100
-----------------
 0011001 0000
```

$$\begin{array}{r} 10100 \\ 10100 \\ \hline 0011001 0000 \end{array}$$

5 bits are used.

8.2.2.3 Folding

- The key k is partitioned into several parts with the same length and the partitions are added to obtain the hash address

- *Shift folding*: add all characters into one
  - Suppose k = 12320324111220
  - Partition: (x_1: 123, x_2: 203, x_3: 241, x_4: 112, x_5: 20)
  - $x_1 + x_2 + x_3 + x_4 + x_5 = 699$

- *Folding at the boundaries*: reverse every other partition before adding
  - reverse x_2, x_4 and add them
  - x_1 + reverse(x_2) + x_3 + reverse(x_4) + x_5
  - 123+302+241+211+20 = 897

8.2.2.5 Converting Keys to Integers

- To use some of the hash functions, keys need to be converted to nonnegative integers.

- Example 8.3: *[Converting String to*

```
unsigned int stringToInt(char *key)
{/* simple additive approach to create a natural number
    that is within the integer range */
   int number = 0;
   while (*key)
      number += *key++;
   return number;
}
```

Program 8.1: Converting a string into a non-negative integer

```
unsigned int stringToInt(char *key)
{/* alternative additive approach to create a natural number
    that is within the integer range */
  int number = 0;
  while (*key)
  {
    number += *key++;
    if (*key) number += ((int) *key++) << 8;
  }
  return number;
}
```

This results in a larger range for the integer returned by the function.

Program 8.2: Alternative way to convert a string into a non-negative integer

8.2.3 Overflow Handling

8.2.3.1 Open Addressing

- Find the closest unfilled bucket when overflow occur.
- ***Linear probing***, *quadratic probing*, *rehashing*, *random probing*

Example 8.4

- 13-bucket table with one slot per bucket
- hash value
  - By the scheme of Program 8.1 a hash function

Identifier	Additive Transformation	x	Hash
for	102 + 111 + 114	327	2
do	100 + 111	211	3
while	119 + 104 + 105 + 108 + 101	537	4
if	105 + 102	207	12
else	101 + 108 + 115 + 101	425	9
function	102 + 117 + 110 + 99 + 116 + 105 + 111 + 110	870	12

Using a circular rotation, the next available bucket is at *ht*[0].

[0]	**function**
[1]	
[2]	**for**
[3]	**do**
[4]	**while**
[5]	
[6]	
[7]	
[8]	
[9]	**else**
[10]	
[11]	
[12]	**if**

Linear probing

- Search the hash table buckets with s = 1 in the order $ht[(h(k)+j)\%b]$

(1) Compute $h(k)$.

(2) Examine the hash table buckets in the order $ht[h(k)]$, $ht[(h(k)+1)\ \%\ b]$, $\cdots$, $ht[(h(k)+j)\ \%\ b]$ until one of the following happens:

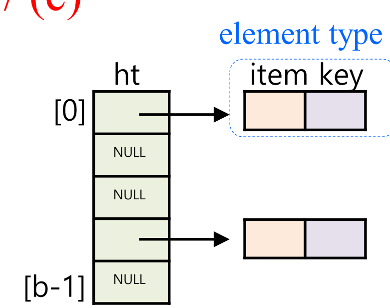
  (a) The bucket $ht[(h(k)+j)\ \%\ b]$ has a pair whose key is k; in this case, the desired pair has been found.

  (b) $ht[(h(k)+j)\%b]$ is empty; k is not in the table.

  (c) We return to the starting position $ht[h(k)]$; the table is full and k is not in the table.

Linear probing

```
element* search        )
{/* search the linear probing hash table ht (each bucket has
    exactly one slot) for k, if a pair with key k is found,
    return a pointer to this pair; otherwise, return NULL */
   int homeBucket, currentBucket;
   homeBucket = h(k); // 0 ≤ h(k) < b
   for (currentBucket = homeBucket; ht[currentBucket]
                     && ht[currentBucket]->key != k;) {
      currentBucket = (currentBucket + 1) % b;
                        /* treat the table as circular */
      if (currentBucket == homeBuket)
         return NULL; /* back to start point *// (c)
   }
   if ( ht[currentBucket] && ht[currentBucket]->key == k)
      return ht[currentBucket]; // (a)
   return NULL; // (b) ht[currentBucket] == NULL
}
```



Program 8.3: Linear probing

Linear probing

- When linear probing is used to resolve overflows, keys tend to cluster together.

- Example :
  - Input sequence: **acos, atoi, char, define, exp, ceil, cos, float, atol, floor, ctime**
  - hash function uses the first character
  - What happens when we try to enter "atol" ?

bucket	x	buckets searched
0	**acos**	1
1	**atoi**	2
2	**char**	1
3	**define**	1
4	**exp**	1
5	**ceil**	4
6	**cos**	5
7	**float**	3
8	**atol**	9
9	**floor**	5
10	**ctime**	9
...		
25		

Figure 8.4: Hash table with linear probing (26 buckets, one slot per bucket)

The average number of comparisons : 41/11 = 3.73

Linear probing

- *With uniform hash function*, the expected average number of key comparisons ρ is

$(2-\alpha)/(2-2\alpha)$, where α is the loading density.

  - Figure 8.4 : α=11/26 = 0.423, ρ =1.36

Improvements

- Quadratic probing
  - Search $h(k)$, $(h(k) + i^2)$ % b

    for $1 \le i \le (b - 1)/2$, where b is the # of buckets

    Quadratic function of i is used as the increment

    b is a prime number.

Improvements

- Rehashing
  - applying a series of hash functions h_1, h_2, ⋯, h_m to reduce clustering
  - examine buckets using $h_i(k)$ ($1 \le i \le m$) sequentially

Improvements

- Random Probing
  - The search for a key, k, in a hash table with b buckets is carried out by examining the buckets in the order $h(k)$, $(h(k)+s(i))\%b$, $1 \leq i \leq b-1$ where $s(i)$ is a pseudo random number.

  - *The random number generator* must satisfy the property that *every number from 1 to* $b-1$ must be generated *exactly once* as i ranges from 1 to $b-1$

Example

Input sequence : 5 8 13 7 21 23
Random numbers : 5 2 3 7 1 4 6
Hash table : 8 buckets with 1 slot

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
ht	8	23	13		21	5		7

k=5 : h(k) = 5%8 = 5
k=8 : h(k) = 8%8 = 0

k=13 : h(k) = 13%8 = 5
(h(k)+s(1))%8 = (5+5)%8 = 2

k=7 : h(k) = 7%8 = 7

k=21 : h(k) = 21%8 = 5
(h(k)+s(1))%8 = (5+5)%8 = 2
(h(k)+s(2))%8 = (5+2)%8 = 7
(h(k)+s(3))%8 = (5+3)%8 = 0
(h(k)+s(4))%8 = (5+7)%8 = 4

k=23 : h(k) = 23%8 = 7
(h(k)+s(1))%8 = (7+5)%8 = 4
(h(k)+s(2))%8 = (7+2)%8 = 1

8.2.3.2 Chaining

- Maintain a linked list of synonyms for each bucket
  - *s* is flexible
  - Each list contains all the synonyms for that bucket

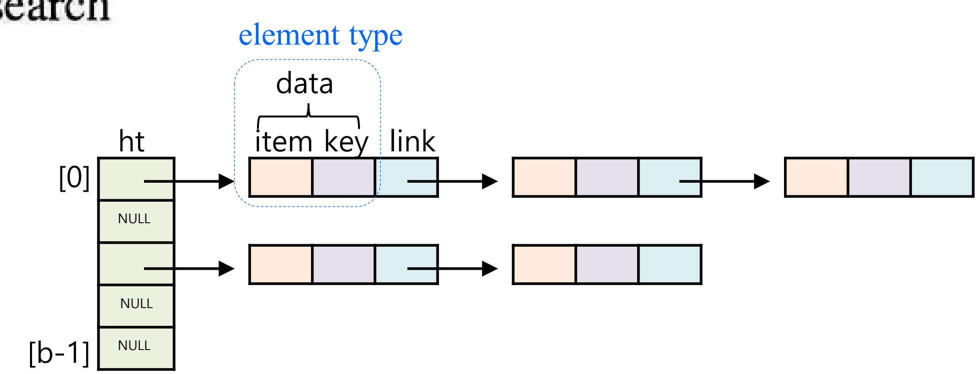
The average number of comparisons : 21/11 = 1.91

[0] → **acos atoi atol**
[1] → *NULL*
[2] → **char ceil cos ctime**
[3] → **define**
[4] → **exp**
[5] → **float floor**
[6] → *NULL*
. . .
[25] → *NULL*

Figure 8.6: Hash chains corresponding to Figure 8.4

```
element* search(int k)
{/* search the chained hash table ht for k, if a pair with
    this key is found, return a pointer to this pair;
    otherwise, return NULL.
   nodePointer current;
   int homeBucket = h(k);   // 0 ≤ h(k) < b
   /* search the chain ht[homeBucket] */
   for (current = ht[homeBucket]; current;
                                  current = current→link)
      if (current→data.key == k) return &current→data;
   return NULL;
}
```

Program 8.4: Chain search



- With uniform hash function, the expected average number of key comparisons ρ is $\approx 1+\alpha/2$.
  - Figure 8.6 : α=11/26 = 0.42, ρ =1.21

- The worst-case number of comparisons: $O(n)$
  - can be reduced to $O(\log n)$ by sorting synonyms in a balanced search tree