Algorithms and Data Structures
Lecture notes: Hash Tables, Cormen Chap. 11

Lecturer: Michel Toulouse

Hanoi University of Science & Technology
michel.toulouse@soict.hust.edu.vn

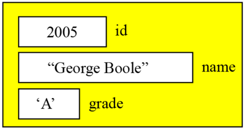
27 juin 2021

Outline

- Introduction
  - Direct addressing
  - Hash tables
- Hash functions
  - The division method
  - The multiplication method
- Handling collisions
  - Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing
- Exercises
- Hash functions as encryption methods

Motivation

The last 4 digits of your student *id* can be used as a *key* that index into an array A of pointers. Each pointer x will be the address of a record storing academic information of a student : $A[id] = x$



student

The array will have $10^4 = 10,000$ entries, if the computer memory is 64 bit addressable, this means the size of the array will be $10,000 \times 8 = 80,000$ bytes, which is a small data structure

Given a student *id*, one can find the address x of his record through $A : x = A[id]$. It cost $O(1)$ to find the record.

This is called direct addressing, a very efficient way to retrieve info in computer memory.

Problems with direct addressing

Some organizations may not provide an id for the entities with which they are interacting (like phone companies or stores) or may interact with entities using large id numbers (like government id card, 12 digits) or credit card companies (credit card numbers have 16 digits)

For example, if your phone number was used to index directly into your data, there are 10 digits, which means the array size will be $10^{10} = 10,000,000,000$ = 10 billions $\times$ 8 = 80 billions of bytes $\times$ 8 = 640 billions of bits. 6,400,000,000,000 bits is still less than the 64 bits addressable computer memory (18,446,744,073,709,551,616 bits), but it is certainly a big chunk of it

Useless to said this is not the way your info is indexed. Rather indirect addressing modes are used, one of them is called hashing (or hash tables)

Direct addressing formally defined

In direct addressing, the keys are used to index entries in a data structure. Suppose :

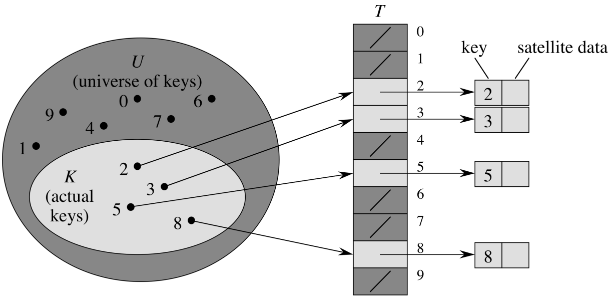
- each key is draw from a set of n numbers $U = \{0, 1, .., n-1\}$
- keys are distinct

Implementation :

- We are given an array $T[0..n-1]$ of pointers (the size of T is the same as the size of the set U)
- Each entry k of T is a pointer to an object x with key k
- If there is no object x with key k then $T[k] = NULL$, i.e. it is a null pointer

T is called a direct-addressing table, it provides search time in $O(1)$ but the size of the table can be huge and filled mostly with NULL entries

Direct addressing example



Operations take $O(1)$ time

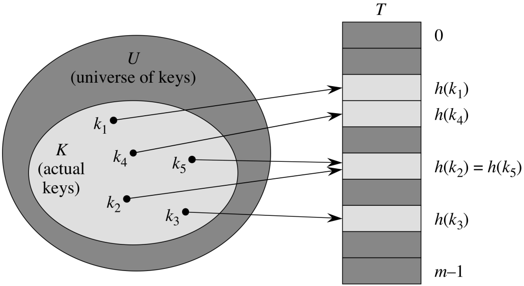
SEARCH(T, k)	INSERT(T, x)	DELETE(T,x)
return T[k]	T[key[x]] = x	T[key[x]] = NILL

Hashing : maps keys to smaller ranges

Instead of storing an element with key k in index k, use a function h and store the element in index $h(k)$

The "hash function" h maps the key from the domain $U = \{0, 1, .., n - 1\}$ into an array $T = [0, .., m - 1]$ called hash table of a much smaller range of m values

Must make sure that $h(k)$ is a legal index into the table T



Issues covered in this lecture

There are many hash functions, we describe two particular methods :

- Hash functions based on the multiplication method
- hash functions using the division method

As the range of the hash table is much smaller than the domain of the keys, some keys are mapped to the same index in the hash table, this is called collision

We study two classes of methods to handle collisions :

- chaining and
- open addressing.

Hashing keys using the division method

The hash function is the modulo operation $a \mod n$.

The dividend a is the key to be hashed while the divisor n is the size of the hash table. The hash value (digest) is the value returned by the modulo function

Assume k is the key and h denotes the hashing function.

Therefore, $h(k) = k \text{ mod } m$, where m the size of the hash table.

Example : If $k = 91$, $m = 20$, then $h(91) = k \text{ mod } 20 = 11$.

It's fast, requires just one division operation.

Disadvantage of the division method

$h(k) = k \bmod m$

Disadvantage : Have to avoid certain values of m :

- Powers of 2. If $m = 2^p$ for integer p, then $h(k)$ is just the least significant p bits of k.
- Examples : $m = 8 = 2^3$
  - h(52) = 4 : h(110100) = 100
  - h(37) = 5 : h(100101) = 101
- The implication is that all keys that end for example with 100 map to the same index (the computation of $h(k)$ depends only on the p least significant bits of the key).

Explanation

Note : The modulo function is the remainder of a division. The remainder can be computed by repeatedly subtracting the divisor until the remainder is smaller than the divisor.

The binary representation of a power of 2 is 1 follow by zeros. Therefore subtractions of a binary number by a power of 2 only impact the bits on the left of the binary number as show in the table below :

decimal	binary	-2^3	decimal remainder	binary remainder
45	101101	-1000	37	100101
37	100101	-1000	29	011101
29	011101	-1000	21	010101
21	010101	-1000	13	001101
13	001101	-1000	5	000101

All the keys where the binary representation ends with 101 will index the same entry in the table (all the keys in the above example index in the same entry of the hash table).

Solution to hash table size

- Solution : pick table size $m =$ a prime number not too close to a power of 2 (or 10). Example $m = 5$
  - h(52) = 2 : h(110100) = 010
  - h(36) = 1 : h(100100) = 001

  (the computation of $h(k)$ depends on all the bits of the key).

Table size is a prime number

Here the table size is 11, which is a prime number. The binary representation of 11 is 1011.

As we can see, each subtraction impacts the 4 rightmost bits that define the index in the hash table.

decimal	binary	1011	decimal remainder	binary remainder
45	101101	-1011	34	100010
34	100010	-1011	23	010111
23	010111	-1011	12	001100
12	001100	-1011	1	000001

The further away the divisor is from a power of two, the more the bits of the binary representation of the divisor are equally 0 and 1.

Also the more digits of the remainder are affected by each subtraction.

Division method applied on strings of characters

Most hashing functions assume that the key is a natural number. Here we show how a hashing function like the division method can be applied on keys that are strings of characters.

Recall : A number in base 10 is
$6789 = (6 \times 10^3) + (7 \times 10^2) + (8 \times 10^1) + (9 \times 10^0) = 6789$

Western computer keyboards have 128 characters, a string of characters can be converted into integers using base 128 (7-bit ASCII values). For example, "CLRS", ASCII values are : $C = 67$, $L = 76$, $R = 82$, $S = 83$

The string "CLRS" is interpreted as the integer
$(67 \times 128^3) + (76 \times 128^2) + (82 \times 128^1) + (83 \times 128^0) = 141,764,947$

Hash fct : multiplication method

The multiplication method is a hashing function addressing the above difficulties. It works as follows :

1. Choose a constant A in the range $0 < A < 1$
2. Multiply key k by A
3. Extract the fractional part of kA
4. Multiply the fractional part by m
5. Take the floor of the result

$h(k) = \lfloor m(kA \bmod 1) \rfloor$

Examples for $m = 8 = 2^3$ and $A = 0.6180$:

$$\begin{aligned} h(52) &= \lfloor 8(52 \times 0.6180 \bmod 1) \rfloor \\ &= \lfloor 8(32.136 \bmod 1) \rfloor \\ &= \lfloor 8(0.136) \rfloor \\ &= \lfloor 1.088 \rfloor \\ &= 1 \end{aligned}$$

$$\begin{aligned} h(36) &= \lfloor 8(36 \times 0.6180 \bmod 1) \rfloor \\ &= \lfloor 8(22.248 \bmod 1) \rfloor \\ &= \lfloor 8(0.248) \rfloor \\ &= \lfloor 1.984 \rfloor \\ &= 1 \end{aligned}$$

Multiplication method : exercises

Compute the hash value of the keys below using the multiplication method :

1. key $= 196$, $m = 8$, $A = 0.22$
2. key $= 353$, $m = 11$, $A = 0.75$

Advantage, disadvantage, implementation considerations

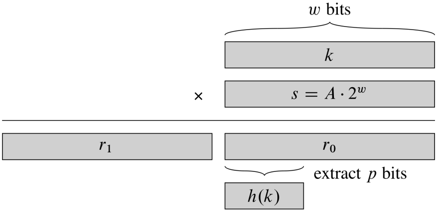
Advantage : Value of m not critical

Disadvantage : Slower than division method, but can be adapted to make efficient use of the computer architecture design

Multiplication method : implementation

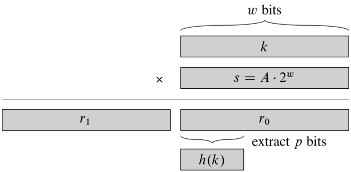
Assume memory words have w bits and the key k fits in a single word

Choose A to be a fraction of the form $\frac{s}{2^w}$ where s is an integer in the range $0 < s < 2^w$



- Multiplying s and k, the result is 2w bits, $r_1 2^w + r_0$
- r_1 is the high-order word of the product and r_0 is the low-order word
- $\lfloor m(kA \bmod 1) \rfloor$ are the p most significant bits of r_0

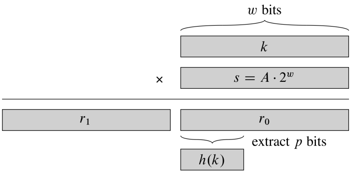
Multiplication method : implementation



Example : $m = 8 = 2^3$ $(p = 3)$; $w = 5$; $k = 21$; $0 < s < 2^5$; $s = 13 \Rightarrow A = 13/32 = 0.40625$

$$
\begin{aligned}
h(21) &= \lfloor 8(21 \times 0.40625 \bmod 1) \rfloor & &= ks = 21 \times 13 = 273 \\
&= \lfloor 8(8.53125 \bmod 1) \rfloor & &= 8 \times 2^5 + 17 \\
&= \lfloor 8(0.53125) \rfloor & &= r_1 = 8,\ r_0 = 17 = 10001 \\
&= \lfloor 4.25 \rfloor & &= \text{ takes the } p = 3 \text{ most significant bits of } r_0 \\
&= 4 & &= 100 = 4
\end{aligned}
$$

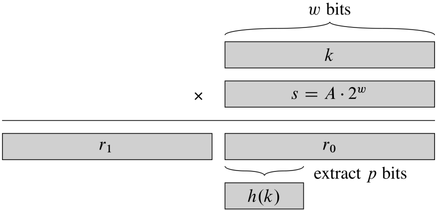
Multiplication method : implementation



Example : $m = 8 = 2^3$ $(p = 3)$; $w = 6$; $k = 52$; $0 < s < 2^6$; $A = 0.625 \Rightarrow s = 0.625 \times 2^6 = 40$

$$\begin{aligned} h(52) &= \lfloor 8(52 \times 0.625 \bmod 1) \rfloor && = ks = 52 \times 40 = 2080 \\ &= \lfloor 8(32.5 \bmod 1) \rfloor && = 2080 = 32 \times 2^6 + 32 \\ &= \lfloor 8(0.5) \rfloor && = r_1 = 32, r_0 = 32 = 100000 \\ &= \lfloor 4 \rfloor && = \text{ takes the } p = 3 \text{ most significant bits of } r_0 \\ &= 4 && = 100 = 4 \end{aligned}$$

Multiplication method : implementation



Example :
$m = 8 = 2^3;\ \ w = 6;\ \ k = 52;\ \ 0 < s < 2^6;\ \ s = 40;\ \ A = 0.625$

We extract the p most significant bits of r_0 because we have selected the table to be of size $m = 2^p$, therefore we need p bits to generate a number in the range $0..m - 1$.

Collisions

Collisions occur when two keys hash to a same index in the hash table

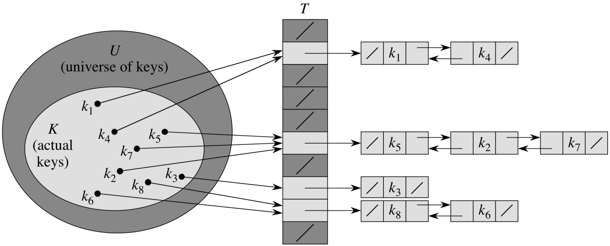
Collisions certainly happen when the number of keys to store is larger than the size m of the hash table T.

- If the number of keys to store is $< m$, collision may or may not happen (i.e. collisions occur even thought the table is not full)

Indexing in hash tables always needs to handle collisions. Two methods are commonly used : chaining and open addressing.

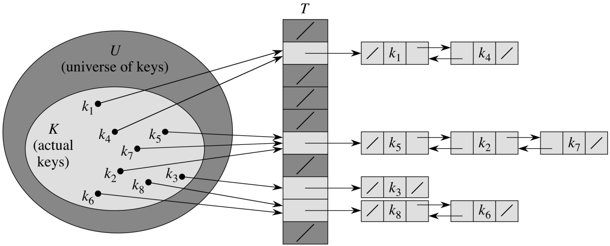
Chaining

Chaining puts elements that hash to the same index in a linked list :



SEARCH(T, k)
search for an element with key k in list $T[h(k)]$. Time proportional to length of the list of elements in $h(k)$
INSERT(T, x)
insert x at the head of the list $T[h(key[x])]$. Worst-case $O(1)$

Chaining



DELETE(T,x)

delete x from the list $T[h(key[x])]$. If lists are singly linked, then deletion takes as long as searching, $O(1)$ if doubly linked

Analysis of running time for chaining

load factor : Given n keys and m indexes in the table : the $\alpha = n/m =$ average # keys per index

Worst case : All keys hash in the same hash table entry, creating a chain of length n. Searching for a key takes $\Theta(n)$ + the time to hash the key. We don't do hashing for the worst case !

Average case analysis

A good hash function satisfies (approximately) the assumption of **simple uniform hashing** :

"*given a hash function h, and a hash table of size m, the probability that two non-equal keys a and b will hash to the same slot is* $P(h(a) = h(b)) = \frac{1}{m}$"

In order words, each key is equally likely to hash in any of the m slots of the hash table

Under the assumption of uniform hashing, the load factor α and the average chain length of a hash table of size m with n elements will be $\alpha = \frac{n}{m}$

Average case analysis

Assumption : Assume simple uniform hashing : each key in table is equally likely to be hashed to any index

- The average cost of an unsuccessful search for a key is $\Theta(1+\alpha)$
- The average cost of a successful search is $1+\alpha/2 = \Theta(1+\alpha)$

If the size m of the hash table is proportional to the number n of elements in the table, say $\frac{1}{2}$, then $n = 2m$, and we have $n \in O(m)$

Consequently, $\alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1)$, in other words, if hash function distribution is uniform, the average time for searching is constant provided the number of elements in the hash table is a multiplicative factor of the size of the table.

Open addressing

When collisions are handled through chaining, the keys that collide are placed in a link list in the same entry where the collision occurred

Open addressing place the keys that collide in another entry of the hash table by searching an empty slot using a *probe sequence*

The hash function is modified to include a probe number i, $h(k, i)$. The value of i increases by 1 from 0 to $m - 1$.

Open addressing : Insertion

```
HASH-INSERT(T, k)
  i = 0
  repeat
    j = h(k, i)
    if T[j] == NIL
      T[j] = k
      return j
    else i = i+1 // probing
  until i == m
  error "hash table overflow"
```

Open addressing : Searching

```
HASH-SEARCH(T, k)
    i = 0
    repeat
        j = h(k, i)
        if T[j] == k
            return j
        else i = i+1
    until T[j] == NIL or i == m
    return NIL
```

Computing probe sequences

Probe sequences determine how empty slots in the hash table are searched. We describe three probe sequence techniques :

- Linear probing
- Quadratic probing
- Double hashing

Linear probing

The probe sequence $h(k,i)$ is constructed from an ordinary hash function $h(k)$

Linear probing $h(k,i) = (h(k)+i) \mod m = (k \mod m) + i \mod m$ for $i = 0, 1, \ldots, m-1$, thus $h(k,0)$ is the initial hashing. If $h(k,0) \neq NIL$, then there is a collision.

Linear probing resolves collisions by looking at the next entry in the table.

Assume the state of the hash table is as below and $h = k \mod 11$:

0	1	2	3	4	5	6	7	8	9	10
44	56	NIL	NIL	81	NIL	39	29	52	NIL	21

Linear probing

A call to $h(28, 0) = 6$ yields a collision. Linear probing examines entries 7, 8 and finally 9.

$h(28, 1) = 7$

0	1	2	3	4	5	6	7	8	9	10
44	56	NIL	NIL	81	NIL	39	29	52	NIL	21
						↑	↑			

$h(28, 2) = 8$

0	1	2	3	4	5	6	7	8	9	10
44	56	NIL	NIL	81	NIL	39	29	52	NIL	21
								↑		

$h(28, 3) = 9$

0	1	2	3	4	5	6	7	8	9	10
44	56	NIL	NIL	81	NIL	39	29	52	NIL	21
									↑	

Linear probing : exercise

Assume the hash table has size $m = 7$ and $h(k) = k \mod 7$. Insert keys 701, 145, 217, 19, 13, 749 in the table, using linear probing

Linear probing : primary clustering

A cluster is a collection of consecutive occupied slots

A cluster that covers the hash of a key is called the primary cluster of the key

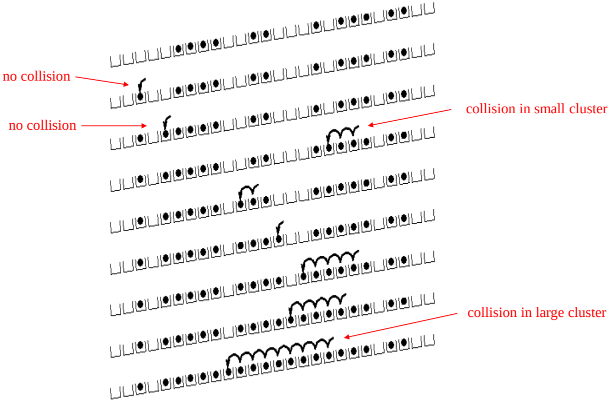
Linear probing can create large primary clusters that will increase the running time of search/insert/delete

Example : hash table has size $m = 7$ and $h(k) = k \mod 7$. Insert keys 14, 15, 1, 35, 29

0	1	2	3	4	5	6
14	15	1	35	29		

Linear probing : clustering

As the hash table starts to fill, the number of clusters increases, then they merge, becoming larger



[R. Sedgewick]

Quadratic probing

Linear probing suffers from primary clustering : long runs of occupied sequences build up : an empty slot that follows i full slots has probability $\frac{i+1}{m}$ to be filled.

Quadratic probing jumps around in the table according to a quadratic function of the probe number : $h(k, i) = (h(k) + c_1 i + c_2 i^2) \bmod m$, where $c_1, c_2 \neq 0$ are constants and $i = 0, 1, \ldots, m-1$. Thus, the initial position probed is $T[h(k)]$.

Quadratic probing

Assuming $c_1 = c_2 = 1$, in this case, the probe sequence will be $h(k, i) = (h(k) + c_1 i + c_2 i^2) \bmod m$

$h(28, 0) = ((28 \bmod 11) + 0 + 0) \bmod 11) = 6 =$ collision

0	1	2	3	4	5	6	7	8	9	10
44	NIL	57	NIL	81	NIL	39	29	52	NIL	21
						↑				

$h(28, 1) = ((28 \bmod 11) + 1 + 1) \bmod 11) = 8 =$ collision

0	1	2	3	4	5	6	7	8	9	10
44	NIL	57	NIL	81	NIL	39	29	52	NIL	21
								↑		

$h(28, 2) = ((28 \bmod 11) + 2 + 4) \bmod 11) = 1$

0	1	2	3	4	5	6	7	8	9	10
44	NIL	57	NIL	81	NIL	39	29	52	NIL	21
	↑									

Quadratic probing : exercises

Assume the hash table has size $m = 7$ and $h(k) = k \mod 7$, $c_1 = c_2 = 1$, so the quadratic probing function is

$$h(k,i) = ((k \mod m) + c_1 i + c_2 i^2) \mod m.$$

1. Insert keys 76, 40, 48, 5, 55
2. Using the same quadratic probing function and same hash table (empty), insert keys 701, 145, 217, 19, 13, 749

Quadratic probing issues : space inefficiency

If the table T is less than half full and m is a prime number, then quadratic probing will find an empty entry in the table.

This is because for i in $[0, \frac{m-1}{2}]$, $h(k,i)$ will never return twice the same index in the table if the load factor $\alpha \leq \frac{1}{2}$ (you can find a proof of this statement in different places, there is one in the section limitations of `https://en.wikipedia.org/wiki/Quadratic_probing`)

If the load factor $\alpha > \frac{1}{2}$, then quadratic probing may find an empty entry

Example : Insert keys 14,15,35,1,5,3

0	1	2	3	4	5	6
14	15	35	1		5	

$h(3,0) = 3$, $h(3,1) = 5$, $h(3,2) = 2$, $h(3,3) = 1$, $h(3,4) = 2$, $h(3,5) = 5$, $h(3,6) = 3$

Conclusion : The size of the table must be expanded each time the load factor exceeds $\frac{1}{2}$ if no other technique is used

Quadratic probing issues : secondary clustering

In quadratic probing, clusters are formed along the path of probing, instead of around the initial hash value of a key

These clusters are called secondary clusters

Secondary clusters are formed as a result of using the same pattern in probing by all keys. If two keys have the same initial hash value, their probe sequences are going to be the same

But this is a milder form of clustering compares to primary clustering

Double hashing

Use two auxiliary hash functions, h_1 and h_2.

h_1 gives the initial probe, and h_2 gives the remaining probes :
$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$

Issue : Must have the hash value $h_2(k)$ relatively prime to m (i.e. no factors in common other than 1). To satisfy this requirement we

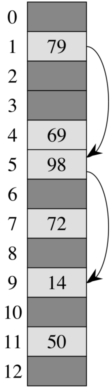
1. could choose m to be a power of 2 and h_2 to always produce an odd number > 1
   - The factors of $m = 2^x$ are 2^{x-y}, $y = 1, 2, \ldots, x - 1$. The factors of $m = 2^5$ are $2^4, 2^3, 2^2$ and 2^1
   - If $h_2(k)$ is an odd number, it cannot be a factor of $m = 2^x$
2. or m prime and h_2 to always produce an integer less than m
   - If $h_2(k) < m$, since m is prime, none of the values greater than 1 and smaller than m can be a factor of m

Handling collisions : Double hashing

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$$

Example : $m = 13, k = 14, h_1(k) = k \bmod 13,$
$h_2(k) = (k \bmod 11) + 1$. For $i = 0$

$$\begin{aligned} h(14,0) &= h_1(14) + 0(h_2(14)) \\ &= (14 \bmod 13) + \\ &\quad 0((14 \bmod 11) + 1) \\ &= 1 + 0(3+1) \\ &= 1 \end{aligned}$$



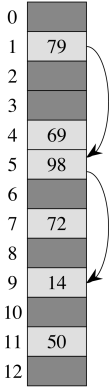
0	
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

Handling collisions : Double hashing

$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$

Example : $m = 13, k = 14, h_1(k) = k \bmod 13,$
$h_2(k) = (k \bmod 11) + 1$. For $i = 1$

$$\begin{aligned} h(14, 1) &= h_1(14) + 1(h_2(14)) \\ &= (14 \bmod 13) + \\ &\quad 1((14 \bmod 11) + 1) \\ &= 1 + 1(3 + 1) \\ &= 5 \end{aligned}$$

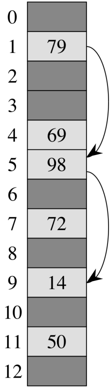


Handling collisions : Double hashing

$h(k, i) = (h_1(k) + ih_2(k)) \text{ mod } m$

Example : $m = 13, k = 14, h_1(k) = k \text{ mod } 13,$
$h_2(k) = (k \text{ mod } 11) + 1$. For $i = 2$

$$\begin{aligned} h(14, 2) &= h_1(14) + 2(h_2(14)) \\ &= (14 \text{ mod } 13) + \\ &\quad 2(1 + (14 \text{ mod } 11)) \\ &= 1 + 2(3 + 1) \\ &= 9 \end{aligned}$$



Double hashing : exercises

Assume the hash table has size $m = 7$. $h_1(k) = k \mod 7$ and $h_2(k) = 5 - (k \mod 5)$, thus $h(k, i) = (h_1(k) + ih_2(k)) \mod 7$.

1. Insert the following sequence of keys : 76, 93, 40, 47, 10, 55
2. Insert the following sequence of keys : 701, 145, 217, 19, 13, 749

Recalibrating the size of the table

If the table is half full, probing cost $\frac{1}{(1-.5)} = 2$ on average. If the table is 90 percent full, then it cost is $\frac{1}{(1-.9)} = 10$.

When the table gets too full, inserting and searching for a key become too costly. We should consider to increase the size of the table

Each time we increase the size of the table we should re-hash all the keys as the new modulo m of the larger table is not the same as the modulo of the original table

Open addressing : Deleting

Use a special value DELETED instead of NIL when marking an index as empty during deletion.

- Suppose we want to delete key k at index j.
- And suppose that sometime after inserting key k, we were inserting key k', and during this insertion we had probed index j (which contained key k).
- And suppose we then deleted key k by storing NIL into index j .
- And then we search for key k'.
- During the search, we would probe index j before probing the index into which key k' was eventually stored.
- Thus, the search would be unsuccessful, even though key k' is in the table.

Exercises

1. Given the values 2341, 4234, 2839, 430, 22, 397, 3920, a hash table of size 7, and hash function h(x) = x mod 7, show the resulting tables after inserting the values in the above order with each of these collision strategies :
   1.1 Chaining
   1.2 Linear probing
   1.3 Quadratic probing where $c_1 = 1$ and $c_2 = 1$
   1.4 Double hashing with second hash function $h_2(x) = (2x - 1) \mod 7$
2. Suppose you a hash table of size $m = 9$, use the division method as hashing function $h(x) = x \mod 9$ and chaining to handle collisions. The following keys are inserted : 5, 28, 19, 15, 20, 33, 12, 17, 10. In which entries of the table do collisions occur ?

Exercises

3. Now suppose you use the same hashing function as above with linear probing to handle collisions, and the same keys as above are inserted. More collisions occur than in the previous question. Where do the collisions occur and where do the keys end up ?

Exercises

4. Fill a hash table when inserts items with the keys D E M O C R A T in that order into an initially empty table of m = 5, using chaining to handle collisions. Use the hash function $11k \mod m$ to transform the kth letter of the alphabet into a table index, e.g., $hash(I) = hash(9) = 99 \mod 5 = 4$
5. Fill a hash table when inserting items with the keys R E P U B L I C A N in that order into an initially empty table of size m = 16 using linear probing. Use the hash function $11k \mod m$ to transform the kth letter of the alphabet into a table index.

Exercises

6. Suppose you use one of the open addressing techniques to handle collisions and you have inserted so many keys/values into your hash table such that all entries are taken. Then collisions occur everytime. What can you do ?
7. Suppose a hash table with capacity $m = 31$ gets to be over .75 full. We decide to rehash. What is a good size choice for the new table to reduce the load factor below .5 and also avoid collisions ?

Hash functions as cryptographic functions

Hash functions have two interesting properties

1. Even if we know the output (hash, digest) we cannot guess the input (the key). Such function are non-invertible, given $f(x)$ we cannot find the inverse $f^{-1}(x)$
2. The output, the digest, always has the same length no matter what was the length of the input (key)

If furthermore a hashing function is *collision resistant*, i.e. the likelihood of a collision is very small, then the hash function can be used to encrypt information.

Cryptographic hash functions

Cryptographic hash functions are hash functions designed specifically to encrypt data

Among them there are two specific classes of cryptographic hash functions known as *Secure Hashing Algorithm* (SHA) : SHA-2 and SHA-3

For example, SHA-2 is a family of hash functions that includes SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256

The number after SHA indicates the length in bits (the number of bits) of the digest produced by each of these cryptographic hash functions

Note : the digest is usually represented in hexadecimal

SHA-2 cryptographic functions

SHA-2 cryptographic functions have another property that make then useful to crypto-currency platforms : a single small change in the key can make a huge difference in the digest produced

For example, adding a period to the end of the sentence below changes almost half (111 out of 224) of the bits in the digest :

SHA-224(”The quick brown fox jumps over the lazy dog”)
730e109bd7a8a32b1cb9d9a09aa2325d2430587ddbc0c38bad911525

SHA-224(”The quick brown fox jumps over the lazy dog.”)
619cba8e8e05826e9b8c519c0a5c68f4fb653e8a3d8aa04bb2c8cd4c