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

15 décembre 2020

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
- Appendix : Blockchains
  - Cryptographic hash functions
  - Blockchains
  - Application of blockchains : Crypto-currencies

Motivation

Suppose a phone company want to provide the ”caller id service”, which given a phone number returns the caller’s name

This service can be implemented using a ”balance” (max depth $O(\log n)$) search tree where the keys are the phone numbers. Search time $O(\log n)$ and space $O(n)$.

In Vietnam the number of different phone numbers is 10^{10} which is quite substantial, $\log_2 10^{10} \approx 33$

The service can be implemented with search time $O(1)$ using direct addressing or hash tables

Direct addressing

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

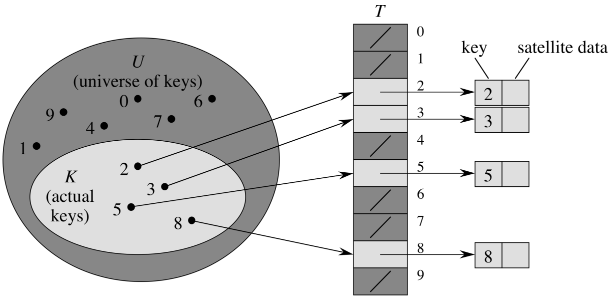
- each key is draw from a set $U = \{0, 1, .., m - 1\}$
- keys are distinct

Implementation :

- We are given an array $T[0..m - 1]$ 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 space can be lost as the table could be huge filled mostly with NULL entries

Direct addressing example



Operations take $O(1)$ time

SEARCH(T, k)
return T[k]

INSERT(T, x)
T[key[x]] = x

DELETE(T,x)
T[key[x]] = NILL

Problem with direct addressing

Direct addressing works well when the range m of keys is relatively small

But if the keys are 32-bit integers

- Problem 1 : direct-address table will have 2^{32} entries, more than 4 billion
- Problem 2 : even if memory is not an issue, the time to initialize the elements to NULL may be

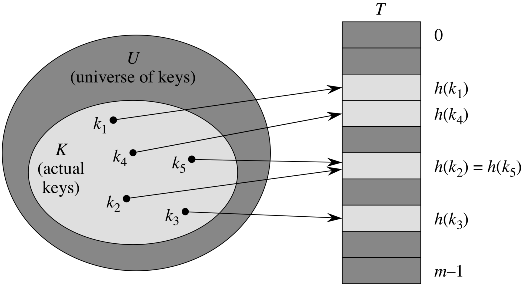
Solution : map the universe of keys to a smaller range $0..m-1$

This mapping is performed by a *hash function*

Hashing : mapping 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 function h is the hash function.
- $h : U \to \{0, 1, \ldots, m-1\}$ so that h[k] is a legal index in T.
- We say that k hashes to index $h(k)$



Hash table issues

Solution : map the universe of keys to a smaller range $0..m-1$ called a *hash table*

This mapping is performed by a hash function

- How to compute hash functions.
  - We will look at the multiplication and division methods.
- The hash function could map multiple keys to the same table index.
  - We will look at two methods to address these ”collisions” : chaining and open addressing.

Hashing keys using the division method

The hashed value is the reminder of a division, i.e. the modulo. The divisor is the size of the hash table.

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

Therefore, $h(k) = k \bmod m$, where m the size of the hash table.

Example : Assume $k = 91$, $m = 20$, then $h(91) = k \bmod 20 = 11$.

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 for 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
$(67 \times 128^3) + (76 \times 128^2) + (82 \times 128^1) + (83 \times 128^0) = 141,764,947$

Difficulties with this approach

$128 = 2^7 = 2^p$, as usual $h(k)$ is the least significant p bits of k.

Same issue as previously, two different strings of characters may yield the same key in the hash table.

So CLRS as $(67 \times 128^3) + (76 \times 128^2) + (82 \times 128^1) + (83 \times 128^0) \bmod 2^7 = 100001110011001010010\mathbf{1010011} = 83 = 1010011$

ABCS as $(65 \times 128^3) + (66 \times 128^2) + (67 \times 128^1) + (83 \times 128^0) \bmod 2^7 = 100000110000101000010\mathbf{1010011} = 83 = 1010011$

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 $A = 0.6180$:

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

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

Advantage, disadvantage, implementation considerations

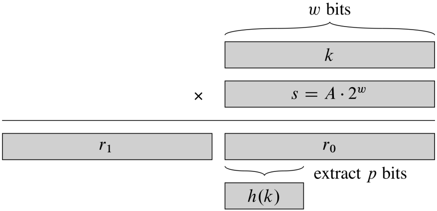
Advantage : Value of m not critical

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

Multiplication method : implementation

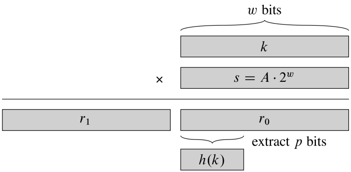
Assume memory words have w bits and the key k fits in a single word

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

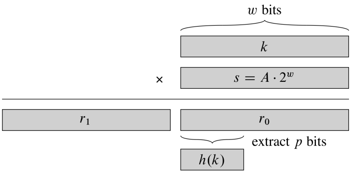
Multiplication method : implementation



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

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

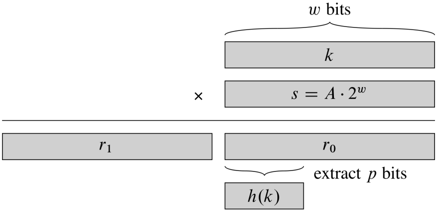
Multiplication method : implementation



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



w bits

k

× $s = A \cdot 2^w$

r_1 r_0

extract p bits

$h(k)$

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 $0..m-1$.

Other hash functions

- **Trivial hash function :** Keys are closed to uniformly distributed over the space of keys, take any bits in the key and collide them together to form the index into the hash table.
- **Mid-squares :** Raises the key (k^2), for a table of 100 entries, takes the two middle digits, uses the corresponding number as index in the table
- **Cryptographic hash functions :** SHA (Secure Hash Algorithm), a family of cryptographic hash functions, it includes SHA-1 and SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512) where the number after SHA indicates the length in bits of the digest (*digest* is what is returned by a hashing function). *These are used to encrypt data, not to index into a table*.

Collisions

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

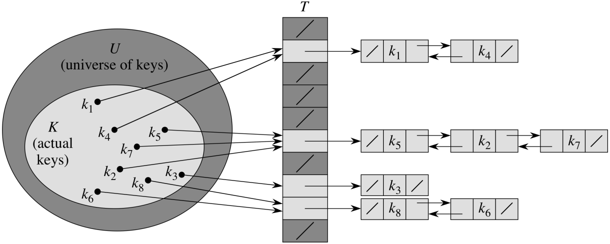
Collisions certainly happen when the number of keys to store is larger than the size m of the hash table T.

- 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

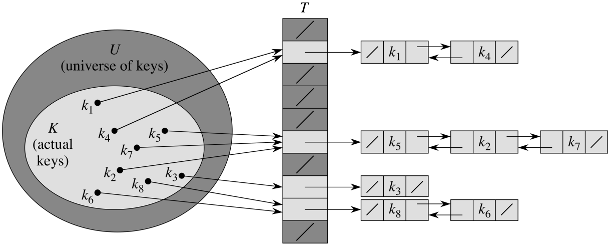
Chaining puts elements that hash to the same index in a linked list :



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 $m - 1$.

The probe sequence $< h(k, 0), h(k, 1), \ldots, h(k, m - 1) >$ should be a permutation of $< 0, 1, \ldots, m - 1 >$ so every position in the table is eventually considered

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

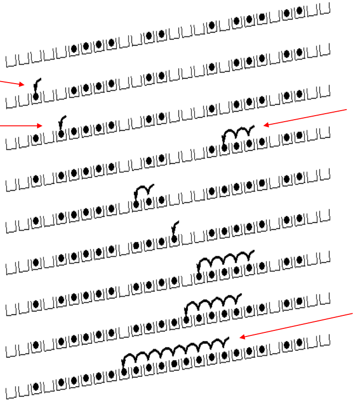
Linear probing can create large primary clusters that will increase the running time of search/insert/delete

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



no collision

no collision

collision in small cluster

collision in large cluster

[R. Sedgewick]

1

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

If $\alpha < \frac{1}{2}$ then quadratic probing will find an empty entry in the table T

If $\alpha > \frac{1}{2}$ 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	

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

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) in order to guarantee that the probe sequence is a full permutation of $[0, 1, \ldots, m - 1]$

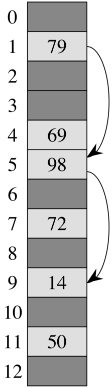
- Could choose m to be a power of 2 and h_2 to always produce an odd number > 1.
- or m prime and h_2 to always produce an integer less than m

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 $i = 0$

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



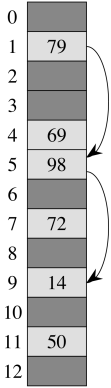
0	
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

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 $i = 1$

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

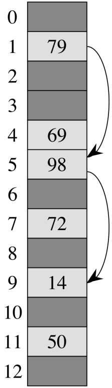


Handling collisions : Double hashing

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

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

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



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

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

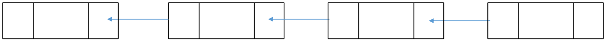
SHA-2 cryptographic functions have another property that make then useful to crypto-currency platforms : a single small change in the key can make a huge difference in the digest produced

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

Blockchains



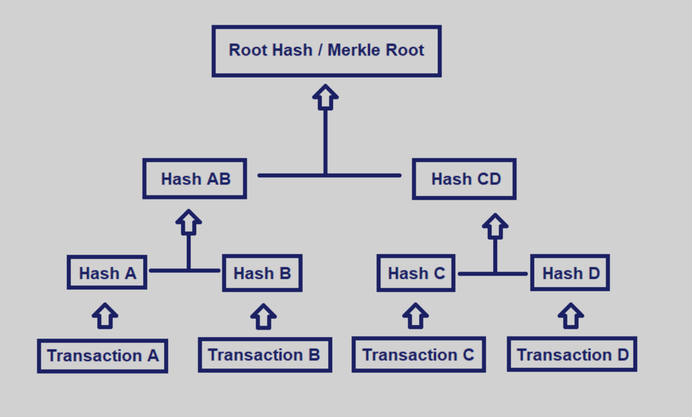
A blockchain is a link list. Nodes of the link list are refereed to in the literature as ”blocks”

This link list has no remove operator, blocks can only be added to the list, the list is constantly growing

The first block is the leftmost one, new blocks are added after the rightmost block

Blocks in a blockchain

A block is a binary tree called ”Merkle” tree



Merkle tree

In a Merkle tree, leaves store data (the data stored in the blockchain) while each internal node stores the hash of its two children

The hashing function used is often SHA-256.

The key of the root node consists at least of the hash of its two children, the hash of the previous block in the blockchain, a time stamp and a nonce (an integer).

The digest of the root node must satisfy a very stringent property : the digest must contain a specific number of leading zero's such as this one

0000000000000e9b8c519c0a5c68f4fb653e8a3d8aa04bb2c8cd4c

Brute force hashing the Merkle tree root node

Remember that the output of SHA functions is unpredictable, a small change in the input can cause a huge difference in the digest

Therefore, the only way to obtain a digest with a specific number of leading zero's is by brute force hashing the input of the root, i.e. iteratively re-hashing the input with a different nonce at each iteration

Brute force hashing is performed until a digest with the required number of leading zero-s is produced

The root is made difficult to hash to provide security on the blockchain against certain types of network attacks such as sybil attacks as well as data manipulation in the context of crypto-currency such as "double spending"

Crypto-currency : bitcoin

One set of applications for blockchains is distributed crypto-currency such as *Bitcoin* and *Ethereum*

A crypto-currency is money that exist only in computer files which are protected by crytographic algorithms. Bitcoin is a well-known crypto-currency, it is used to paid mainly for business transactions

Bitcoin uses the blockchain to store a digital *ledger*, i.e. a record of all transactions made between business entities

An example of transactions is you paying for a coffee at Starbucks, the ledger will record that you spent x bitcoins while the Starbucks coffee shop has received x bitcoins

Blockchain in bitcoin

The digital ledger of bitcoin, i.e. a very long list of transactions, is stored in the blocks of a blockchain

Each block, i.e. the leaves of the Merkle tree, store a small set of transactions

When transactions are generated, they are sent on the bitcoin network where computer nodes collect and put them in a block.

Once there is enough transactions to fill a block, the transactions are hashed using SHA-256 up to the root, which is then hashed itself until the digest meet the number of leading zero's requirement

Then the block is added to the blockchain, and in principle it can never be removed

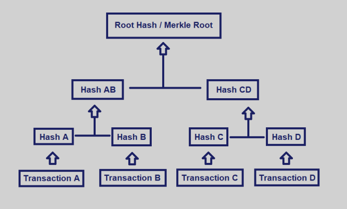
Verifying encryption

It is difficult to generate the digest for the root of Merkle tree.

However, verifying that the root digest is correct is very simple :

- one just needs to take the input string of the root, including the nonce,
- run SHA-256 on it and verify that the digest is the same as the one that is stored in the root node of the block

Blockchains are immutable



If one transaction is modified in the Merkle tree, then the chain of hashings along the path from that transaction to the root will generate different digests

The input string provided to the verification of the root digest will be different then the one that has been used to computed the current root digest, therefore detecting the corrupted data.

Attacking immutability : double spending

Bitcoin nodes have special software and sometime hardware that calculates how many bitcoins an account has

This software goes through the recorded transactions in the digital ledger

Since bitcoins only exist in digital files, it is tempting to try to spend more than one time a bitcoin, this is called double spending

Next slide is an example of how it works

Example of double spending

You go to Starbucks, buy a coffee and paid with bitcoins. This transaction will be stored in a block B and will said in particular that you have x bitcoins less to spend.

Next, you go sit at a table, connect on the Starbucks wifi network and modify the transaction in block B, saying for example that you didn't spend bitcoins for the coffee transaction.

Then you can spend again the same bitcoins, double spending !

Protection against double spending

To succeed to double spend the same bitcoin, you must also re-hash all the internal nodes in the subtree of the Merkle tree that contains the transaction.

Then you must re-hash the root node of the block. This is extremely difficult

Then if blocks have been added after the block that contains your transaction, then you must also re-hash the root of each of them

This tasks is almost impossible, this is why transactions stores in blockchain are considered immutable

Double spending is a real problem

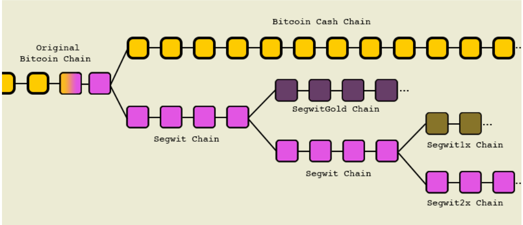
As you might guess double spending cannot be executed as just described

Nonetheless it is a real problem for crypto-currencies like bitcoin

The strategy consists to invalidate the block that contains the transaction you want to erase

This is achieved by creating ”forks” in the blockchain. But this is a long story...

BITCOIN’S MANY FORKS


Bitcoin Cash Chain
Original Bitcoin Chain
SegwitGold Chain
Segwit Chain
Segwit1x Chain
Segwit Chain
Segwit2x Chain