COMP90038
Algorithms and Complexity

Lecture 17: Hashing

(with thanks to Harald Søndergaard & Michael Kirley)

Casey Myers
Casey.Myers@unimelb.edu.au
David Caro Building (Physics) 274

Review from Lecture 16: Sorting by Counting

- We can now create a sorted array $S[1, \cdots, n]$ of the items by simply slotting items into pre-determined slots in S (a third linear scan).

$$A = 6\ 3\ 3\ 8\ 1\ 0\ 8\ 7\ 9\ 2\ 5\ 3\ 5\ 3\ 1\ 8\ 7\ 6\ 5\ 1\ 2\ 1\ 5\ 3$$

key	0	1	2	3	4	5	6	7	8	9
$Cumu$	0	1	5	7	12	12	16	18	20	23

- Place the first record (with key 6) in S[18] and decrement $Cumu[6]$ (so that the next ‘6’ will go into slot 17), and so on.

for $i \leftarrow 1$ **to** n **do**
  $S[Cumu[A[i]]] \leftarrow A[i]$
  $Cumu[A[i]] \leftarrow Cumu[A[i]] - 1$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
6	3	3	8	1	0	8	7	9	2	5	3	5	3	1	8	7	6	5	1	2	1	5	3
0	1	1	1	1	2	2	3	3	3	3	3	5	5	5	5	6	6	7	7	8	8	8	9

Review from Lecture 16: Horspool's String Search Algorithm

- Comparing from right to left in the pattern.

- Very good for random text strings

S T R I N G S E A R C H E X A M P
E X A M

- We can do better than just observing a mismatch here.

- Because the pattern has no occurrence of I, we might as well slide it 4 positions along.

- This is based only on knowing the pattern.

Review from Lecture 16: Horspool's String Search Algorithm

function $\textsc{Horspool}(P[0, \cdots, m-1], m, T[0, \cdots, n-1], n)$
    $\textsc{FindShifts}(P, m)$
    $i \leftarrow m - 1$
    while $i < n$ **do**
        $k \leftarrow 0$
        while $k < m$ **and** $P[m-1-k] = T[i-k]$ **do**
            $k \leftarrow k + 1$
        if $k = m$ **then**
            return $i - m + 1$
        else
            $i \leftarrow i + \mathit{Shift}[T[i]]$
    return -1

```
JIM_SAW_ME_IN_A_BARBERSHOP
BARBER
    BARBER
     BARBER
           BARBER
             BARBER
                BARBER
```

Pattern: BARBER
Text: JIM_SAW_ME_IN_A_BARBERSHOP

Character	A	B	C	D	E	F	...	R	...	Z	_
Shift	4	2	6	6	1	6	6	3	6	6	6

Improving Searching

- Array or linked list
  - Access by position
    - $O(n)$
    - Can we do better?

Improving Searching

- Array or linked list
  - Access by position
    - $O(n)$
    - Can we do better?

- Sorted list/array
  - Binary search
  - $O(\log n)$

Improving Searching

- Array or linked list
  - Access by position
    - $O(n)$
    - Can we do better?

- Sorted list/array
  - Binary search
  - $O(\log n)$

- Binary search tree
  - $O(\log n)$ — if balanced

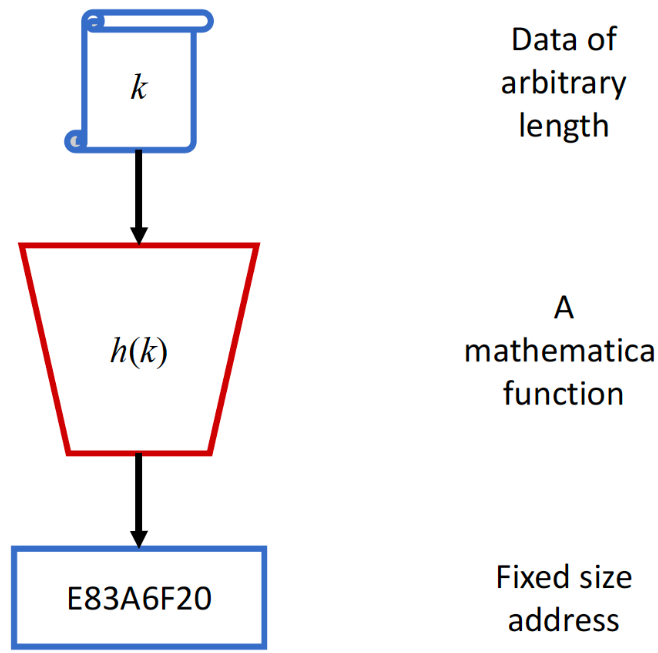
Improving Searching

- Array or linked list
  - Access by position
    - $O(n)$
    - Can we do better?

- Sorted list/array
  - Binary search
  - $O(\log n)$

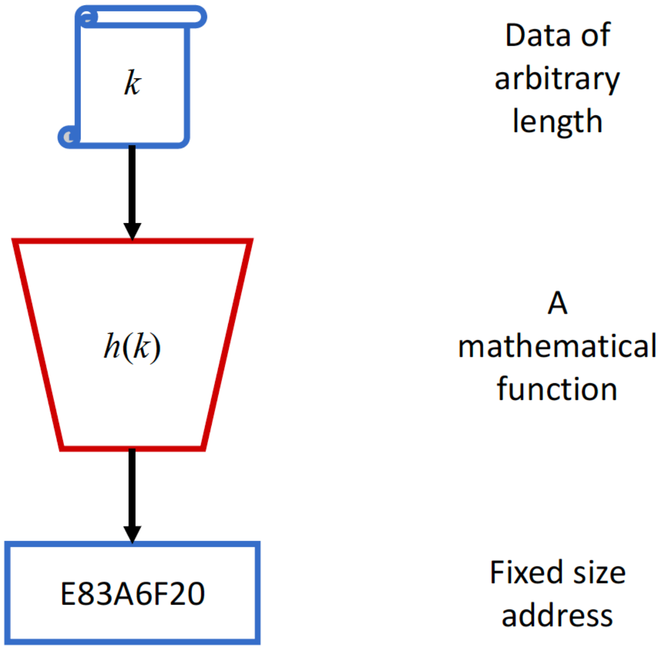
- Binary search tree
  - $O(\log n)$ — if balanced

- Can we do better?
  - Hashing (at the cost of spending a bit of space)
  - $O(1)$



Hashing

- Hashing is a standard way of implementing the abstract data type "dictionary".
- Implemented well, it makes data retrieval very fast.
- A key can be anything, as long as we can map it efficiently to a positive integer. In particular, the set K of keys needs not be bounded.
- Assume we have a table of size m (the hash table).
- The idea is to have a function h: $K \to \{1, \ldots, m\}$ (the hash function) determine where records are stored: A record with key k should be stored in location h(k).
- The address h(k) is the hash address.



The Hash Table

- We can think of the hash table as an abstract data structure supporting operations:
  - find
  - insert
  - lookup (search and insert if not there)
  - initialise
  - delete
  - rehash
- The challenges
  - Design of hash functions.
  - Collision handling.

The Hash Function

- If we have a hash table of size m and keys are integers, we may define

$$h(n) = n \bmod m.$$

- But keys may be other things, such as strings of characters, and the hash function should apply to these and still be easy (cheap) to compute.

- We need to choose m so that it is large enough to allow efficient operations, without taking up excessive memory.

- The hash function should distribute keys evenly along the cells of the hash table.

The Hash Function

- If we have a hash table of size m and keys are integers, we may define

$$h(n) = n \bmod m.$$

- Examples of modulo operation (remainder after division):
  - $5 \bmod 11 = 5$
  - $76999 \bmod 11 = 10$
  - $120 \bmod 11 = 10$

- Example hash function: $h(n) = n \bmod 23$

n	19	392	179	359	262	321	97	468
h(n)								

The Hash Function

- If we have a hash table of size m and keys are integers, we may define

$$h(n) = n \bmod m.$$

- Examples of modulo operation (remainder after division):
  - $5 \bmod 11 = 5$
  - $76999 \bmod 11 = 10$
  - $120 \bmod 11 = 10$

- Example hash function: $h(n) = n \bmod 23$

n	19	392	179	359	262	321	97	468
h(n)	19	1	18	14	9	22	5	8

Hashing of Strings

- For simplicity we assume $\mathrm{A} \longmapsto 0, \mathrm{B} \longmapsto 1, \mathrm{C} \longmapsto 2$ etc.

char	A	B	C	D	E	F	G	H	I	J	K	L	M
s_i	0	1	2	3	4	5	6	7	8	9	10	11	12
char	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
s_i	13	14	15	16	17	18	19	20	21	22	23	24	25

- We use this encoding to find the hash function of a string $s = s_0 s_1 s_2 \cdots$

$$h(s) = \left(\sum_{i=0}^{|s|-1} s_i \right) mod\ m$$

- Example, consider $m = 13$, and the list of strings:

$$[A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED]$$

								SUM	$h(s)$
A									
0								0	0

								SUM	$h(s)$
A									
0								0	0
F	O	O	L						
5	14	14	11					44	5

								SUM	$h(s)$
A									
0								0	0
F	O	O	L						
5	14	14	11					44	5
A	N	D							
0	13	3						16	3

								SUM	$h(s)$
A									
0								0	0
F	O	O	L						
5	14	14	11					44	5
A	N	D							
0	13	3						16	3
H	I	S							
7	8	18						33	7

								SUM	$h(s)$
A									
0								0	0
F	O	O	L						
5	14	14	11					44	5
A	N	D							
0	13	3						16	3
H	I	S							
7	8	18						33	7
M	O	N	E	Y					
12	14	13	4	24				67	2

								SUM	$h(s)$
A									
0								0	0
F	O	O	L						
5	14	14	11					44	5
A	N	D							
0	13	3						16	3
H	I	S							
7	8	18						33	7
M	O	N	E	Y					
12	14	13	4	24				67	2
A	R	E							
0	17	4						21	8

								SUM	$h(s)$
A									
0								0	0
F	O	O	L						
5	14	14	11					44	5
A	N	D							
0	13	3						16	3
H	I	S							
7	8	18						33	7
M	O	N	E	Y					
12	14	13	4	24				67	2
A	R	E							
0	17	4						21	8
S	O	O	N						
18	14	14	13					59	7

THE UNIVERSITY OF MELBOURNE

								SUM	$h(s)$
A									
0								0	0
F	O	O	L						
5	14	14	11					44	5
A	N	D							
0	13	3						16	3
H	I	S							
7	8	18						33	7
M	O	N	E	Y					
12	14	13	4	24				67	2
A	R	E							
0	17	4						21	8
S	O	O	N						
18	14	14	13					59	7
P	A	R	T	E	D				
15	0	17	19	4	3			58	6

								SUM	$h(s)$
A									
0								0	0
F	O	O	L						
5	14	14	11					44	5
A	N	D							
0	13	3						16	3
H	I	S							
7	8	18						33	7
M	O	N	E	Y					
12	14	13	4	24				67	2
A	R	E							
0	17	4						21	8
S	O	O	N						
18	14	14	13					59	7
P	A	R	T	E	D				
15	0	17	19	4	3			58	6

Hashing of Strings

- We assume a binary representation of the 26 characters, with 5 bits per character (0—31)
- Instead of adding, we concatenate the binary strings
- Consider the example key: $M\ Y\ K\ E\ Y$
- Assume a hash table of size $m = 101$.

$char$	s_i	$bin(s_i)$	$char$	s_i	$bin(s_i)$
A	0	00000	N	13	01101
B	1	00001	O	14	01110
C	2	00010	P	15	01111
D	3	00011	Q	16	10000
E	4	00100	R	17	10001
F	5	00101	S	18	10010
G	6	00110	T	19	10011
H	7	00111	U	20	10100
I	8	01000	V	21	10101
J	9	01001	W	22	10110
K	10	01010	X	23	10111
L	11	01011	Y	24	11000
M	12	01100	Z	25	11001

Hashing of Strings

$char$	M	Y	K	E	Y
s_i	12	24	10	4	24
$bin(s_i)$	01100	11000	01010	00100	11000
i	0	1	2	3	4

- Now concatenate the binary string:

$$M\ Y\ K\ E\ Y \longmapsto 0110011000010100010011000\ (=13379736)$$
$$13379736 \bmod 101 = 64$$

- So 64 is the position of string of string $M\ Y\ K\ E\ Y$ in the hash table.

- We deliberately chose m to be prime.

$$13379736 = 12 \times 32^4 + 24 \times 32^3 + 10 \times 32^2 + 4 \times 32^1 + 24 \times 32^0$$

- With m $= 32$, the hash value of any key is the last character's value!

Handling Long Strings as Keys

- More precisely, let chr be the function that gives a character's number (between 0 and 25 under our simple assumptions), so for example $chr(c) = 2$.

- Then we have

$$\text{hash(s)} = \sum_{i=0}^{|s|-1} chr(s_i) \times 32^{|s|-i-1}$$

- For example,

$$\text{hash(V E R Y L O N G K E Y)} = (21 \times 32^{10} + 4 \times 32^{9} + \cdots)\ mod\ 101$$

- The stuff between parentheses quickly becomes ab impossibly large number!
  - DEC: 23804165628760600
  - BIN: 1010100100100011100001100110100011110100001001000011000

Horner's Rule

- Fortunately there is a trick that allows us to avoid large numbers in the hash calculations. Instead of

$$21 \times 32^{10} + 4 \times 32^{9} + 17 \times 32^{8} + 24 \times 32^{7} + \cdots$$

factor out repeatedly:

$$(\cdots((21 \times 32 + 4) \times 32 + 17) \times 32 + \cdots) + 24$$

- Example:

$$\begin{aligned} p(x) &= 2x^4 - x^3 + 3x^2 + x - 5 \\ &= x(2x^3 - x^2 + 3x + 1) - 5 \\ &= x(x(2x^2 - x + 3) + 1) - 5 \\ &= x(x(x(2x - 1) + 3) - 1) - 5 \end{aligned}$$

Horner's Rule

- Fortunately there is a trick that allows us to avoid large numbers in the hash calculations. Instead of

$$21 \times 32^{10} + 4 \times 32^{9} + 17 \times 32^{8} + 24 \times 32^{7} + \cdots$$

factor out repeatedly:

$$\left(\cdots\left(\left(21 \times 32 + 4\right) \times 32 + 17\right) \times 32 + \cdots \right) + 24$$

- Now utilise these properties of modular arithmetic:

$$(x + y) \bmod m = \left((x \bmod m) + (y \bmod m)\right) \bmod m$$
$$(x \times y) \bmod m = \left((x \bmod m) \times (y \bmod m)\right) \bmod m$$

- So for each sub-expression it suffices to take values modulo m.

Horner's Rule

- Fortunately there is a trick that allows us to avoid large numbers in the hash calculations. Instead of

$$21 \times 32^{10} + 4 \times 32^{9} + 17 \times 32^{8} + 24 \times 32^{7} + \cdots$$

factor out repeatedly:

$$(\cdots((21 \times 32 + 4) \times 32 + 17) \times 32 + \cdots) + 24$$

- Step 1: $h(0) = (21 \times 32 + 4)\ mod\ 101$
- Step 2: $h(1) = (h(0) \times 32 + 17)\ mod\ 101$
- Step 3: $h(2) = (h(1) \times 32 + 24)\ mod\ 101$

…

The Hash Function and Collisions

- The hash function should be as random as possible.
- Here we assume $m = 23$ and $\mathrm{h(k)} = k \ mod \ m$.
- In some cases different keys will be mapped to the same hash table address.
- When this happens we have a collision.
- Different hashing methods resolve collisions differently.

Key	Address
19	19
392	1
179	18
359	14
663	19
262	9
639	18
321	22
97	5
468	8
814	9

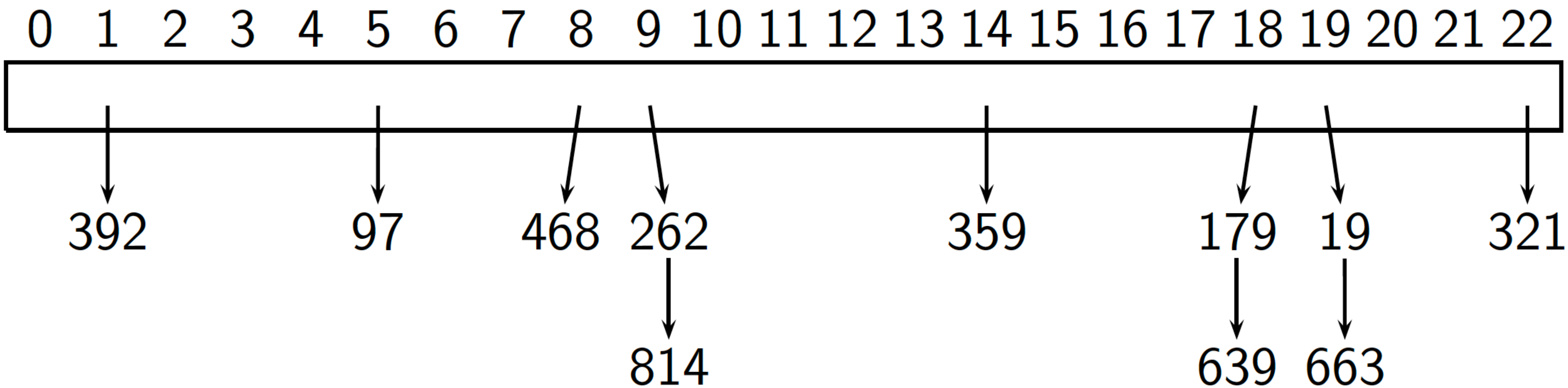
The Hash Function and Collisions

- The hash function should be as random as possible.
- Here we assume $m = 23$ and $\mathrm{h(k)} = k \bmod m$.
- In some cases different keys will be mapped to the same hash table address.
- When this happens we have a collision.
- Different hashing methods resolve collisions differently.

Key	Address
19	19
392	1
179	18
359	14
663	19
262	9
639	18
321	22
97	5
468	8
814	9

Separate Chaining

- Element k of the hash table is a list of keys with the hash value k.



- This gives easy collision handling.
- The load factor $\alpha = n/m$, where n is the number of items stored.
- Number of probes in successful search $\approx 1 + \alpha/2$.
- Number of probes in unsuccessful search $\approx \alpha$.

Separate Chaining Pros and Cons

- Compared with sequential search, reduces the number of comparisons by a factor of m.
- Good in a dynamic environment, when (number of) keys are hard to predict.
- The chains can be ordered, or records may be "pulled up front" when accessed.
- Deletion is easy.
- However, separate chaining uses extra storage for links.

Open-Addressing Methods

THE UNIVERSITY OF MELBOURNE

- With open-addressing methods (also called closed hashing) all records are stored in the hash table itself (not linked lists hanging off the table).
- There are many methods of this type. We only discuss two:
  - linear probing
  - double hashing
- For these methods, the load factor $\alpha \leq 1$.

Linear Probing

- In case of collision, try the next cell, then the next, and so on.

- After the arrival of 19 (19), 392 (1), 179 (18), 663 (19), 639 (18), 321 (22):

0	1	2	3	…	16	17	18	19	20	21	22
	392			…			179	19	663	639	321

- Search proceeds in a similar fashion.

- If we get to the end of the hash table, we wrap around.

- For example, if key 20 now arrives it will be placed in cell 0.

Linear Probing

- Again let m be the table size, and n be the number of records stored.
- As before, $\alpha = n/m$ is the load factor.
- Average number of probes:
  - Successful search: $\frac{1}{2} + \frac{1}{2(1-\alpha)}$
  - Unsuccessful: $\frac{1}{2} + \frac{1}{2(1-\alpha)^2}$

For successful search:

α	#probes
0.1	1.06
0.25	1.17
0.5	1.50
0.75	2.50
0.9	5.50
0.95	10.50

Linear Probing Pros and Cons

- Space-efficient.
- Worst-case performance miserable; must be careful not to let the load factor grow beyond 0.9.
- Comparative behaviour, m=11113, $n = 10000$, α=0.9:
  - Linear probing: 5.5 probes in average (success)
  - Binary search: 12.3 probes on average (success)
  - Linear search: 5000 probes on average (success)
- Clustering is a major problem: the collision handling strategy leads to clusters of contiguous cells being occupied.
- Deletion is almost impossible.

Double Hashing

- To alleviate the clustering problem in linear probing, there are better ways of resolving collisions.
- One is double hashing which uses a second hash function s to determine an offset to be used in probing for a free cell.
- For example, we may choose $s(k) = 1 + k \; mod \; 97$.
- By this we mean, if $h(k)$ is occupied, next try $h(k) + s(k)$, then $h(k) + 2\, s(k)$, and so on.
- This is another reason why it is good to have m being a prime number. That way, using $h(k)$ as the offset, we will eventually find a free cell if there is one.

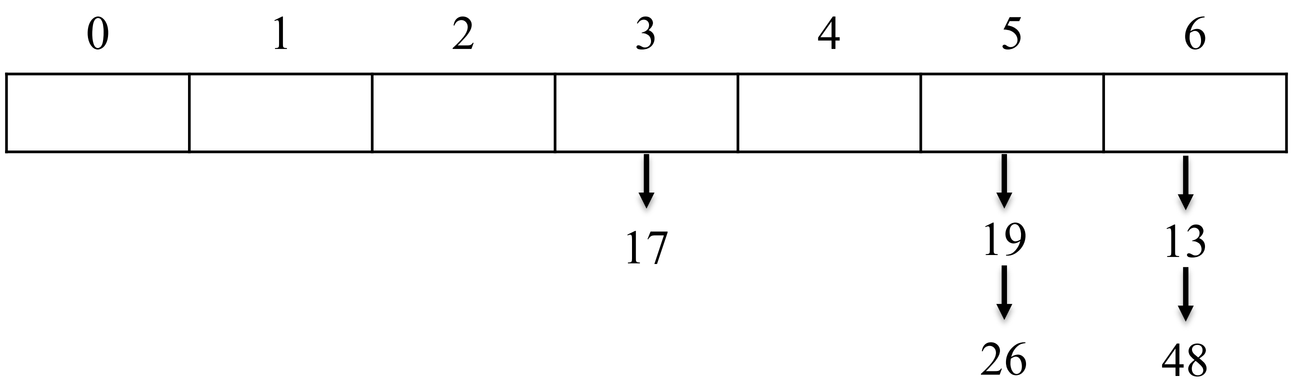
Collision Handling Example

- Consider $h(k) = k \ mod \ 7$. Draw the resulting hash tables after inserting $19, 26,13,48,17$ (in this order).

- Separate chaining

Collision Handling Example

- Consider $h(k) = k \ mod \ 7$. Draw the resulting hash tables after inserting $19, 26, 13, 48, 17$ (in this order).
- Separate chaining



Collision Handling Example

- Consider $h(k) = k \ mod \ 7$. Draw the resulting hash tables after inserting $19, 26{,}13{,}48{,}17$ (in this order).
- Separate chaining

0	1	2	3	4	5	6
			↓ 17		↓ 19	↓ 13
					↓ 26	↓ 48

- Linear probing

0	1	2	3	4	5	6
13	48		17		19	26

Collision Handling Example

- Consider $h(k) = k \ mod \ 7$. Draw the resulting hash tables after inserting $19, 26{,}13{,}48{,}17$ (in this order).

- Separate chaining

0	1	2	3	4	5	6
			↓ 17		↓ 19 ↓ 26	↓ 13 ↓ 48

- Linear probing

0	1	2	3	4	5	6
13	48		17		19	26

- Double hashing, using $s(k) = 5 - (k \ mod \ 5)$ offset

0	1	2	3	4	5	6
	48	26	17		19	13

Rehashing

- The standard approach to avoiding performance deterioration in hashing is to keep track of the load factor and to rehash when is reaches, say, 0.9.

- Rehashing means allocating a larger hash table (typically twice the current size), revisiting each item, calculating its hash address in the new table, and inserting it.

- This "stop-the-world" operation will introduce long delays at unpredictable times, but it will happen relatively infrequently.

Rabin-Karp String Search

- The Rabin-Karp string search algorithm is based on string hashing.

- To search for a string p (of length m) in a larger string s, we can calculate hash(p) and then check every substring $s_i \cdots s_{i+m-1}$ to see if it has the same hash value. Of course, if it has, the strings may still be different; so we need to compare them in the usual way.

- If $p = s_i \cdots s_{i+m-1}$ then the hash values are the same; otherwise the values are almost certainly going to be different.

- Since false positives will be so rare, the $O(m)$ time it takes to actually compare the strings can be ignored.

Rabin-Karp String Search

Search String `c a t s` **Hash = 0x51537230**

Input `t h e c a t s a t o n ...`

Window

`t h e c` Hash = 0x12415273

`h e c a` Hash = 0x41246364

`e c a t` Hash = 0x64523623

`c a t s` **Hash = 0x51537230**

Rabin-Karp String Search

Search String | c | a | t | s | **Hash = 0x51537230**

Input | t | h | e | c | a | t | s | a | t | o | n | ...

Window

t | h | e | c | Hash = 0x12415273

h | e | c | a | Hash = 0x41246364

e | c | a | t | Hash = 0x64523623

c | a | t | s | **Hash = 0x51537230**

Length of pattern = M;

Hash(p) = hash value of pattern;

Hash(t) = hash value of first M letters in body of text;

do

 if (hash(p) == hash(t))

  brute force comparison of pattern and selected section of text

  hash(t) = hash value of next section of text, one character over

while (end of text **or** brute force comparison == true)

Rabin-Karp String Search

- Example: consider the text 31415926535, find the pattern 26 by using the hash function $h(k) = k \ mod \ 11$. The hash for the pattern is: $h(26) = 26 \ mod \ 11 = 4$.

3	1	4	1	5	9	2	6	5	3	5	$h(k)$
3	1										9

Rabin-Karp String Search

- Example: consider the text 31415926535, find the pattern 26 by using the hash function $h(k) = k \; mod \; 11$. The hash for the pattern is: $h(26) = 26 \; mod \; 11 = 4$.

3	1	4	1	5	9	2	6	5	3	5	$h(k)$
3	1										9
	1	4									3

Rabin-Karp String Search

- Example: consider the text 31415926535, find the pattern 26 by using the hash function $h(k) = k \ mod \ 11$. The hash for the pattern is: $h(26) = 26 \ mod \ 11 = 4$.

3	1	4	1	5	9	2	6	5	3	5	$h(k)$
3	1										9
	1	4									3
		4	1								8

Rabin-Karp String Search

- Example: consider the text 31415926535, find the pattern 26 by using the hash function $h(k) = k \ mod \ 11$. The hash for the pattern is: $h(26) = 26 \ mod \ 11 = 4$.

3	1	4	1	5	9	2	6	5	3	5	$h(k)$
3	1										9
	1	4									3
		4	1								8
			1	5							4

Rabin-Karp String Search

- Example: consider the text 31415926535, find the pattern 26 by using the hash function $h(k) = k \; mod \; 11$. The hash for the pattern is: $h(26) = 26 \; mod \; 11 = 4$.

3	1	4	1	5	9	2	6	5	3	5	$h(k)$
3	1										9
	1	4									3
		4	1								8
			1	5							4

Rabin-Karp String Search

- Example: consider the text 31415926535, find the pattern 26 by using the hash function $h(k) = k \; mod \; 11$. The hash for the pattern is: $h(26) = 26 \; mod \; 11 = 4$.

3	1	4	1	5	9	2	6	5	3	5	$h(k)$
3	1										9
	1	4									3
		4	1								8
			1	5							4
				5	9						4

Rabin-Karp String Search

- Example: consider the text 31415926535, find the pattern 26 by using the hash function $h(k) = k\ mod\ 11$. The hash for the pattern is: $h(26) = 26\ mod\ 11 = 4$.

3	1	4	1	5	9	2	6	5	3	5	$h(k)$
3	1										9
	1	4									3
		4	1								8
			1	5							4
				5	9						4

Rabin-Karp String Search

- Example: consider the text 31415926535, find the pattern 26 by using the hash function $h(k) = k\ mod\ 11$. The hash for the pattern is: $h(26) = 26\ mod\ 11 = 4$.

3	1	4	1	5	9	2	6	5	3	5	$h(k)$
3	1										9
	1	4									3
		4	1								8
			1	5							4
				5	9						4
					9	2					4

Rabin-Karp String Search

- Example: consider the text 31415926535, find the pattern 26 by using the hash function $h(k) = k \; mod \; 11$. The hash for the pattern is: $h(26) = 26 \; mod \; 11 = 4$.

3	1	4	1	5	9	2	6	5	3	5	$h(k)$
3	1										9
	1	4									3
		4	1								8
			1	5							4
				5	9						4
					9	2					4

Rabin-Karp String Search

- Example: consider the text 31415926535, find the pattern 26 by using the hash function $h(k) = k\ mod\ 11$. The hash for the pattern is: $h(26) = 26\ mod\ 11 = 4$.

3	1	4	1	5	9	2	6	5	3	5	$h(k)$
3	1										9
	1	4									3
		4	1								8
			1	5							4
				5	9						4
					9	2					4
						2	6				4

Rabin-Karp String Search

- Repeatedly hashing strings of length m seems like a bad idea. However, the hash values can be calculated incrementally. The hash value of the length-m substring s that starts at position j is:

$$hash(s, j) = \sum_{i=0}^{m-1} chr(s_{j+i}) \times a^{m-i-1},$$

where a is the alphabet size. From that we we can get the next hash value, for the substring that starts at position $j + 1$, quite cheaply:

$$hash(s, j + 1) = \left(hash(s, j) - a^{m-1} chr(s_j)\right) \times a + chr(s_{j+m})$$

modulo m. Effectively we just subtract the contributions of s_j and add the contributions of s_{j+m}, for the cost of two multiplications, one addition and one subtraction.

Rabin-Karp String Search

- Example: has all 3-substrings of "there".

- The first substring "the" $= t \cdot (26)^2 + h \cdot (26) + e$

- If we have "the", can we compute "her"?

$$
\begin{aligned}
\text{“}her\text{”} &= h \cdot (26)^2 + e \cdot (26) + r \\
&= 26 \cdot (h \cdot (26) + e) + r \\
&= 26 \cdot (t.(26)^2 + h \cdot (26) + e - t.(26)^2) + r \\
&= 26 \cdot (\text{“}the\text{”} - t.(26)^2) + r
\end{aligned}
$$

Rabin-Karp String Search

- Example: has all 3-substrings of "there".

- The first substring "the" $= t \cdot (26)^2 + h \cdot (26) + e$

- If we have "the", can we compute "her"?

$$
\begin{aligned}
\text{"}her\text{"} &= h \cdot (26)^2 + e \cdot (26) + r \\
&= 26 \cdot (h \cdot (26) + e) + r \\
&= 26 \cdot (t.(26)^2 + h \cdot (26) + e - t.(26)^2) + r \\
&= 26 \cdot (\text{"}the\text{"} - t.(26)^2) + r
\end{aligned}
$$

- i.e. subtract the first letter's contribution to the number, shift, and add the last letter.

Why Not Always Use Hashing?

- Some drawbacks:
  - If an an application call for traversal of all items in sorted order, a hash table is no good.
  - Also, unless we use separate chaining, deletion is virtually impossible.
  - It may be hard to predict the volume of data, and rehashing is an expensive “stop-the-world” operation.

When to Use Hashing?

- All sorts of information retrieval applications involving thousands to millions of keys.
- Typical example: symbol tables used by compilers. The compiler hashes all (variable, function, etc.) names and stores information related to each – no deletion in this case.
- When hashing is applicable, it is usually superior; a well-tuned hash table will outperform its competitors.
- Unless you let the load factor get too high, or you botch up the hash function. IT is a good idea to print statistics to check that the function really does spread keys uniformly across the hash table.

Coming Up Next

- Dynamic programming and optimisation.