COMP90038
Algorithms and Complexity

Lecture 17: Hashing

(with thanks to Harald Søndergaard & Michael Kirley)

Andres Munoz-Acosta

munoz.m@unimelb.edu.au

Peter Hall Building G.83

On the previous lecture

- We talked about **input enhancement**, a technique that **pre-processes** the input, and stores **additional information** obtained in the form of a **table**

- We discussed **three algorithms** that use **input enhancement**
  - Distribution counting
  - Horspool's string matching algorithm
  - Knuth-Morris-Prat string matching algorithm

The failure table for the pattern P = GCAGGT with alphabet CGTA is

[0 1 2 0 1 3]

[1 1 0 2 2 1]

[0 1 1 0 3 3]

[0 1 1 0 2 2]

Today's lecture

- **Hashing** is a standard approach to implement a **dictionary**, an ADT on which each record is composed of <attribute name, value> pairs, e.g., a student record:
  - Attributes: Student ID, Name, data of birth, address, major, etc…

Today's lecture

- **Hashing** is a standard approach to implement a **dictionary**, an ADT on which each record is composed of <attribute name, value> pairs, e.g., a student record:
  - Attributes: Student ID, Name, data of birth, address, major, etc…

- Each record is identified by an arbitrary **key** of any type: integer, char, even string. However:
  - The key must map efficiently to a positive integer
  - The set K of keys need not be bounded

Today's lecture

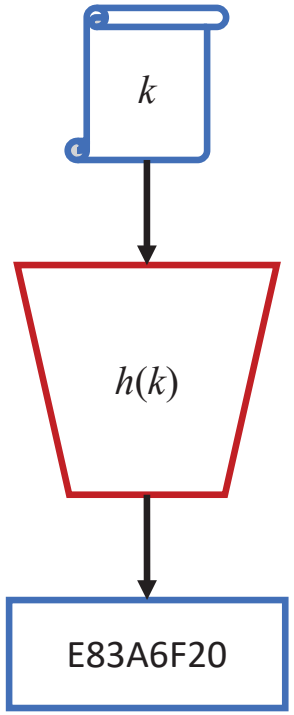
- **Hashing** is a standard approach to implement a **dictionary**, an ADT on which each record is composed of <attribute name, value> pairs, e.g., a student record:
  - Attributes: Student ID, Name, data of birth, address, major, etc…

- Each record is identified by an arbitrary **key** of any type: integer, char, even string. However:
  - The key must map efficiently to a positive integer
  - The set K of keys need not be bounded

- Records are stored in a **hash table** of size m, which should be:
  - Large enough for efficient operation
  - Small enough to limit memory usage

Hashing

- A **hash function** $h(k)$ determines an **index** in the hash table
  - A record with key k **should be stored** in location $h(k)$



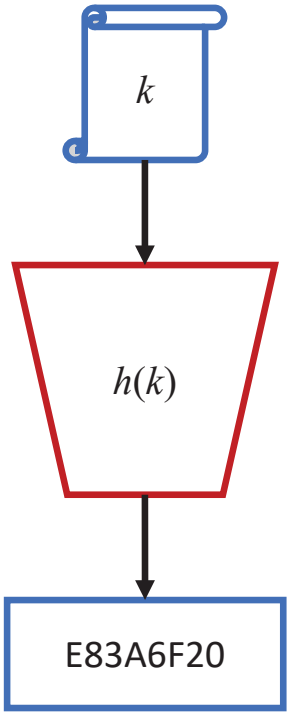
Data of arbitrary length

A mathematical function

Fixed size address

Hashing

- A **hash function** $h(k)$ determines an **index** in the hash table
  - A record with key k **should be stored** in location $h(k)$
- The **hash address** is the value of $h(k)$
  - **Different keys** could have the **same address**. This is known as a **collision**



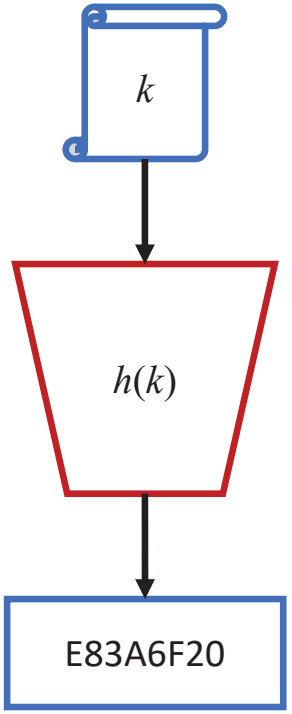
Data of arbitrary length

A mathematical function

Fixed size address

Hashing

- A **hash function** $h(k)$ determines an **index** in the hash table
  - A record with key k **should be stored** in location $h(k)$
- The **hash address** is the value of $h(k)$
  - **Different keys** could have the **same address**. This is known as a **collision**
- The challenges in implementing a **hash table** are:
  - Design a **robust** hash function
  - Handling of collisions



Data of arbitrary length

A mathematical function

Fixed size address

Hash functions

- The **ideal function** is **cheap** to compute and distributes the keys **evenly** across the table

Hash functions

- The **ideal function** is **cheap** to compute and distributes the keys **evenly** across the table

- For example, if the keys are **integers**, a simple function could be $h(n) = n \bmod m$. If $m = 23$:

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

Hash functions

- Given the encoding below, a hash table of size 13 and the function:

$$h(s) = \left(\sum_{i=0}^{|s|-1} \mathrm{a}_i \right) \mod 13$$

Calculate the address for the following list of keys:

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

char	A	B	C	D	E	F	G	H	I	J	K	L	M
a	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
a	13	14	15	16	17	18	19	20	21	22	23	24	25

Hash functions

	SUM	$h(s)$
A		
0	0	0

Hash functions

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

Hash functions

					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

Hash functions

					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

Hash functions

						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

Hash functions

						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

Hash functions

					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

Hash functions

						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

Hash functions

						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	← REPEAT!
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	← REPEAT!
P	A	R	T	E	D			
15	0	17	19	4	3	58	6	

Hash functions

- Assume a binary representation of the 26 characters, with **5 bits/character** (0 to 31)

char	a	bin(a)	char	a	bin(a)
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

Hash functions

- Assume a binary representation of the 26 characters, with **5 bits/character** (0 to 31)

- Instead of adding, we **concatenate** the binary strings
  - By concatenating them, we are basically multiplying by 32

char	a	bin(a)	char	a	bin(a)
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

Hash functions

- Assume a binary representation of the 26 characters, with **5 bits/character** (0 to 31)
- Instead of adding, we **concatenate** the binary strings
  - By concatenating them, we are basically multiplying by 32
- Our hash table is of size 101 (***m* is prime**)

char	a	bin(a)	char	a	bin(a)
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

Hash functions

- Assume a binary representation of the 26 characters, with **5 bits/character** (0 to 31)
- Instead of adding, we **concatenate** the binary strings
  - By concatenating them, we are basically multiplying by 32
- Our hash table is of size 101 (***m* is prime**)
- Our key is ‘MYKEY’

char	a	bin(a)	char	a	bin(a)
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

Hash functions

							k	$k \mod 101$
Char	M	Y	K	E	Y			
a	12	24	10	4	24			
$\text{BIN}(a)$	01100	11000	01010	00100	11000			
i	4	3	2	1	0			
32^i	1048576	32768	1024	32	1			
$a \times \left(32^i\right)$	12582912	786432	10240	128	24	13379736		64

Hash functions

						k	$k \bmod 101$
Char	M	Y	K	E	Y		
a	12	24	10	4	24	← DECIMAL REPRESENTATION	
$\text{BIN}(a)$	01100	11000	01010	00100	11000		
i	4	3	2	1	0		
32^i	1048576	32768	1024	32	1		
$a \times \left(32^i\right)$	12582912	786432	10240	128	24	13379736	64

Hash functions

						k	$k \bmod 101$
Char	M	Y	K	E	Y		
a	12	24	10	4	24		
$\text{BIN}(a)$	01100	11000	01010	00100	11000	← BINARY REPRESENTATION	
i	4	3	2	1	0		
32^i	1048576	32768	1024	32	1		
$a \times \left(32^i\right)$	12582912	786432	10240	128	24	13379736	64

Hash functions

						k	$k \bmod 101$
Char	M	Y	K	E	Y		
a	12	24	10	4	24		
$\text{BIN}(a)$	01100	11000	01010	00100	11000		
i	4	3	2	1	0	← INDEX OF THE CHARACTER	
32^i	1048576	32768	1024	32	1		
$a \times \left(32^i\right)$	12582912	786432	10240	128	24	13379736	64

Hash functions

						k	$k \bmod 101$
Char	M	Y	K	E	Y		
a	12	24	10	4	24		
$\text{BIN}(a)$	01100	11000	01010	00100	11000		
i	4	3	2	1	0		
32^i	1048576	32768	1024	32	1	← VALUE AFTER CONCATENATION	
$a \times \left(32^i\right)$	12582912	786432	10240	128	24	13379736	64

Hash functions

						k	$k \bmod 101$
Char	M	Y	K	E	Y		
a	12	24	10	4	24		
$\text{BIN}(a)$	01100	11000	01010	00100	11000		
i	4	3	2	1	0		
32^i	1048576	32768	1024	32	1		
$a \times \left(32^i\right)$	12582912	786432	10240	128	24	13379736	64

A HUGE VALUE

Hash functions

						k	$k \bmod 101$
Char	M	Y	K	E	Y		
a	12	24	10	4	24		
$\text{BIN}(a)$	01100	11000	01010	00100	11000		
i	4	3	2	1	0		
32^i	1048576	32768	1024	32	1		
$a \times \left(32^i\right)$	12582912	786432	10240	128	24	13379736	64

- Note that the hash function is a polynomial:

$$h(s) = a_{|s|-1}32^{|s|-1} + a_{|s|-2}32^{|s|-2} + \cdots + a_1 32 + a_0$$

Hash functions

- What would happen if our key is the longer string ‘VERYLONGKEY’

$$h(VERYLONGKEY) = \left(21 \times 32^{10} + 4 \times 32^{9} + \cdots + 4 \times 32 + 24\right) \mod 101$$

Hash functions

- What would happen if our key is the longer string ‘VERYLONGKEY’

$$h(VERYLONGKEY) = \left(21 \times 32^{10} + 4 \times 32^{9} + \cdots + 4 \times 32 + 24\right) \mod 101$$

- The stuff between parentheses becomes a very large number:
  - DEC: 2380416562876060O
  - BIN: 10101001001000111000011001101000111101000010010000110000

Hash functions

- What would happen if our key is the longer string ‘VERYLONGKEY’

$$h(VERYLONGKEY) = \left(21 \times 32^{10} + 4 \times 32^{9} + \cdots + 4 \times 32 + 24\right) \mod 101$$

- The stuff between parentheses becomes a very large number:
  - DEC: 2380416562876060O
  - BIN: 10101001001000111000011001101000111101000010010000110000
- Calculating this polynomial by brute force is very expensive

Horner’s rule

- **Horner's rule** is a trick to simplify polynomial calculation

$$p(x) = a_3 \times x^3 + a_2 \times x^2 + a_1 \times x + a_0$$

Horner’s rule

- **Horner's rule** is a trick to simplify polynomial calculation

$$p(x) = a_3 \times x^3 + a_2 \times x^2 + a_1 \times x + a_0$$

- Factorizing x we have:

$$p(x) = (((a_3 \times x) + a_2) \times x + a_1) \times x + a_0$$

Horner’s rule

- **Horner's rule** is a trick to simplify polynomial calculation

$$p(x) = a_3 \times x^3 + a_2 \times x^2 + a_1 \times x + a_0$$

- Factorizing x we have:

$$p(x) = (((a_3 \times x) + a_2) \times x + a_1) \times x + a_0$$

- Applying the modulus we have:

$$p(x) = ((((a_3 \times x) + a_2) \times x + a_1) \times x + a_0) \mod m$$

Horner's rule

- We can use the following properties of modular arithmetic:

$$x \boxplus y = (x + y) \mod m = ((x \mod m) + (y \mod m)) \mod m$$

$$x \boxtimes y = (x \times y) \mod m = ((x \mod m) \times (y \mod m)) \mod m$$

Horner's rule

- We can use the following properties of modular arithmetic:

$$x \boxplus y = (x + y) \mod m = ((x \mod m) + (y \mod m)) \mod m$$

$$x \boxtimes y = (x \times y) \mod m = ((x \mod m) \times (y \mod m)) \mod m$$

- The modulus **distributes across all operations**:

$$p(x) = (((((a_3 \boxtimes x) \boxplus a_2) \boxtimes x) \boxplus a_1) \boxtimes x) \boxplus a_0$$

- The result of each **operation does not exceed** m

Handling collisions

- The hash function should be as **random** as possible

- In some cases **different keys** are mapped to the **same address**. For example $h(n) = n \bmod 23$

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

Handling collisions

- The hash function should be as **random** as possible

- In some cases **different keys** are mapped to the **same address**. For example $h(n) = n \bmod 23$

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

Handling collisions

- The hash function should be as **random** as possible

- In some cases **different keys** are mapped to the **same address**. For example $h(n) = n \bmod 23$

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

Handling collisions

- The hash function should be as **random** as possible

- In some cases **different keys** are mapped to the **same address**. For example $h(n) = n \bmod 23$

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

Handling collisions

- The hash function should be as **random** as possible
- In some cases **different keys** are mapped to the **same address**. For example $h(n) = n \bmod 23$

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

- This is called a **collision**
  - There are different methods to resolve collisions

Separate chaining

- Each element k of the hash table is a **linked list**, which makes collision handling very easy

- **Exercise**: For $h(n) = n \bmod 23$ add to this table [83 110 14]

ADDRESS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
LIST		392				97			468	262					359				179	19			
										814									639	663			

Separate chaining

- Each element k of the hash table is a **linked list**, which makes collision handling very easy

- **Exercise**: For $h(n) = n \bmod 23$ add to this table [83 110 14]

ADDRESS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
LIST		392				97			468	262					359				179	19			
										814					83				639	663			
															14				110				

Separate chaining

- Each element k of the hash table is a **linked list**, which makes collision handling very easy

- **Exercise**: For $h(n) = n \bmod 23$ add to this table [83 110 14]

ADDRESS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
LIST		392				97			468	262					359				179	19			
										814					83				639	663			
															14				110				

- The **load factor** $\alpha = n/m$, where n is the number of items to be stored, indicates how "full" the table is
  - Number of probes in **successful** search $\sim 1 + (\alpha/2)$
  - Number of probes in **unsuccessful** search $\sim \alpha$

Separate chaining

- Compared with **sequential search**, separate chaining reduces the number of comparisons by a factor of m
- Good in a dynamic environment, when the number of keys is hard to predict
- The chains can be ordered, or records may be “pulled up front” when accessed
- Deletion is easy
- Separate chaining uses additional storage for the links

Open-addressing methods

- **Open-addressing** methods (also called **closed hashing**) store all records in the hash table itself (not in linked lists hanging off the table)

- There are many methods of this type. We focus on two:
  - **linear probing**
  - **double hashing**

- For these methods, the load factor $\alpha \leq 1$
  - The table must be larger than the number of elements

Linear probing

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

- Assume the following data (and its **keys**) arriving one at the time:

[19(**19**) 392(**1**) 179(**18**) 663(**19**→**20**) 639(**18**→**21**) 321(**22**) …]

Linear probing

- In case of collision, try the next cell, then the next, and so on.
- Assume the following data (and its **keys**) arriving one at the time:

[19(**19**) 392(**1**) 179(**18**) 663(**19**→**20**) 639(**18**→**21**) 321(**22**) …]

- Search proceeds in similar fashion

Linear probing

- In case of collision, try the next cell, then the next, and so on.
- Assume the following data (and its **keys**) arriving one at the time:

[19(**19**) 392(**1**) 179(**18**) 663(**19**→**20**) 639(**18**→**21**) 321(**22**) …]

- Search proceeds in similar fashion
- If we get to the end of the table, we wrap around, e.g., if key 20 arrives, it will be placed in cell 0.

Linear probing

- **Exercise**: Add [83(**14**) 110(**18**) 497(**14**)] to the table

ADDRESS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
LIST		392				97			468	262	814				359				179	19	663	639	321

Linear probing

- **Exercise**: Add [83(**14**) 110(**18**) 497(**14**)] to the table

ADDRESS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
LIST	110	392				97			468	262	814				359	83	497		179	19	663	639	321

Linear probing

- **Exercise**: Add [83(**14**) 110(**18**) 497(**14**)] to the table

ADDRESS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
LIST	110	392				97			468	262	814				359	83	497		179	19	663	639	321

- As before, $\alpha = n/m$ is the load factor. The average number of probes:
  - Successful search: $0.5 + 1/(2(1- \alpha))$
  - Unsuccessful: $0.5 + 1/(2(1- \alpha)^2)$

Linear probing

- **Exercise**: Add [83(**14**) 110(**18**) 497(**14**)] to the table

ADDRESS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
LIST	110	392				97			468	262	814				359	83	497		179	19	663	639	321

- As before, $\alpha = n/m$ is the load factor. The average number of probes:
  - Successful search: $0.5 + 1/(2(1- \alpha))$
  - Unsuccessful: $0.5 + 1/(2(1- \alpha)^2)$

- Linear probing is space-efficient

Linear probing

- **Exercise**: Add [83(**14**) 110(**18**) 497(**14**)] to the table

ADDRESS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
LIST	110	392				97			468	262	814				359	83	497		179	19	663	639	321

- As before, $\alpha = n/m$ is the load factor. The average number of probes:
  - Successful search: $0.5 + 1/(2(1- \alpha))$
  - Unsuccessful: $0.5 + 1/(2(1- \alpha)^2)$
- Linear probing is space-efficient
- Worst-case performance miserable; load factor must stay around 0.9

Linear probing

- Compare the performance of a table with $m = 11113$, $n = 10000$, $\alpha = 0.9$:
  - Linear probing: 5.5 probes on average (success)
  - Binary search: 12.3 probes on average (success)
  - Linear search: 5000 probes on average (success)

Linear probing

- Compare the performance of a table with $m = 11113$, $n = 10000$, $\alpha = 0.9$:
  - Linear probing: 5.5 probes on average (success)
  - Binary search: 12.3 probes on average (success)
  - Linear search: 5000 probes on average (success)

- Linear probing leads to large groups of contiguous cells being occupied (**clustering**).

Linear probing

- Compare the performance of a table with $m = 11113$, $n = 10000$, $\alpha = 0.9$:
  - Linear probing: 5.5 probes on average (success)
  - Binary search: 12.3 probes on average (success)
  - Linear search: 5000 probes on average (success)

- Linear probing leads to large groups of contiguous cells being occupied (**clustering**).

- Deletion is almost impossible.
  - You could **rehash** everything (reallocate the data to a larger table by revisiting each item, calculating its address, and inserting it) or put a NULL as placeholder

Linear probing

- Compare the performance of a table with $m = 11113$, $n = 10000$, $\alpha = 0.9$:
  - Linear probing: 5.5 probes on average (success)
  - Binary search: 12.3 probes on average (success)
  - Linear search: 5000 probes on average (success)

- Linear probing leads to large groups of contiguous cells being occupied (**clustering**).

- Deletion is almost impossible.
  - You could **rehash** everything (reallocate the data to a larger table by revisiting each item, calculating its address, and inserting it) or put a NULL as placeholder
  - Rehashing is a **"stop-the-world"** operation that introduces long delays at unpredictable times, but happens relatively infrequently

Double hashing

- **Double hashing** is a way to resolve the clustering problem
  - It uses a second hash function s to determine an **offset** used in probing for a free cell
  - If $h(k)$ is occupied, next try $h(k) + s(k)$, then $h(k) + 2s(k)$, and so on.

$k \leftarrow [35,\ 20,\ 26,\ 62,\ 48]$
$n \leftarrow 5$
int $h[n] \leftarrow$ NULL
$h\,[0] \leftarrow k\,[0] \mod 11$
for $i \leftarrow 1$ **to** $n - 1$ **do**
    $h\,[i] \leftarrow k\,[i] \mod 11$
    while IsCollision$(h\,[0, \ldots, i-1]\,, h\,[i]) ==$ TRUE **do**
        $h\,[i] \leftarrow (h\,[i] + (3 + k\,[i] \mod 7)) \mod 11$

Double hashing

- **Double hashing** is a way to resolve the clustering problem
  - It uses a second hash function s to determine an **offset** used in probing for a free cell
  - If $h(k)$ is occupied, next try $h(k) + s(k)$, then $h(k) + 2s(k)$, and so on.

$k \leftarrow [35,\ 20,\ 26,\ 62,\ 48]$
$n \leftarrow 5$
int $h[n] \leftarrow$ NULL
$h[0] \leftarrow k[0] \mod 11$
for $i \leftarrow 1$ **to** $n - 1$ **do**
    $h[i] \leftarrow k[i] \mod 11$
    while IsCollision$(h[0, \ldots, i-1], h[i]) ==$ TRUE **do**
        $h[i] \leftarrow (h[i] + (3 + k[i] \mod 7)) \mod 11$

THE ADDRESS MUST ALSO BE **mod 11**, OTHERWISE IT WON'T FIT IN THE TABLE

Double hashing

- **Double hashing** is a way to resolve the clustering problem
  - It uses a second hash function s to determine an **offset** used in probing for a free cell
  - If $h(k)$ is occupied, next try $h(k) + s(k)$, then $h(k) + 2s(k)$, and so on.

$k \leftarrow [35,\ 20,\ 26,\ 62,\ 48]$
$n \leftarrow 5$
int $h[n] \leftarrow$ NULL
$h\,[0] \leftarrow k\,[0] \mod 11$
for $i \leftarrow 1$ **to** $n - 1$ **do**
    $h\,[i] \leftarrow k\,[i] \mod 11$
    while IsCollision$(h\,[0, \ldots, i-1]\,, h\,[i]) ==$ TRUE **do**
        $h\,[i] \leftarrow (h\,[i] + (3 + k\,[i] \mod 7)) \mod 11$

s(k) ACCUMULATES AT EACH ITERATION OF THE WHILE LOOP

Double hashing

- **Double hashing** is a way to resolve the clustering problem
  - It uses a second hash function s to determine an **offset** used in probing for a free cell
  - If $h(k)$ is occupied, next try $h(k) + s(k)$, then $h(k) + 2s(k)$, and so on.

$k \leftarrow [35,\ 20,\ 26,\ 62,\ 48]$
$n \leftarrow 5$
int $h[n] \leftarrow$ NULL
$h\,[0] \leftarrow k\,[0] \mod 11$
for $i \leftarrow 1$ **to** $n - 1$ **do**
    $h\,[i] \leftarrow k\,[i] \mod 11$
    while IsCollision$(h\,[0, \ldots, i-1]\,, h\,[i]) ==$ TRUE **do**
        $h\,[i] \leftarrow (h\,[i] + (3 + k\,[i] \mod 7)) \mod 11$

- The result after this algorithm is $h = [2\ 9\ 4\ 7\ 0]$

Double hashing

- **Double hashing** is a way to resolve the clustering problem
  - It uses a second hash function s to determine an **offset** used in probing for a free cell
  - If $h(k)$ is occupied, next try $h(k) + s(k)$, then $h(k) + 2s(k)$, and so on.

$k \leftarrow [35,\ 20,\ 26,\ 62,\ 48]$
$n \leftarrow 5$
int $h[n] \leftarrow$ NULL
$h\,[0] \leftarrow k\,[0] \mod 11$
for $i \leftarrow 1$ **to** $n-1$ **do**
    $h\,[i] \leftarrow k\,[i] \mod 11$
    while IsCollision$(h\,[0,\ldots,i-1]\,, h\,[i]) ==$ TRUE **do**
        $h\,[i] \leftarrow (h\,[i] + (3 + k\,[i] \mod 7)) \mod 11$

- The result after this algorithm is $h = [2\ 9\ 4\ 7\ 0]$

- If m is a **prime number**, we will eventually find a free cell if there is one

Rabin-Karp algorithm

- The Rabin-Karp algorithm uses hashing to match strings
- To search for a string $P[1...m]$ in $T[1...n]$, we calculate $h(P)$ and then compare every substring $T[i..i+m\text{-}1]$ hash value

Rabin-Karp algorithm

- The Rabin-Karp algorithm uses hashing to match strings

- To search for a string $P[1...m]$ in $T[1...n]$, we calculate $h(P)$ and then compare every substring $T[i..i+m\text{-}1]$ hash value
  - If the hash values are the same, it is **possible** that $P[1...m] = T[i...i+m\text{-}1]$, so we compare the substring the usual way

Rabin-Karp algorithm

- The Rabin-Karp algorithm uses hashing to match strings

- To search for a string $P[1...m]$ in $T[1...n]$, we calculate $h(P)$ and then compare every substring $T[i..i+m\text{-}1]$ hash value
  - If the hash values are the same, it is **possible** that $P[1...m] = T[i...i+m\text{-}1]$, so we compare the substring the usual way
  - Otherwise the stings are **certainly** different

Rabin-Karp algorithm

- The Rabin-Karp algorithm uses hashing to match strings

- To search for a string $P[1...m]$ in $T[1...n]$, we calculate $h(P)$ and then compare every substring $T[i..i+m\text{-}1]$ hash value
  - If the hash values are the same, it is **possible** that $P[1...m] = T[i...i+m\text{-}1]$, so we compare the substring the usual way
  - Otherwise the stings are **certainly** different

- False positives are rare. Hence, the $O(m)$ time it takes to compare the strings can be ignored

Rabin-Karp algorithm

- Repeatedly hashing strings of length *m* is inefficient

Rabin-Karp algorithm

- Repeatedly hashing strings of length *m* is inefficient
- In practice, hash values are calculated **incrementally**, as follows:

$$h\left(T\left[j \ldots j+m-1\right]\right) = \sum_{i=0}^{m-1} \mathrm{chr}(T\left[j+i\right]) \times \alpha^{m-i-1}$$

where α is the alphabet size.

Rabin-Karp algorithm

- Repeatedly hashing strings of length *m* is inefficient
- In practice, hash values are calculated **incrementally**, as follows:

$$h\left(T\left[j \ldots j+m-1\right]\right) = \sum_{i=0}^{m-1} \mathrm{chr}(T\left[j+i\right]) \times \alpha^{m-i-1}$$

  where α is the alphabet size.
- We can get the hash value for the substring starting at $j+1$ **cheaply**:

$$h\left(T\left[j+1 \ldots j+m\right]\right) = \left(h\left(T\left[j \ldots j+m-1\right]\right) - \alpha^{m-1}\mathrm{chr}\left(T\left[j\right]\right)\right) \times \alpha + \mathrm{chr}\left(T\left[j+m\right]\right)$$

  modulo m, i.e., we just subtract the contribution of $T[j]$ and add the contribution of $T[j+m]$, with two multiplications, one addition and one subtraction.

Rabin-Karp algorithm

- Let's work with this example ($\alpha = 4$)
  - The hash function $h(k) = k \bmod 11$ (the alphabet is too small)
  - The pattern is $P[1..m] = \text{TAATG}$ ($m = 5$, $k = 603$, $h = 9$)
  - The string is $T[1..n] = \text{GACCGCGTGAGATAATGTCA}$
  - The alphabet is [A T G C]

Rabin-Karp algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED ($k = 915$, $h = 2$)	T	A	A	T	G															

Rabin-Karp algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED ($k = 915$, $h = 2$)	T	A	A	T	G															
FAILED ($k = 592$, $h = 9$, T!=A)		T	A	A	T	G														

Rabin-Karp algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED ($k = 915$, $h = 2$)	T	A	A	T	G															
FAILED ($k = 592$, $h = 9$, T!=A)		T	A	A	T	G														
FAILED ($k = 1347$, $h = 5$)			T	A	A	T	G													

Rabin-Karp algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED ($k = 915$, $h = 2$)	T	A	A	T	G															
FAILED ($k = 592$, $h = 9$, T!=A)		T	A	A	T	G														
FAILED ($k = 1347$, $h = 5$)			T	A	A	T	G													
FAILED ($k = 1294$, $h = 7$)				T	A	A	T	G												

Rabin-Karp algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED ($k = 915$, $h = 2$)	T	A	A	T	G															
FAILED ($k = 592$, $h = 9$, T!=A)		T	A	A	T	G														
FAILED ($k = 1347$, $h = 5$)			T	A	A	T	G													
FAILED ($k = 1294$, $h = 7$)				T	A	A	T	G												
FAILED ($k = 1083$, $h = 5$)					T	A	A	T	G											

Rabin-Karp algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED ($k = 915$, $h = 2$)	T	A	A	T	G															
FAILED ($k = 592$, $h = 9$, T!=A)		T	A	A	T	G														
FAILED ($k = 1347$, $h = 5$)			T	A	A	T	G													
FAILED ($k = 1294$, $h = 7$)				T	A	A	T	G												
FAILED ($k = 1083$, $h = 5$)					T	A	A	T	G											
FAILED ($k = 1261$, $h = 7$)						T	A	A	T	G										

Rabin-Karp algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED ($k = 915$, $h = 2$)	T	A	A	T	G															
FAILED ($k = 592$, $h = 9$, T!=A)		T	A	A	T	G														
FAILED ($k = 1347$, $h = 5$)			T	A	A	T	G													
FAILED ($k = 1294$, $h = 7$)				T	A	A	T	G												
FAILED ($k = 1083$, $h = 5$)					T	A	A	T	G											
FAILED ($k = 1261$, $h = 7$)						T	A	A	T	G										
FAILED ($k = 951$, $h = 5$)							T	A	A	T	G									

Rabin-Karp algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED ($k = 915$, $h = 2$)	T	A	A	T	G															
FAILED ($k = 592$, $h = 9$, T!=A)		T	A	A	T	G														
FAILED ($k = 1347$, $h = 5$)			T	A	A	T	G													
FAILED ($k = 1294$, $h = 7$)				T	A	A	T	G												
FAILED ($k = 1083$, $h = 5$)					T	A	A	T	G											
FAILED ($k = 1261$, $h = 7$)						T	A	A	T	G										
FAILED ($k = 951$, $h = 5$)							T	A	A	T	G									
FAILED ($k = 733$, $h = 7$)								T	A	A	T	G								

Rabin-Karp algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED ($k = 915$, $h = 2$)	T	A	A	T	G															
FAILED ($k = 592$, $h = 9$, T!=A)		T	A	A	T	G														
FAILED ($k = 1347$, $h = 5$)			T	A	A	T	G													
FAILED ($k = 1294$, $h = 7$)				T	A	A	T	G												
FAILED ($k = 1083$, $h = 5$)					T	A	A	T	G											
FAILED ($k = 1261$, $h = 7$)						T	A	A	T	G										
FAILED ($k = 951$, $h = 5$)							T	A	A	T	G									
FAILED ($k = 733$, $h = 7$)								T	A	A	T	G								
FAILED ($k = 886$, $h = 6$)									T	A	A	T	G							

Rabin-Karp algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED ($k = 915$, $h = 2$)	T	A	A	T	G															
FAILED ($k = 592$, $h = 9$, T!=A)		T	A	A	T	G														
FAILED ($k = 1347$, $h = 5$)			T	A	A	T	G													
FAILED ($k = 1294$, $h = 7$)				T	A	A	T	G												
FAILED ($k = 1083$, $h = 5$)					T	A	A	T	G											
FAILED ($k = 1261$, $h = 7$)						T	A	A	T	G										
FAILED ($k = 951$, $h = 5$)							T	A	A	T	G									
FAILED ($k = 733$, $h = 7$)								T	A	A	T	G								
FAILED ($k = 886$, $h = 6$)									T	A	A	T	G							
FAILED ($k = 473$, $h = 0$)										T	A	A	T	G						

Rabin-Karp algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED ($k = 915$, $h = 2$)	T	A	A	T	G															
FAILED ($k = 592$, $h = 9$, T!=A)		T	A	A	T	G														
FAILED ($k = 1347$, $h = 5$)			T	A	A	T	G													
FAILED ($k = 1294$, $h = 7$)				T	A	A	T	G												
FAILED ($k = 1083$, $h = 5$)					T	A	A	T	G											
FAILED ($k = 1261$, $h = 7$)						T	A	A	T	G										
FAILED ($k = 951$, $h = 5$)							T	A	A	T	G									
FAILED ($k = 733$, $h = 7$)								T	A	A	T	G								
FAILED ($k = 886$, $h = 6$)									T	A	A	T	G							
FAILED ($k = 473$, $h = 0$)										T	A	A	T	G						
FAILED ($k = 869$, $h = 0$)											T	A	A	T	G					

Rabin-Karp algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED ($k = 915$, $h = 2$)	T	A	A	T	G															
FAILED ($k = 592$, $h = 9$, T!=A)		T	A	A	T	G														
FAILED ($k = 1347$, $h = 5$)			T	A	A	T	G													
FAILED ($k = 1294$, $h = 7$)				T	A	A	T	G												
FAILED ($k = 1083$, $h = 5$)					T	A	A	T	G											
FAILED ($k = 1261$, $h = 7$)						T	A	A	T	G										
FAILED ($k = 951$, $h = 5$)							T	A	A	T	G									
FAILED ($k = 733$, $h = 7$)								T	A	A	T	G								
FAILED ($k = 886$, $h = 6$)									T	A	A	T	G							
FAILED ($k = 473$, $h = 0$)										T	A	A	T	G						
FAILED ($k = 869$, $h = 0$)											T	A	A	T	G					
FAILED ($k = 406$, $h = 10$)												T	A	A	T	G				

Rabin-Karp algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED ($k = 915$, $h = 2$)	T	A	A	T	G															
FAILED ($k = 592$, $h = 9$, T!=A)		T	A	A	T	G														
FAILED ($k = 1347$, $h = 5$)			T	A	A	T	G													
FAILED ($k = 1294$, $h = 7$)				T	A	A	T	G												
FAILED ($k = 1083$, $h = 5$)					T	A	A	T	G											
FAILED ($k = 1261$, $h = 7$)						T	A	A	T	G										
FAILED ($k = 951$, $h = 5$)							T	A	A	T	G									
FAILED ($k = 733$, $h = 7$)								T	A	A	T	G								
FAILED ($k = 886$, $h = 6$)									T	A	A	T	G							
FAILED ($k = 473$, $h = 0$)										T	A	A	T	G						
FAILED ($k = 869$, $h = 0$)											T	A	A	T	G					
FAILED ($k = 406$, $h = 10$)												T	A	A	T	G				
FOUND $i = 13$ ($k = 603$, $h = 9$)													T	A	A	T	G			

Hashing

- When applicable, a well-tuned hashing table will **outperform** its competitors
  - **Unless** the load factor becomes too high, or the hash function is not robust

Hashing

- When applicable, a well-tuned hashing table will **outperform** its competitors
  - **Unless** the load factor becomes too high, or the hash function is not robust
  - Continuous verification of the key's spread on the table may be necessary

Hashing

- When applicable, a well-tuned hashing table will **outperform** its competitors
  - **Unless** the load factor becomes too high, or the hash function is not robust
  - Continuous verification of the key's spread on the table may be necessary

- Hashing does have **drawbacks**:
  - If the application calls for traversal of all items in sorted order, a hash table is no good

Hashing

- When applicable, a well-tuned hashing table will **outperform** its competitors
  - **Unless** the load factor becomes too high, or the hash function is not robust
  - Continuous verification of the key's spread on the table may be necessary

- Hashing does have **drawbacks**:
  - If the application calls for traversal of all items in sorted order, a hash table is no good
  - Deletion is virtually impossible, unless you use separate chaining

Hashing

- When applicable, a well-tuned hashing table will **outperform** its competitors
  - **Unless** the load factor becomes too high, or the hash function is not robust
  - Continuous verification of the key's spread on the table may be necessary

- Hashing does have **drawbacks**:
  - If the application calls for traversal of all items in sorted order, a hash table is no good
  - Deletion is virtually impossible, unless you use separate chaining
  - It may be hard to predict the volume of data, and rehashing is expensive.

With the functions h(k) = k mod 7 and h'(k) = 5 – (k mod 5), which are the addresses for [19 26 13 48 17] with double hashing?

[13(0) 48(1) 17(3) 19(5) 26(6)]

[48(0) 17(3) 26(4) 19(5) 13(6)]

[26(0) 48(2) 17(3) 19(5) 13(6)]

[48(1) 26(2) 17(3) 19(5) 13(6)]

Next lecture

- Dynamic programming
  - Basics of optimisation
  - The coin row problem (Levitin Section 8.1)
  - Knapsack problem (Levitin Section 8.2)
  - Message passing on a tree problem