COMP20007 Workshop Week 9

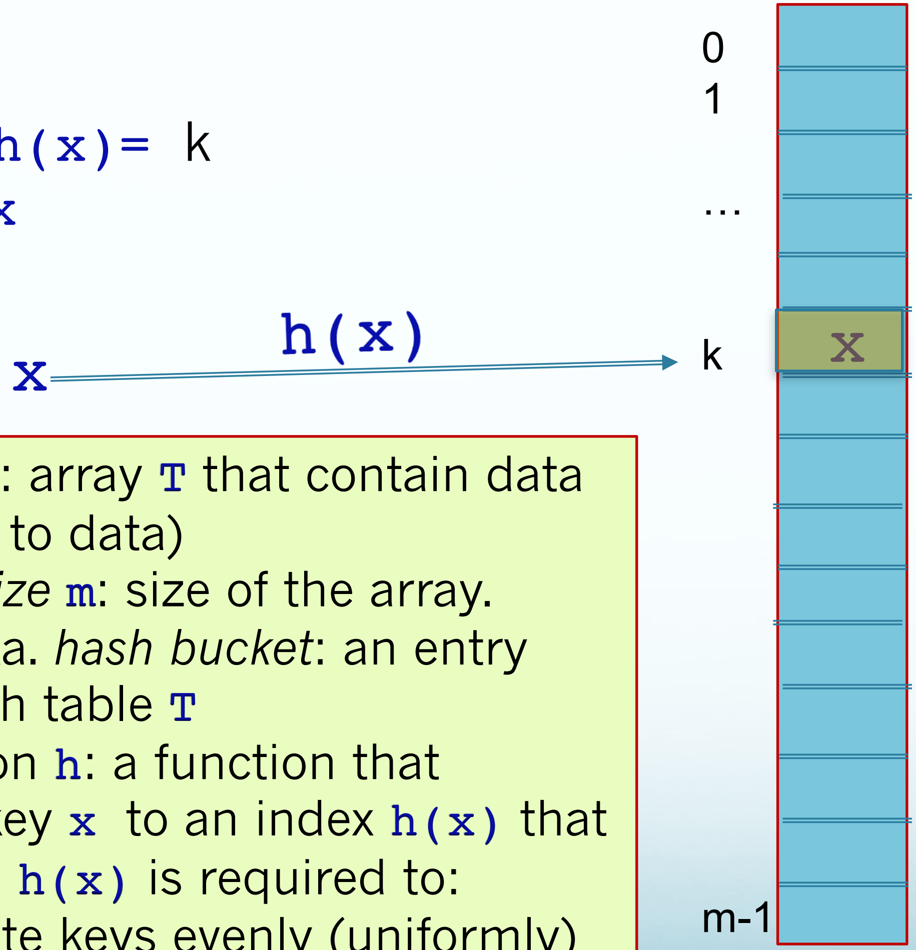
	Preparation: - *have draft papers and pen ready, or ready to work on whiteboard* - be ready with Ed.Week 10 Workshop
1	Hashing: Q 10.1, 10.2
2	Huffman Coding: Questions 10.3, 10.4
3	Revision on demands: Complexity (Question 10.5) Solving Recurrences
LAB	Lab: playing with hashing code

hashing/hashtable = ?

Hash Table: dictionary with average θ(1) search/insert/delete

- *Hashing= hash table* `T` + *hash function* `h(x)`: store key `x` at `T[h(x)]`

suppose `h(x)=` k
for some `x`



- *hash table* `T`: array `T` that contain data (or pointers to data)
- *hash table size* `m`: size of the array.
- *hash slot*, aka. *hash bucket*: an entry `T[i]` of hash table `T`
- hash function `h`: a function that converts a key `x` to an index `h(x)` that `0≤h(x)<m`, `h(x)` is required to:
  - distribute keys evenly (uniformly) along the table,
  - be efficient (θ(1))

Example:
storing a list of <= 100 student records, each student has a unique student number in the range of:

```
1. 1..100
2. 5,001..6,000
3. 1..10,000
```

h(x)= ?, m=?

Collisions

- Collision when `h(x1) = h(x2)` for some `x1≠ x2`.
- Collisions are normally unavoidable.

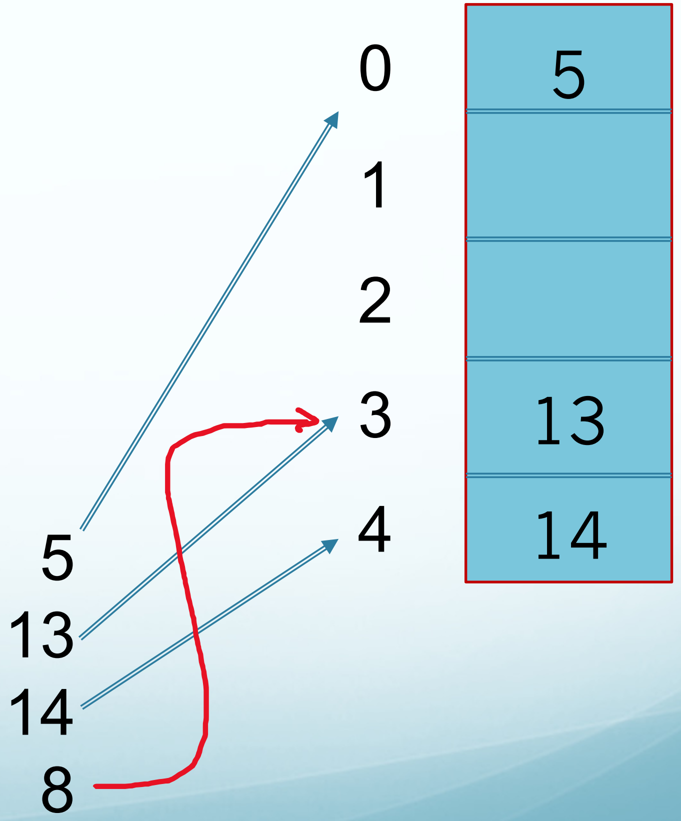
Collisions

Example:
`m=5`, `h(x)= x% m`
Here: `h(8)= h(5)`

Methods *to reduce collisions:*

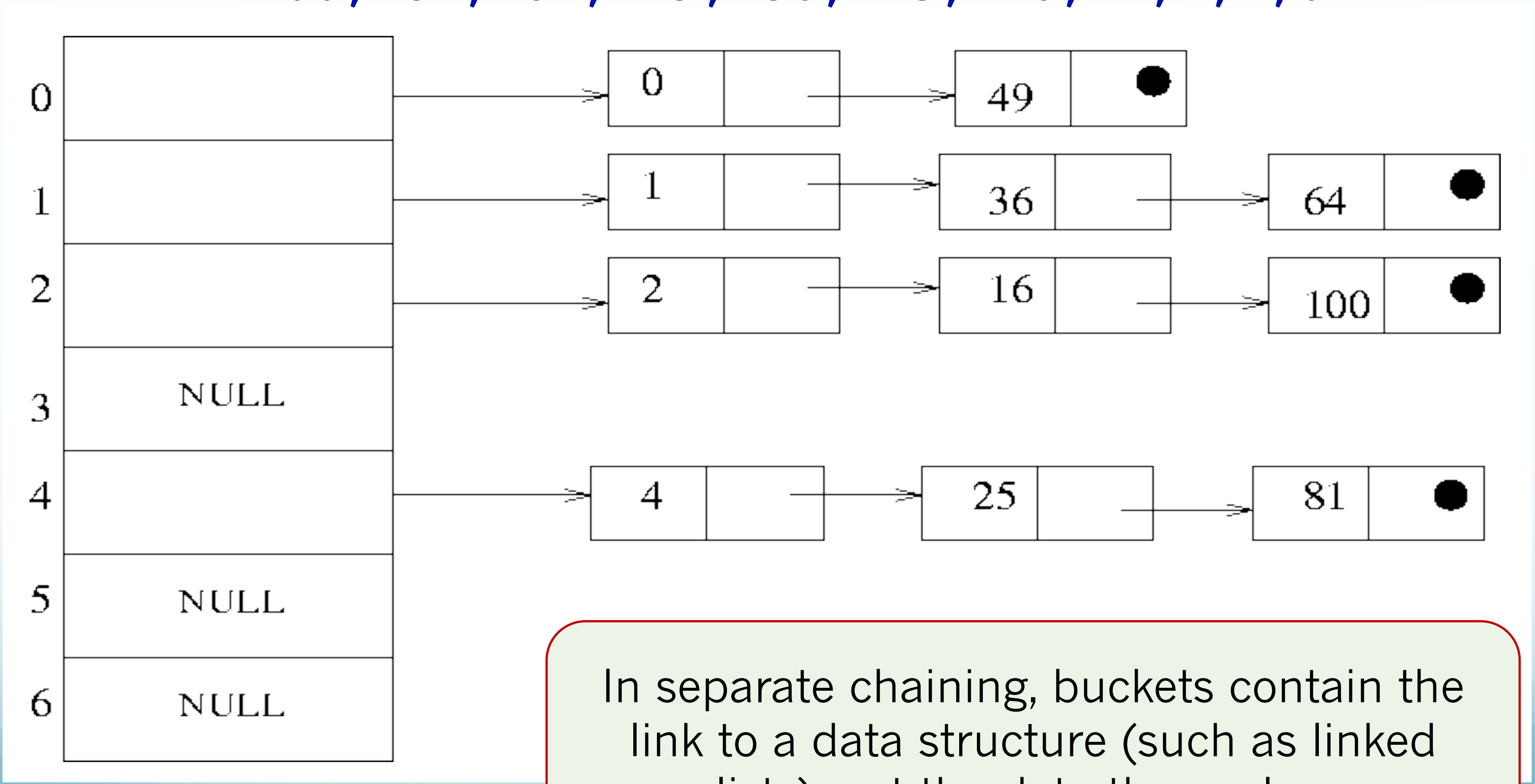
- using *a prime number* for hash table size `m`.
- making the table size `m` *big*

Even though, collisions might still happen



Collision Solution 1: Separate Chaining

`h(x) = x % 7,` keys entered in decreasing order:
`100, 81, 64, 49, 36, 25, 16, 4,2,1,0`



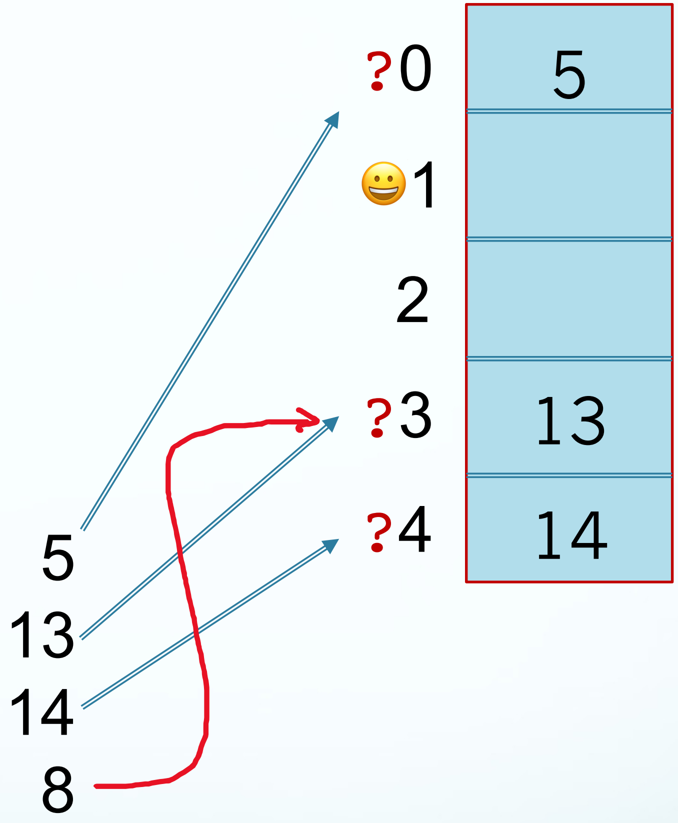
In separate chaining, buckets contain the link to a data structure (such as linked lists), not the data themselves.

Solution 2: Linear Probing (here, data are in the buckets)

linear probing= *when colliding, find the successive empty cell*.

Example: `m=5, h(x)= x mod m,`

keys inserted: 5, 13, 4, 8



Hashing with linear probing:

- When inserting we do some probes until getting a vacant slot:
  `h(x)` replaced by
  `H(x, probe)= (h(x) + probe) mod m`
  where `probe` is `0, 1, 2 …`
  until reaching a vacant slot
- The same procedure for search
- Deletion is problematic! (why?)

Double Hashing

When colliding, look forward for empty cells at distance `h2(x)`

Example: `m=5, h(x)= x mod m,`

`h2(x)= x mod 3,`

keys inserted: 5, 13, 4, 8

Hashing with double hashing:

similar to *linear probing*, but employ a second hash function h2(x):

`H(x,probe)= (h(x)+ probe*h2(x)) mod m`

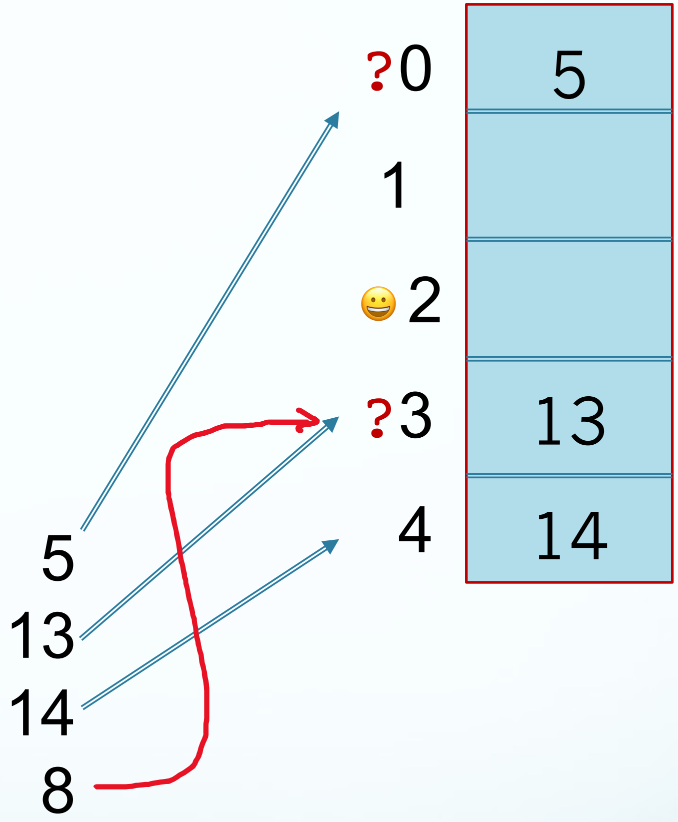
where `probe` is `0, 1, 2, …` (until reaching a vacant slot).

Note that:

- `h2(x) ≠ 0 for all x`, (why?)
- to be good, `h2(x)` should be co-prime with `m`, (how?)

Note: *linear probing* is just a special case of *double hashing* when `h2(x)=1`.

Both linear probing and double hashing are referred to as *Open Addressing methods*.



Q 10.1, 10.2 [Group/Individual]: Separate chaining

Q 10.1: Consider a hash table in which the elements inserted into each slot are stored in a linked list. The table has a fixed number of slots `L=2`. The hash function to be used is `h(k)=k mod L`.

a) Show the hash table after insertion of records with the keys
`17 6 11 21 12 33 5 23 1 8 9`

b) Can you think of a better data structure to use for storing the records that overflow each slot?

Q 10.2: Consider a hash table in which each slot can hold one record and additional records are stored elsewhere in the table using linear probing with steps of size `i=1`. The table has a fixed number of slots `L=8`. The hash function to be used is `h(k)=k mod L`.

a) Show the hash table after insertion of records with the keys
`17 7 11 33 12 18 9`

b) Repeat using linear probing with steps of size `i = 2`. What problem arises, and what constraints can we place on `i` and `L` to prevent it?

c) Can you think of a better way to find somewhere else in the table to store overflows?

Q 10.1: Separate chaining

Consider a hash table in which the elements inserted into each slot are stored in a linked list. The table has a fixed number of slots `L=2`. The hash function to be used is `h(k)=k mod L`.

a) Show the hash table after insertion of records with the keys
`17 6 11 21 12 33 5 23 1 8 9`

b) Can you think of a better data structure to use for storing the records that overflow each slot?

Your solution & notes:

Q 10.2: Open addressing

Consider a hash table in which each slot can hold one record and additional records are stored elsewhere in the table using linear probing with steps of size `i=1`. The table has a fixed number of slots `L=8`. The hash function to be used is `h(k)=k mod L`.

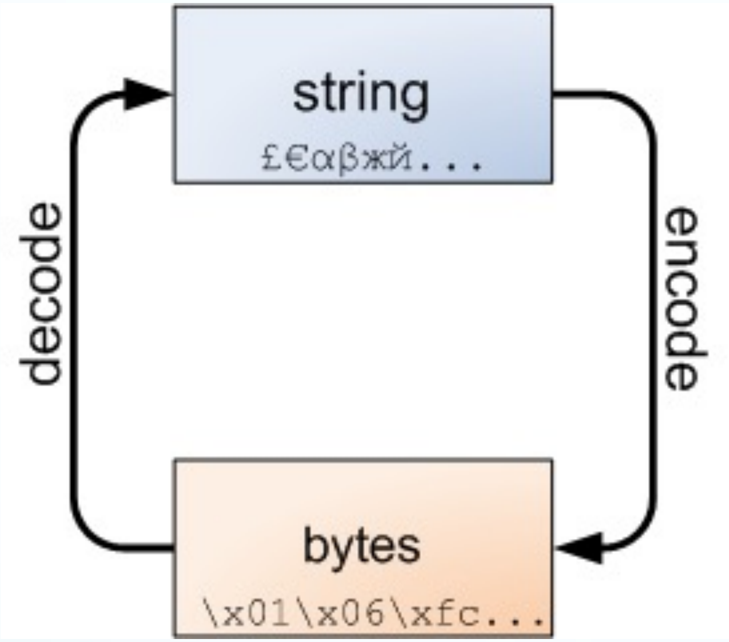
a) Show the hash table after insertion of records with the keys
`17 7 11 33 12 18 9`

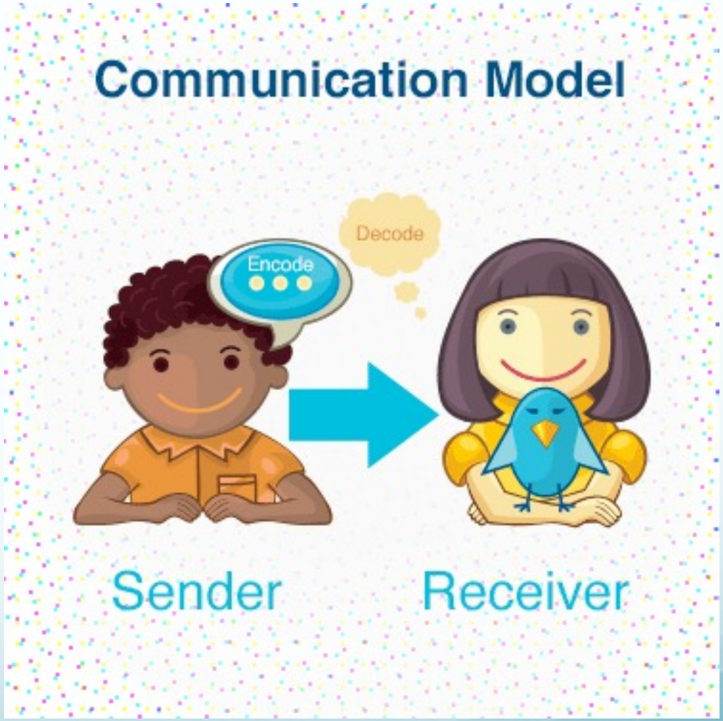
b) Repeat using linear probing with steps of size `i = 2`. What problem arises, and what constraints can we place on `i` and `L` to prevent it?

c) Can you think of a better way to find somewhere else in the table to store overflows?

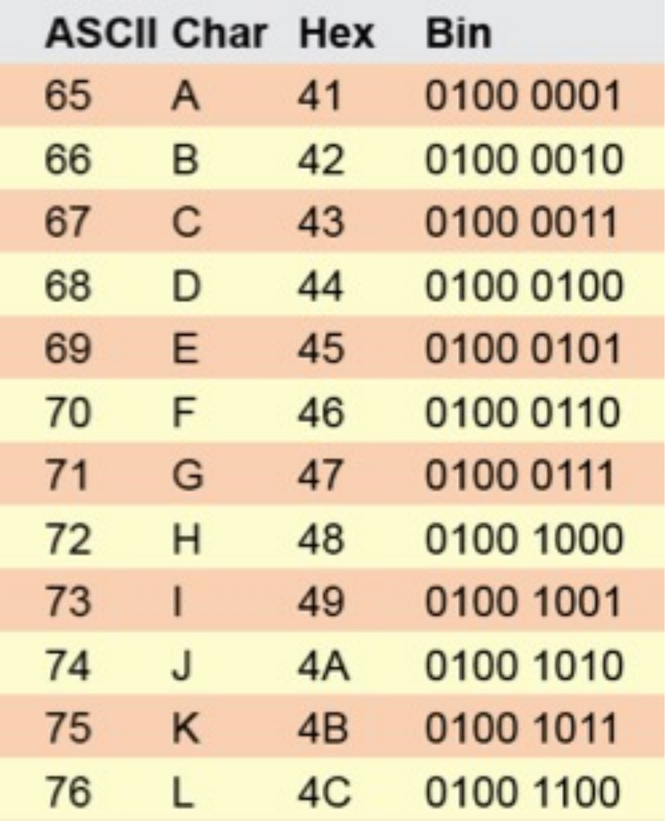
Your solution & notes:

Coding: used for storage, communication and ...

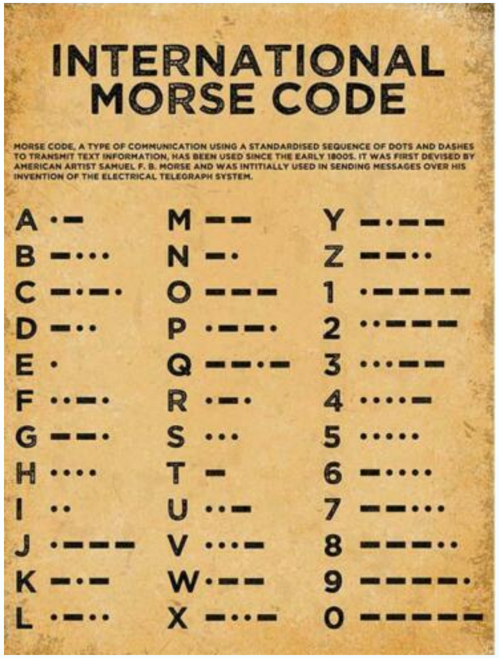




Coding & Data Compression



ASCII	Char	Hex	Bin
65	A	41	0100 0001
66	B	42	0100 0010
67	C	43	0100 0011
68	D	44	0100 0100
69	E	45	0100 0101
70	F	46	0100 0110
71	G	47	0100 0111
72	H	48	0100 1000
73	I	49	0100 1001
74	J	4A	0100 1010
75	K	4B	0100 1011
76	L	4C	0100 1100



INTERNATIONAL MORSE CODE

MORSE CODE, A TYPE OF COMMUNICATION USING A STANDARDISED SEQUENCE OF DOTS AND DASHES TO TRANSMIT TEXT INFORMATION, HAS BEEN USED SINCE THE EARLY 1800S. IT WAS FIRST DEVISED BY AMERICAN ARTIST SAMUEL F. B. MORSE AND WAS INTITIALLY USED IN SENDING MESSAGES OVER HIS INVENTION OF THE ELECTRICAL TELEGRAPH SYSTEM.

A	·–	M	––	Y	–·––
B	–···	N	–·	Z	––··
C	–·–·	O	–––	1	·––––
D	–··	P	·––·	2	··–––
E	·	Q	––·–	3	···––
F	··–·	R	·–·	4	····–
G	––·	S	···	5	·····
H	····	T	–	6	–····
I	··	U	··–	7	––···
J	·–––	V	···–	8	–––··
K	–·–	W	·––	9	––––·
L	·–··	X	–··–	0	–––––

a variable-length code

a fixed-length code

= ?

Encoding (Compressing)

The task:

Input: a message T such as `that cat, that bat, that hat` over some alphabet

Output: an encoded message T' – an efficient storage of T with the guarantee that T can be reproduced from T'. For example

T'= 111000111010001011…

ASCII code: each letter is replaced by a 8-bit codeword → 28 bytes

Principle of Data Compression: Use less number of bits (shorter codeword) for symbol that appears more frequently.

Input message: `that cat, that bat, that hat`

alphabet= [`a` `b` `c` `h` `t` `,` `  `] (or perhaps all ASCII characters)

How to compress: 3 steps

1.Modeling: making assumptions about the structure of messages

`that cat, that bat, that hat` (character model)

`that cat, that bat, that hat` (word model)

`that cat, that bat, that hat` (bi-character model)

2. Statistics: find symbol distribution

3. Coding: build the *code table* and do *encoding*

Input message: `that cat, that bat, that hat`

alphabet= [a b c h t ,] (or perhaps all ASCII characters)

1. Modeling: making assumptions about the structure of messages

`that cat, that bat, that hat` (character model)

2. Statistics: build table of frequencies, aka weight table. For this character model:

a	b	c	h	t	,	_
6/28	1/28	1/28	4/28	9/28	2/28	5/28

or just

a	b	c	h	t	,	_
6	1	1	4	9	2	5

3. Coding: build the *code table* and do *encoding*

a	b	c	h	t	,	_
01	0000	0001	100	11	001	101

→ 11100011110100010111...

Input message: `that cat, that bat, that hat`

alphabet= [`a` `b` `c` `h` `t` `,` ` `] (or perhaps all

1. Modeling: making assumptions about the st

`that cat, that bat, that hat`

2. Statistics: build table of frequencies,

a	b	c	h
6/28	1/28	1/28	4/2

or just

a	b	c	h
6	1	1	4

3. Coding: build the *code table* and do *encodi*

a	b	c	h	t		
01	0000	0001	100	11	001	101

→ 11100011110100010111...

this code is prefix-free: no codeword is a prefix of another codeword

[so, decoding is possible]

Huffman Coding = a method for building *minimum-redundancy* code (given a table of frequencies)

Build Huffman code:

- make a node for each weight
- join 2 *smallest weights* and make a parent node (of binary tree), continue until having a single root
- for each node, assign 0- and 1-bit for 2 associated edges

Example: build a Huffman code for

a	b	c	h	t	,	_
6	1	1	4	9	2	5

do it!

Huffman Coding

Note: there are different versions of Huffman code for a same weight table (depending on dealing with ties, assigning 0- and 1-bits), all we need to do is to choose a way and keep consistency. For instance (*canonical Huffman's coding*):

- when joining 2 weights into one, always make the smaller weight be the left child (hence, need to always keeps current roots in weight ordering)
- choose a consistent way for breaking ties
- when assigning code, always set 0 to the left edge, 1 to the right edge

Additional notes

- When sending encoded messages, the sender also need to send the weight table (or something equivalent).
- It's important that the receiver/decoder builds the code in the same way as the sender/encoder does.

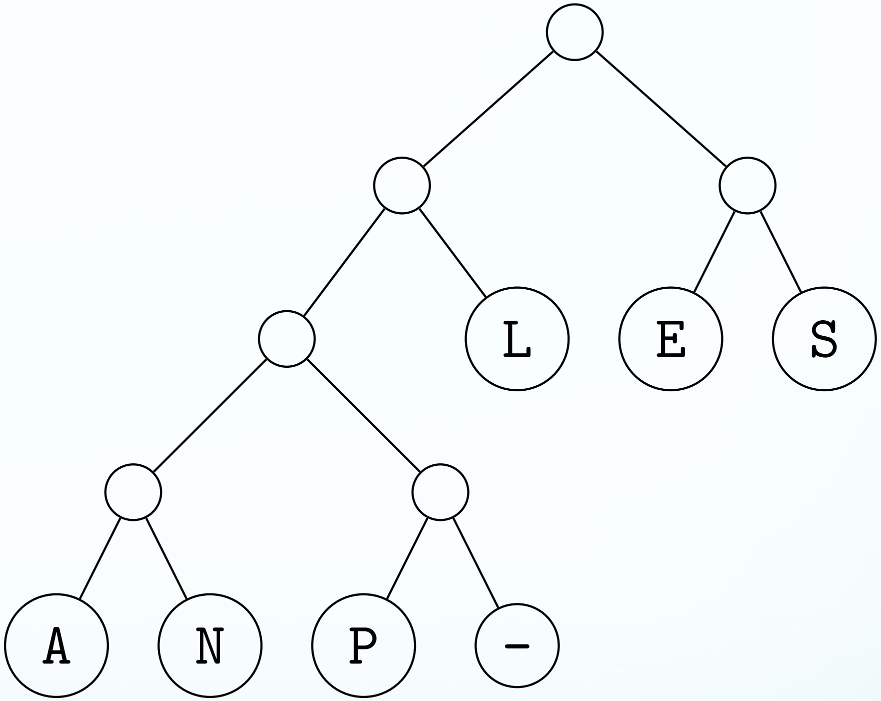
Q 10.3&4: Huffman Code Generation

Q 10.3: Huffman's Algorithm generates prefix-free code trees for a given set of symbol frequencies. Using these algorithms generate two code trees based on the frequencies in the following message:

`losslesscodes`

What is the total length of the compressed message using the Huffman code?

Problem 4: the code tree



Q 10.4: Decode:

00100110000011100011011011110011110110100010011011110001101111

Q 10.3: Huffman Code Generation

Huffman's Algorithm generates prefix-free code trees for a given set of symbol frequencies. Using these algorithms generate two code trees based on the frequencies in the following message:

`losslesscodes`

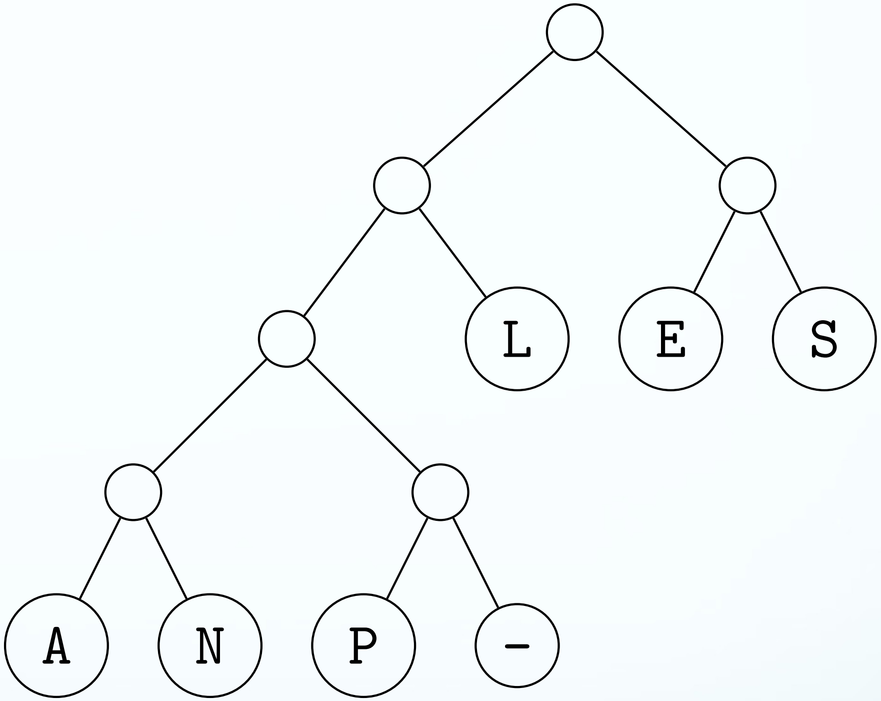
What is the total length of the compressed message using the Huffman code?

Your solution:

Q 10.4: Canonical Huffman decoding

The code tree was generated using Huffman's algorithm, and converted into a Canonical Huffman code tree. Note: _ denotes space.

Assign codewords to the symbols in the tree, such that left branches are denoted 0 and right branches are denoted 1.
Use the resulting code to decompress the following message:



00100110000011100011011011110011110110100010011011110001101111

Your soln:

Revision 1: Complexity Analysis – Lec W2, Workshop W3; W4- recurrences,

$1 \prec \log n \prec n^{\varepsilon} \prec n^{c} \prec n^{\log n} \prec c^{n} \prec n^{n}$ where $0 < \varepsilon < 1 < c$

$O(\,f(n) + g(n)\,) = O(\,\max\{f(n), g(n)\}\,)$
$O(c\,f(n)) = O(\,f(n)\,)$
$O(\,f(n) \times g(n)\,) = O(f(n)) \times O(g(n))$

note: these 3 also applied to big-θ

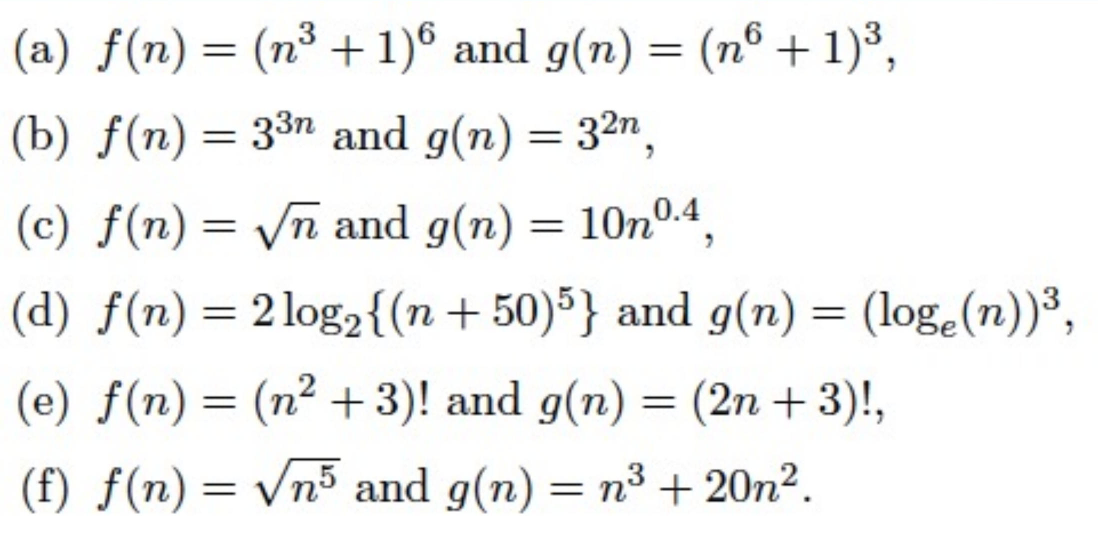
$1+2+ \ldots + n = n(n+1)/2 = \Theta(n^2)$
$1^2 + 2^2 + \ldots + n^2 = n(n+1)(2n+1)/6 = \Theta(n^3)$
$1 + x + x^2 + \ldots + x^n = (x^{n+1}-1)/(x-1) \quad (x \neq 1)$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & f(n) = O(g(n)) \\ c & f(n) = \Theta(\,g(n)\,) \\ \infty & f(n) = \Omega(g(n)) \end{cases}$$

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

Revision exercises: Q 10.5+

Q 10.5: For each of the following cases, indicate whether *f(n)* is *O(g(n))*, or ***Ω****(g(n))*, or both (that is, ***ϴ****(g(n)*)



(a) $f(n) = (n^3 + 1)^6$ and $g(n) = (n^6 + 1)^3$,

(b) $f(n) = 3^{3n}$ and $g(n) = 3^{2n}$,

(c) $f(n) = \sqrt{n}$ and $g(n) = 10n^{0.4}$,

(d) $f(n) = 2\log_2\{(n + 50)^5\}$ and $g(n) = (\log_e(n))^3$,

(e) $f(n) = (n^2 + 3)!$ and $g(n) = (2n + 3)!$,

(f) $f(n) = \sqrt{n^5}$ and $g(n) = n^3 + 20n^2$.

Q 10.5+: Solve the following recurrence relations. Give both a closed form expression in terms of n and a Big-Theta bound.

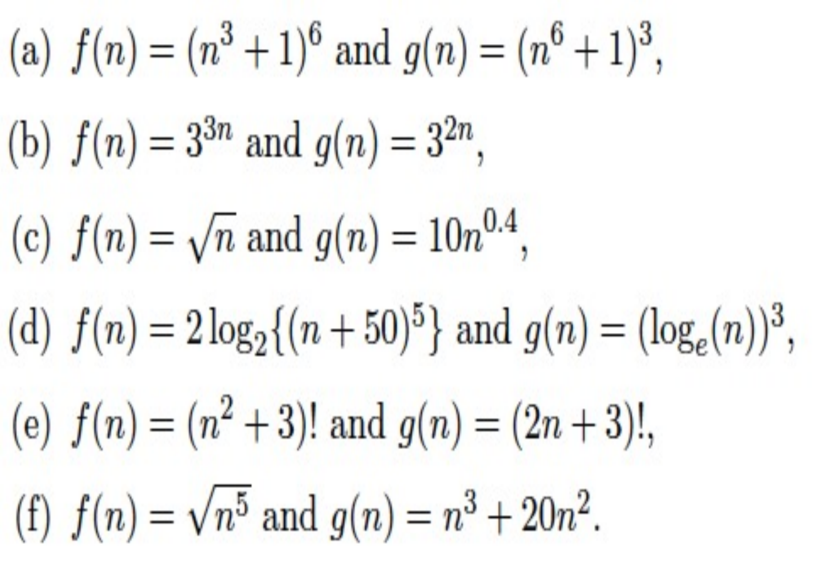
a) T(n)= T(n/2)+1, T(1)= 1

b) T(n)= T(n-1) + n/5, T(0)= 0

Other exercises: review complexity exercises for Workshops Week 3, 4, 7

R1 exercises: Q10.5

For each of the following cases, indicate whether *f(n)* is *O(g(n))*, or ***Ω****(g(n))*, or both (that is, ***ϴ****(g(n))*)



(a) $f(n) = (n^3+1)^6$ and $g(n) = (n^6+1)^3$,

(b) $f(n) = 3^{3n}$ and $g(n) = 3^{2n}$,

(c) $f(n) = \sqrt{n}$ and $g(n) = 10n^{0.4}$,

(d) $f(n) = 2\log_2\{(n+50)^5\}$ and $g(n) = (\log_e(n))^3$,

(e) $f(n) = (n^2+3)!$ and $g(n) = (2n+3)!$,

(f) $f(n) = \sqrt{n^5}$ and $g(n) = n^3 + 20n^2$.

Your solution/notes:

a)

b)

c)

d)

e)

R1 exercises: Q10.5+

Q 10.5+: Solve the following recurrence relations. Give both a closed form expression in terms of n and a Big-Theta bound.

a) $T(n) = T(n/2) + 1, \; T(1) = 1$

b) $T(n) = T(n-1) + n/5, \; T(0) = 0$

Notes:

- Apply the Master Theorem (`Workshop W7`) if possible (ie. if having a, b, and $\Theta(n^d)$ or $O(n^d)$, *and if the question just asks about big-O/big-* Θ)
- Otherwise, using substitution to expand until T(1) or T(0)

Your solution/notes:

a)

b)

Lab

“play” with the hashing code by following the instructions in Ed.

and/or continue with reviewing.