COMP20007 Workshop Week 12

LAB	**Preparation:** - *have draft papers and pen ready, or ready to work on whiteboard* - be ready with Ed.Week 12 Workshop 1. Hashing: Q 12.1, 12.2 2. Huffman Coding: Questions 12.3, 12.4 Assignment 2 OR: Revision on demands: 12.5: Quicksort & Mergesort 12.6/10.6+: Complexity 10.6: Solving Recurrences OR: Lab: playing with hashing code

hashing/hashtable = ?

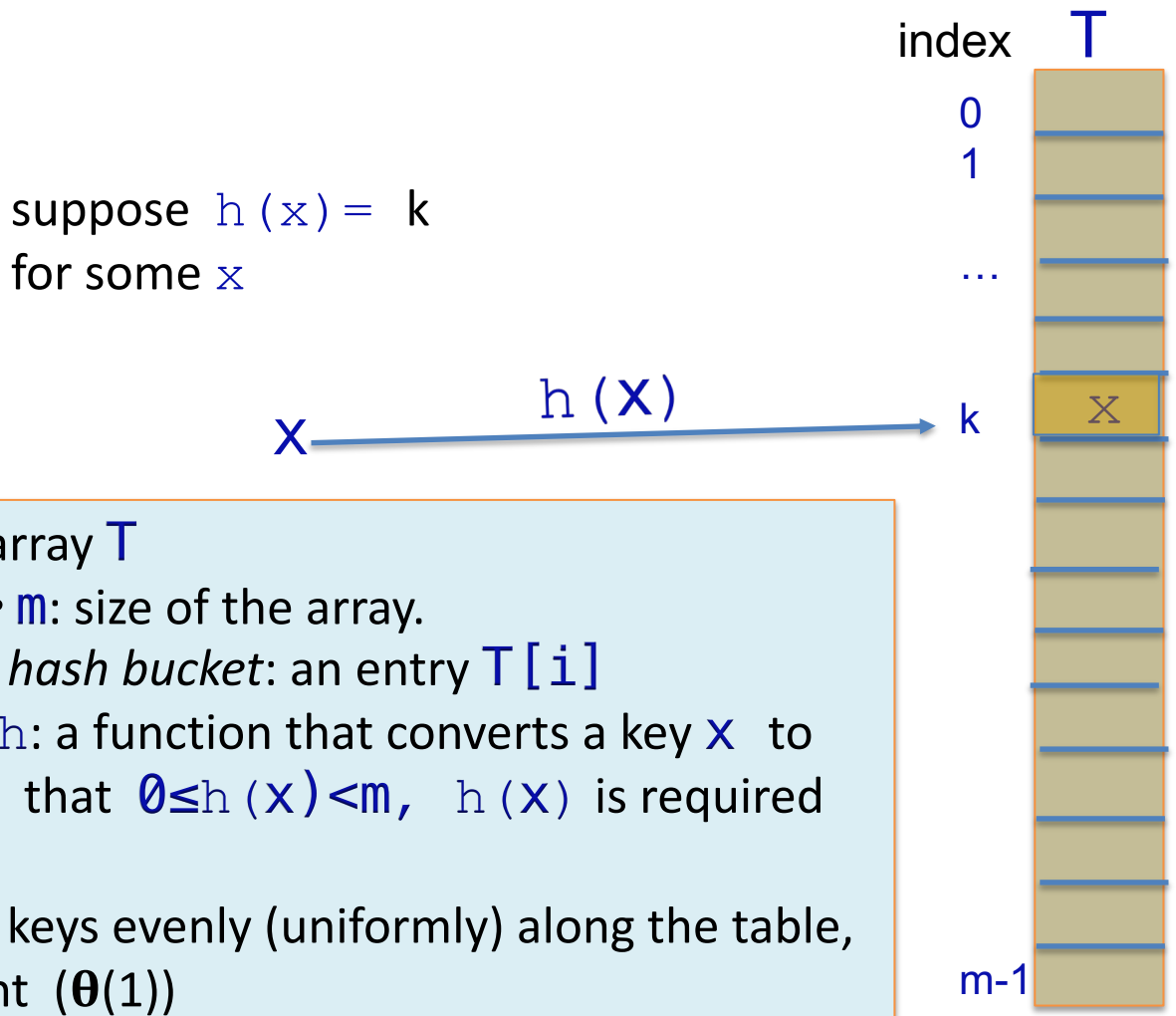
We have a dynamic collection of distinct integers and we need time-efficiency for all 3 basic operations: searching for, inserting, and deleting a datum.

What is (most) efficient data structure for a set of 70 distinct integer keys which have values:

a) unlimited in range
b) between 0 and 100
c) between 101 and 200
d) between 0 and 500

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

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



- *hash table* `T`: array `T`
- *hash table size* `m`: size of the array.
- *hash slot*, aka. *hash bucket*: an entry `T[i]`
- 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 ($\boldsymbol{\theta}(1)$)

Collisions

Collision when `h(x1) = h(x2)` for some `x1≠ x2`.

Examples:

`h(x)= x mod 101`

for `x1= 1`, `x2= 102`

Collisions are normally unavoidable.

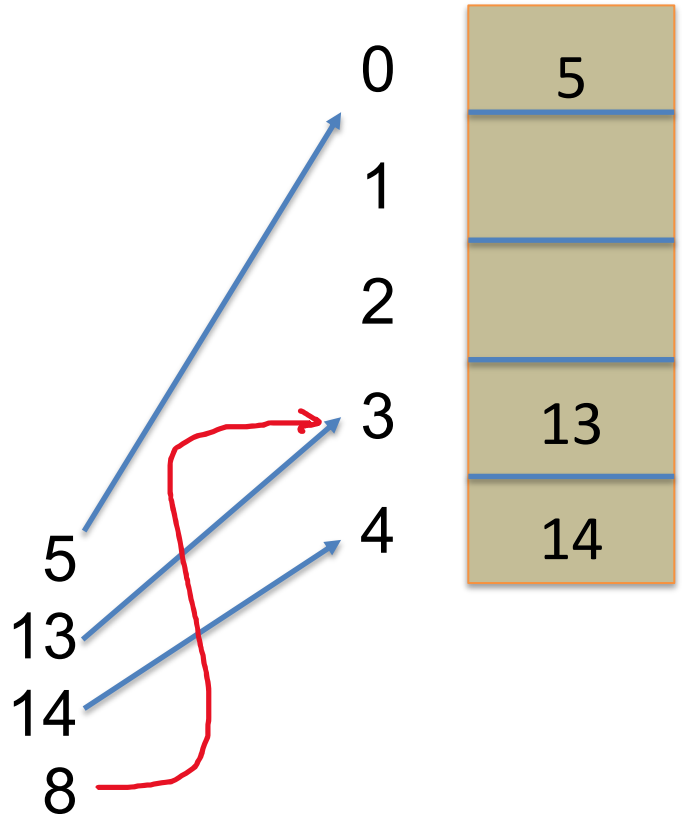
Collisions

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*

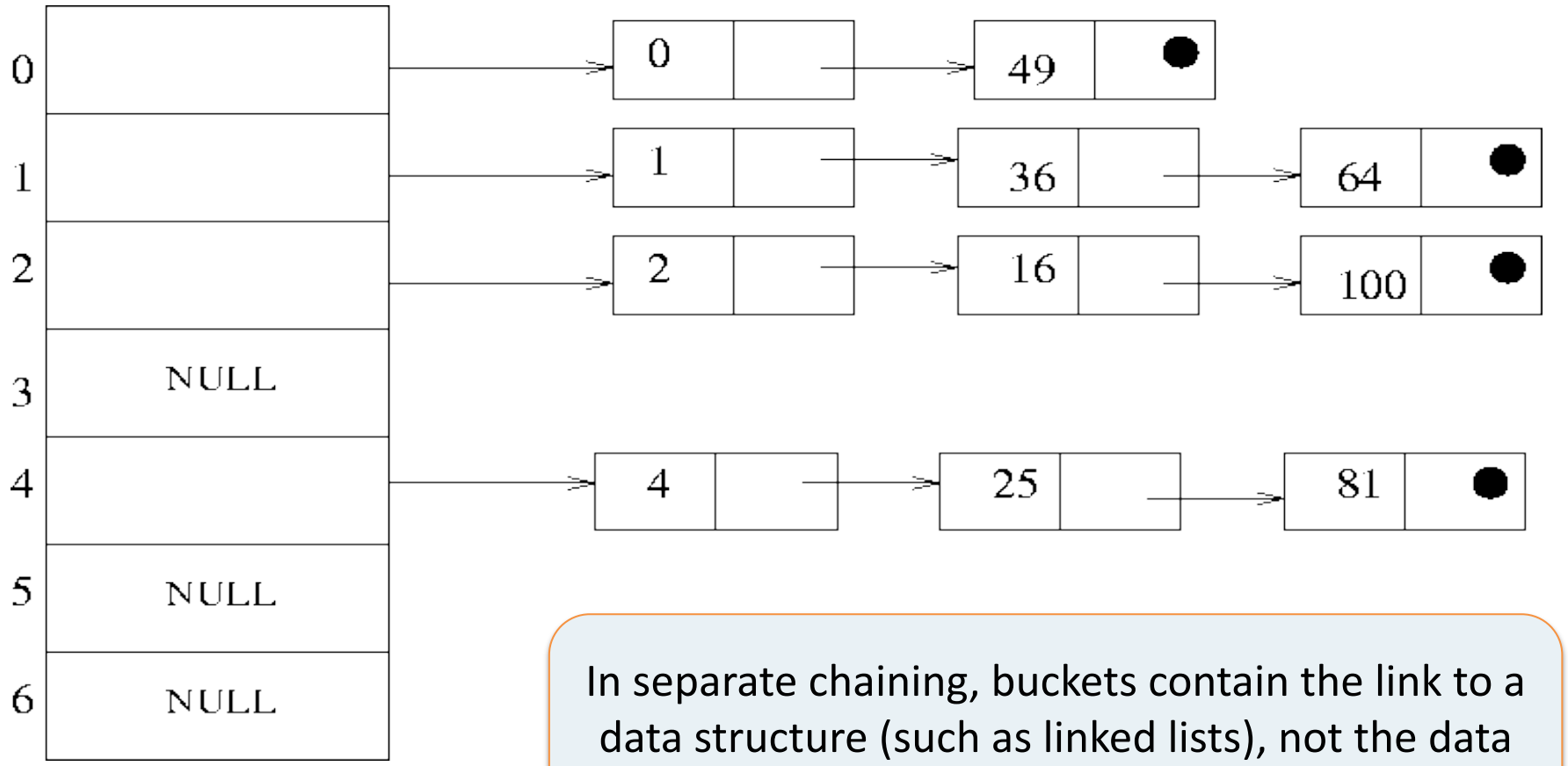
Even though, collisions might still happen



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`



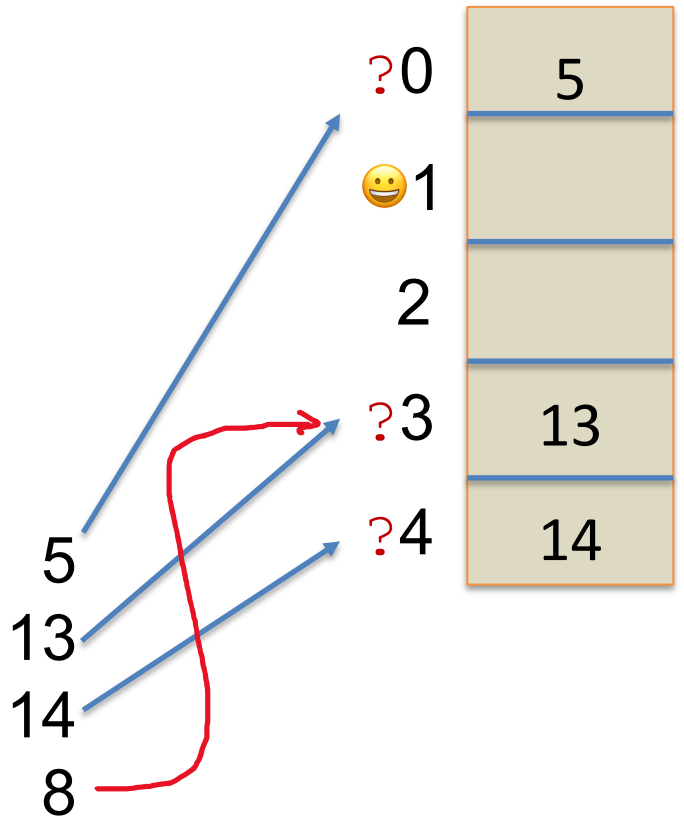
In separate chaining, buckets contain the link to a data structure (such as linked lists), not the data themselves.

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



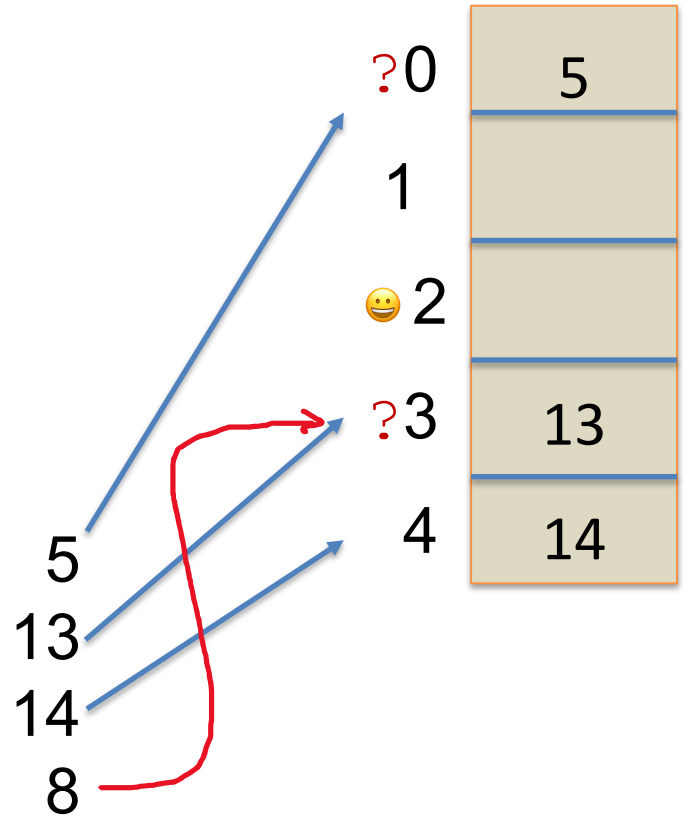
Hashing with linear probing:

- 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 …`
  ie. *examining forward with step= 1 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)`, *ie. examing forard with constant step=* `h2(x)`

Example: `m=5, h(x)= x mod m,`
`h2(x)= x mod 3 + 1,`
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 12.1, 12.2 [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?

Check: Q 12.1: 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:

a)

0	$6 \to 12 \to 8$
1	$17 \to 11 \to 21 \to 33 \to 5 \to 23 \to 1 \to 9$

b) array/sorted array? balanced search trees such as AVL or 2-3 tree?

but why the above hashing is so bad? perhaps we should solve a more general problem: how to make that hashing better, including increasing table size (with a prime number), changing hash function.

Check: Q 12.2: 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:

a)

0	1	2	3	4	5	6	7
	17	33	11	12	18	9	7

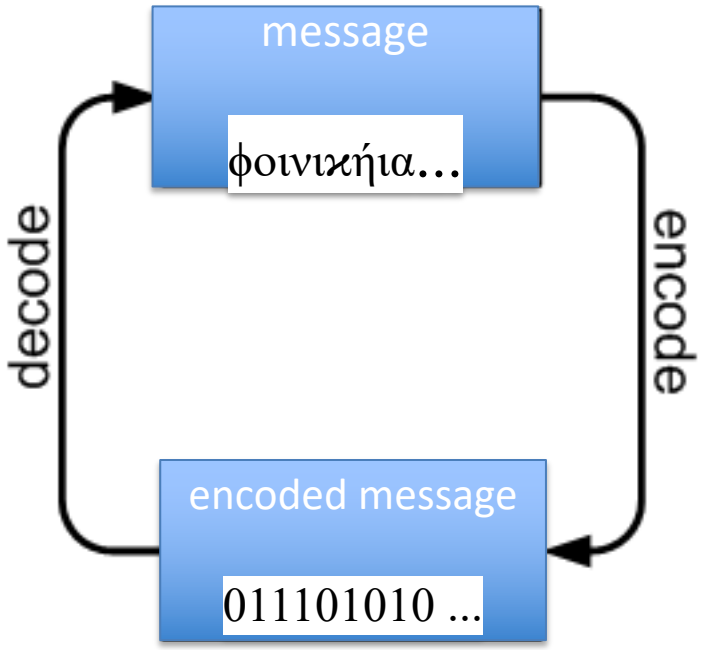
b)

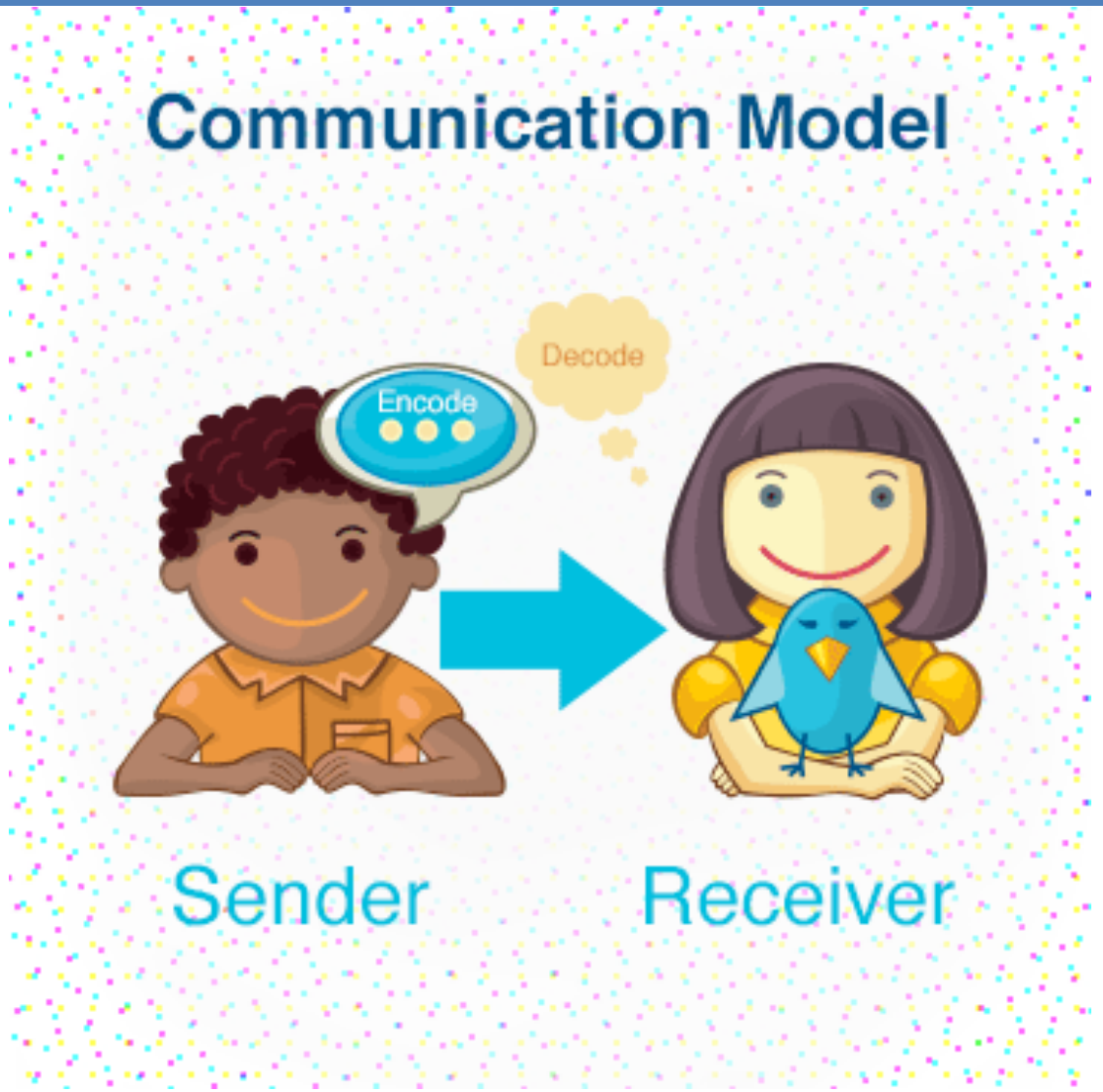
0	1	2	3	4	5	6	7
	17	18	11	12	33		7
	9?		9?		9?		9?

this is double hashing with h2(x)=2, trouble because m and 2 are not co-prime, choose h2(x)= odd number such as 3 would help!

c) double hashing with h2(x)≠ 0 and h2(x) co-prime with m, such as: m prime and h2(x)<m, or m=2^k, h2(x) odd

Coding: used for storage, communication, compression, …





Compression: the encoded message normally has smaller size (in bits)

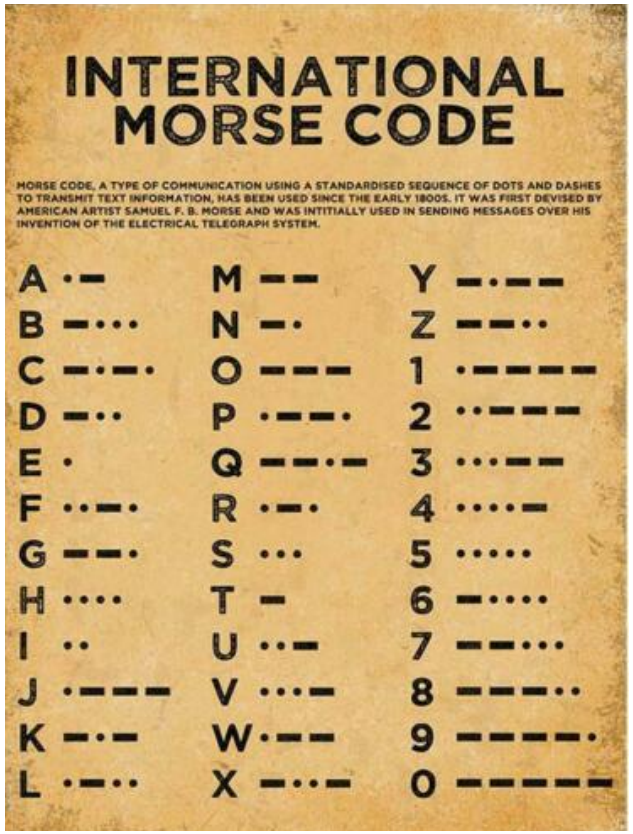
Encoding/Compressing: each symbol of the message is replaced with a bit pattern called its codeword

Decoding/De-compressing: encoded message is converted to the original message using codeword table

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

a fixed-length code

AB → 0100001010000010



a variable-length code

AB → 

codeword length	same bit-length for all codewords	shorter codewords for more-frequent symbols
space-efficient?	No (unless uniform distribution)	Yes, space-efficient
random access to k-th symbol	Yes	No, need to decode previous k-1 symbols first

Encoding (Compressing)

The task:

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

Output: an encoded message T' with the guarantee that T can be reproduced from T'.

For example

T'= 1110001111010001011 1…

ASCII code: each letter is replaced by a 8-bit codeword → need 28 bytes for the above T

Principle of Data Compression: Use less number of bits (shorter codeword) for symbol that appears more frequently → variable-length encoding

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, defining “symbol”:

`that cat, that bat, that hat` (character model: symbol=character)

`that cat, that bat, that hat` (word model : symbol=word)

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

2. Statistics: find symbol distribution (ie. frequency of each symbol)

3. Coding: using the distribution to build the *code table* then *encoding (*writing the *encoded message*)

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

then do the encoding, symbol-by-symbol

`that cat.....`

→ 11100011110100010111...

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

then do the encoding, symbol-by-symbol

`that cat.....`

→ 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 prefix-free* code

Build Huffman code using a given weight table = building a binary tree in bottom-up manner:

- make a node for each weight
- join 2 *smallest weights* and make a parent node (of binary tree) with sum of weights, 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!

Q 12.3: 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:

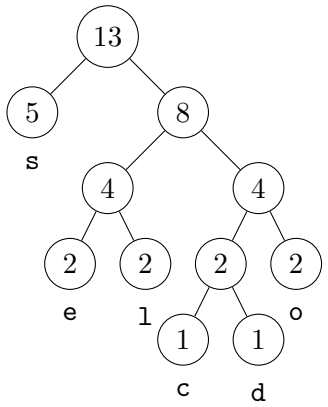
Check Q 12.3: Huffman Code Generation

`losslesscodes`

Your solution: one tree could be

Frequency counts:

s	l	o	e	c	d
5	2	2	2	1	1



s	0
l	101
o	111
e	100
c	1100
d	1101

Hence, the encoded version of `losslesscodes` is:

101 111 0 0 101 100 0 0 1100 111 1101 100 0

Note:

- give another tree
- other solutions are possible, but message length is always 31
- ➔ a need for a standard: canonical Huffman code

Canonical Huffman Coding [not important for this course, but good to know]

→ There are different versions of Huffman code for a same weight table, all we need to do is to choose a way and keep consistency.

example rules for 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)
- is same weight: simpler trees first + use alphabetical order
- when assigning code, always set 0 to the left edge, 1 to the right edge

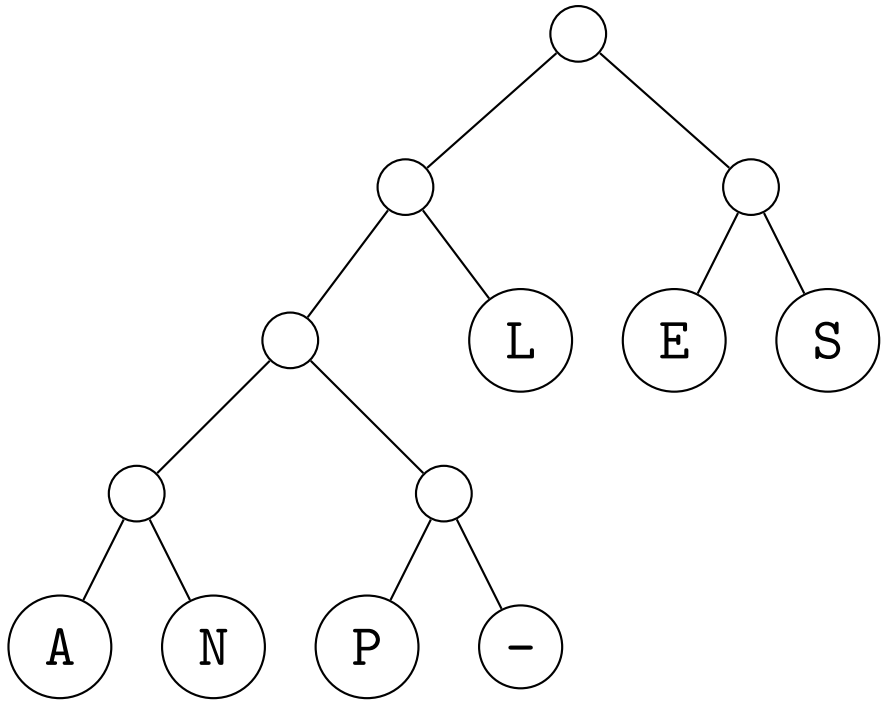
Additional notes

- When sending encoded messages, the sender also need to send the codeword table, or 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 12.4: 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:



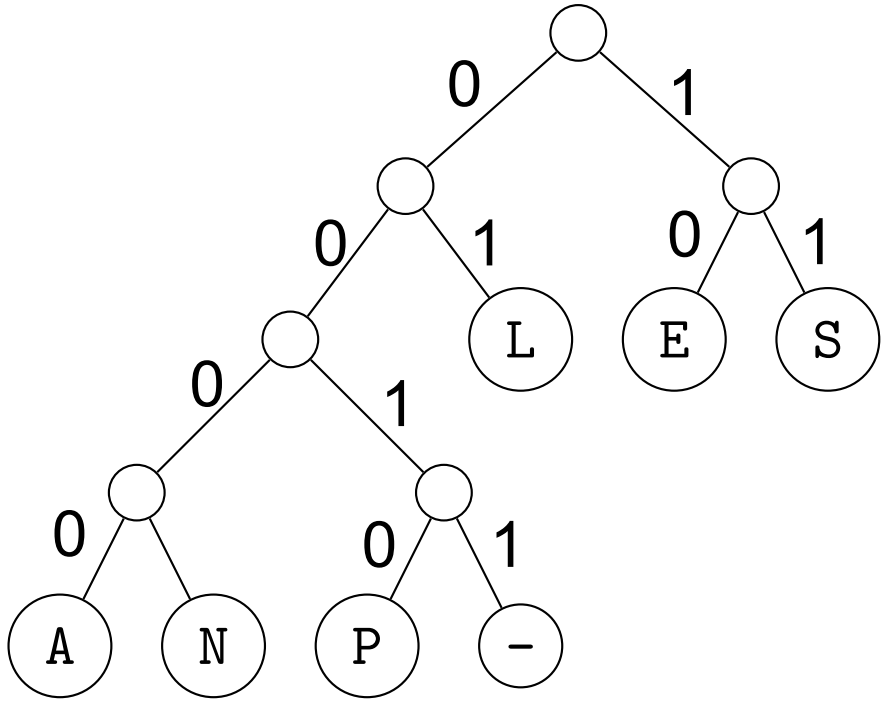

```
00100110000011100011011011110011111011010001001101111000 1101111
```

Your soln:

Q 12.4: 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:




```
0010011000001110001101101111001111011010001001101111001101111
```

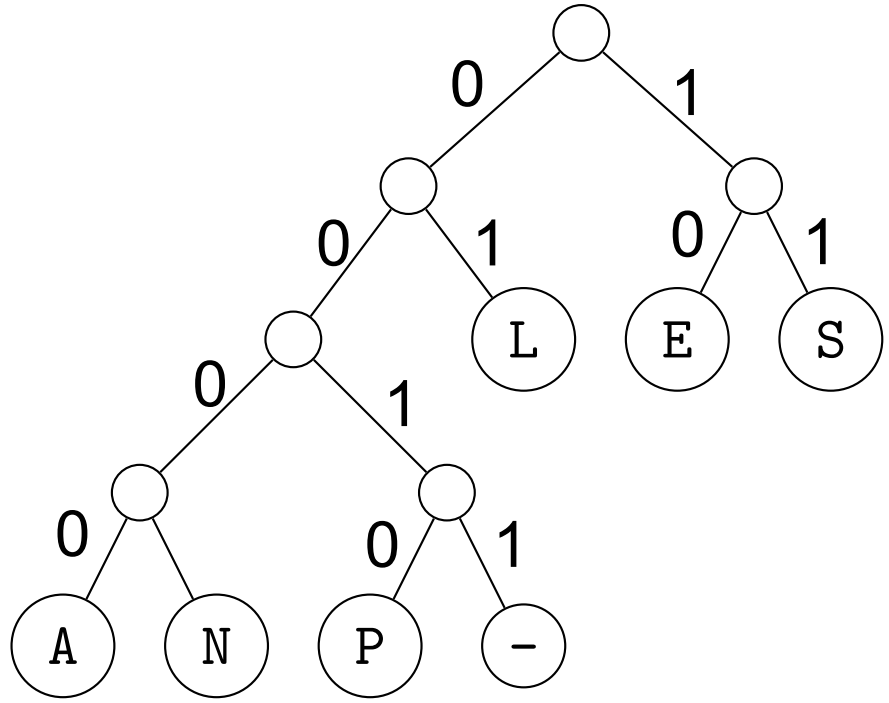
Your soln:

Check Q 12.4: 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:

symbol	codeword	length
A	0000	4
N	0001	4
P	0010	4
_	0011	4
L	01	2
E	10	2
S	11	2



00100110000011100011011011110011111011010001001101111000110111

Your soln: PLEASE LESS SLEEPLESSNESS

Revision exercises of Week 10 & 12

Q 12.6: Quicksort & Mergesort

Given the array:

```
[3, 8, 5, 2, 1, 3, 5, 4, 8]
```

a.Perform a single *Hoare Partition* on the array, taking the first element as the pivot

b.Perform Quicksort on the array. You may use whatever partitioning strategy you like (*i.e.*, you don't need to follow a particular algorithm).

c.Perform Mergesort on the array

Q 12.5/10.6+: For each of the following cases, indicate whether $f(n)$ is $O(g(n))$, or $\Omega(g(n))$, or both (that is, $\Theta(g(n))$)

(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 10.6: 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) $T(n) = T(n-1) + n/5$, $T(0) = 0$

Lab

- A2
- or “play” with the hashing code by following the instructions in Ed.
- and/or continue with reviewing.

Algorithms are fun!

Thank You!
&
Good Luck!