COMP20003 Workshop Week 6

Hashing and … assignment 2

Distribution Counting (aka. Counting Sort)

Hashing

Programming: W6.3

Peer Activities

Assignment 2: Q&A

Assignment 2

A Special Case for Sorting

Special Example: sort an array of non-negative integers, each ≤ 2:

```
input array: {0,1,2,0,0,1,2,1,1,0,0,0}
```

What sorting method is most time-efficient in this case?

An Unusual Sorting Method: Distribution Counting (aka. Counting Sort)

Special/Unusual:

- Not using key comparisons as in insertion sort, quick sort, ...

Conditions: keys are integers in a small range, ie. max(key)-min(key) ≪ n ,

Special Example: sort an array of non-negative integers, each ≤ 2:

```
input array: {0,1,2,0,0,1,2,1,1,0,0,0}
```

freq(0) =6 freq(1)= 4 freq(2)=2

Sorted array: { 0,0,0,0,0,0, 1,1,1,1, 2, 2 }

keys 0 start from index 0

keys 1 starts from index **6**

6 = freq(0)
= freq(<1)

keys 2 starts from index 10

10 = freq(0 & 1)
= freq(<**2**)

Counting Sort for sorting array A[0..n-1]

Input: A [i-1] + F[i-1]

Step 2: scan A[] and copy elements to sorted array B. For each A[j] :

B[C[[0..n-1] where

$0 \le A[i] \le k$, and $k \ll n$

Output: B[0..n-1] which is the sorted version of A[]

Step 1a: build frequency table F[] such that

F[**i**] = freq of **i** in A[]

Step 1b: compute the accumulator array C[]:

C[**i**] = starting position of value **i** in the sorted array B[].

C[0]= 0

C[i] := C[A[j]]++]= A[j]

which is equivalent to:

x := A[j]

B[C[x]] := A[j]

C[x]= C[x]+1

A[0..11]= {2,0,1,0,3, 01,2,1,1,,0,0}

k= 3

F[]= table of frequencies

idx	0	1	2	3	
	5	4	2	1	

C[i]= table of "next position for i"

idx	0	1	2	3	
	0	5	9	11	

A[]= {2,0,1,0,3,1,2,1,1,0,0,0}

B:

Distribution Counting summary

Unlike other sorting algorithms, *DC does not use key comparison.*

Normally not applicable. Can only be useful when

- all keys are integers in small range
- ie. when $min \le keys \le max$ and $max-min \in O(n)$

Time complexity, supposing $r = max-min+1$*:*

- $\Theta(n+r)$, or
- $\Theta(n)$ if $r \in O(n)$

Special properties:

- *not in-place*, ie. requiring additional arrays for data records
- additional memory: $\Theta(n+r)$, or $\Theta(n)$ if $r \in O(n)$
- the sorting is *stable*, ie. it preserves the relative order of equal keys

Hashing: Introduction

Task: Build a dictionary of n<1000 student records where stud_no (keys) are unique and in the range of 1..999. Insert and Search are major operations.

Q1: What concrete data type is the best?

Q2: What if keys are in the range of 1..99999 and we still want to keep the dictionary's size small?

Hashing: Introduction

Task: Build a dictionary of n<1000 student records where stud_no (keys) are unique and in the range of 1..999. Insert and Search are major operations.

Q1: What concrete data type is the best? Array `T[1000]`, store `key` in `T[key]`

Q2: What if keys are in the range of 1..99999 and we still want to keep the dictionary's size small? store `key` in `T[key % 1000]` ?

The technique is called hashing

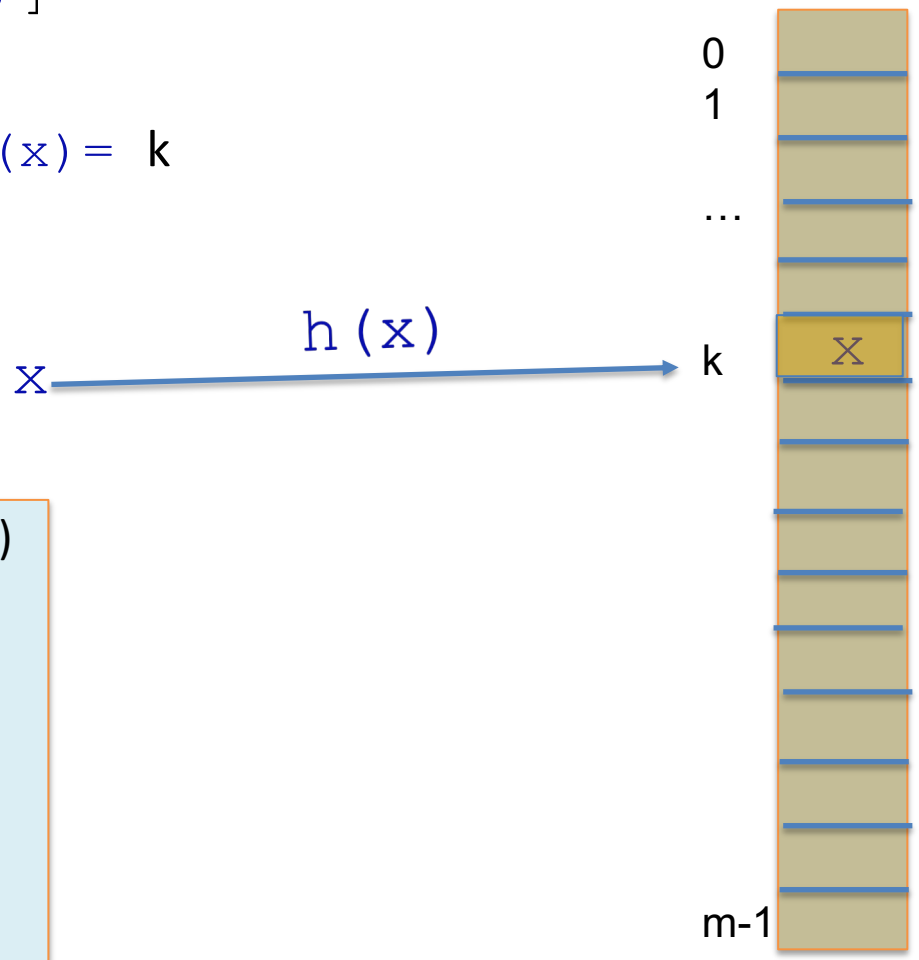
Hashing at a glance:

- just using an array to keep data (or keep track of data)
- not using key comparison to decide where to store x, just basically store at index h(x)
- average (=expected!) O(1) for insert/search/delete
- but probably tricky in how to map key to the array index

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

Hashing= *hash table* `T` + *hash function* `h(x)`:
store the record for 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 T.
- *hash slot*, aka. *hash bucket*: an entry `T[i]` of hash table `T`
- hash function `h`: a function that maps 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 (O(1))

Collisions

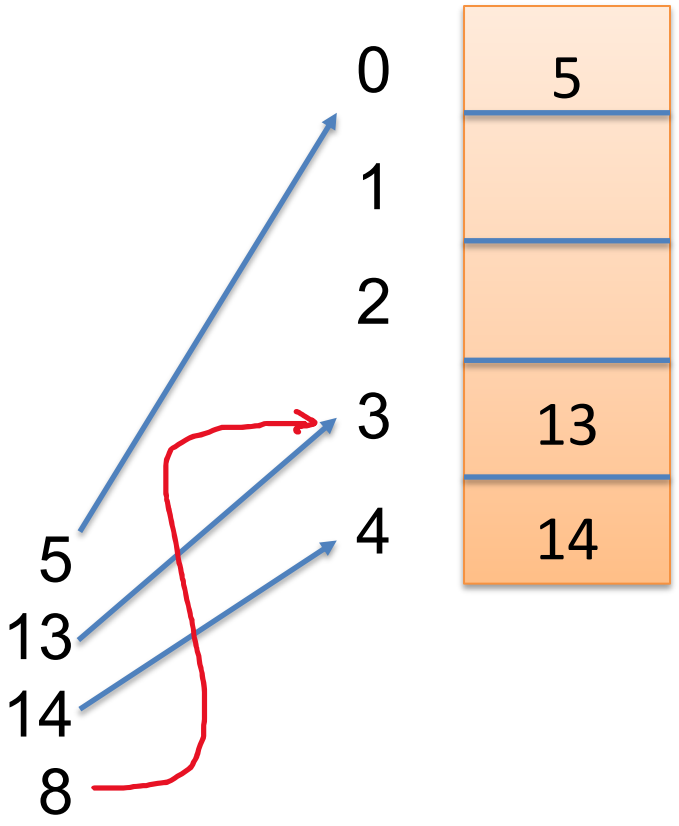
`h(x1) = h(x2)` for some `x1 ≠ x2`.

Collisions are normally unavoidable.

Method *to reduce collisions*

- using a prime number for hash table size `m`.
- making the table size `m` big (as the expense of space efficiency).

Collision happens anyway, and needs to be dealt with.



Small example of collisions.

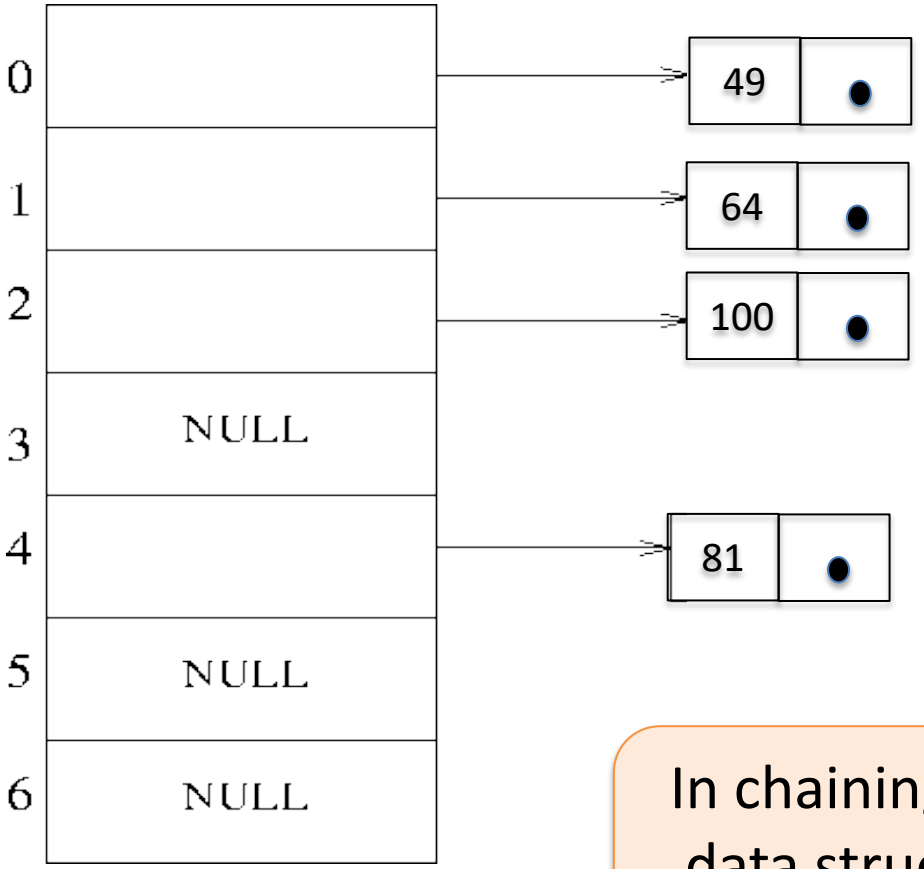
`m= 5, h(x)= x % m`

keys inserted: 5, 13, 14, 8

Collision Solution 1: Chaining (aka. Separate Chaining)

`h(x) = x % 7,` keys (entered in decreasing order in this example):

`100, 81, 64, 49, 36, 25, 16, 4,2,1,0`

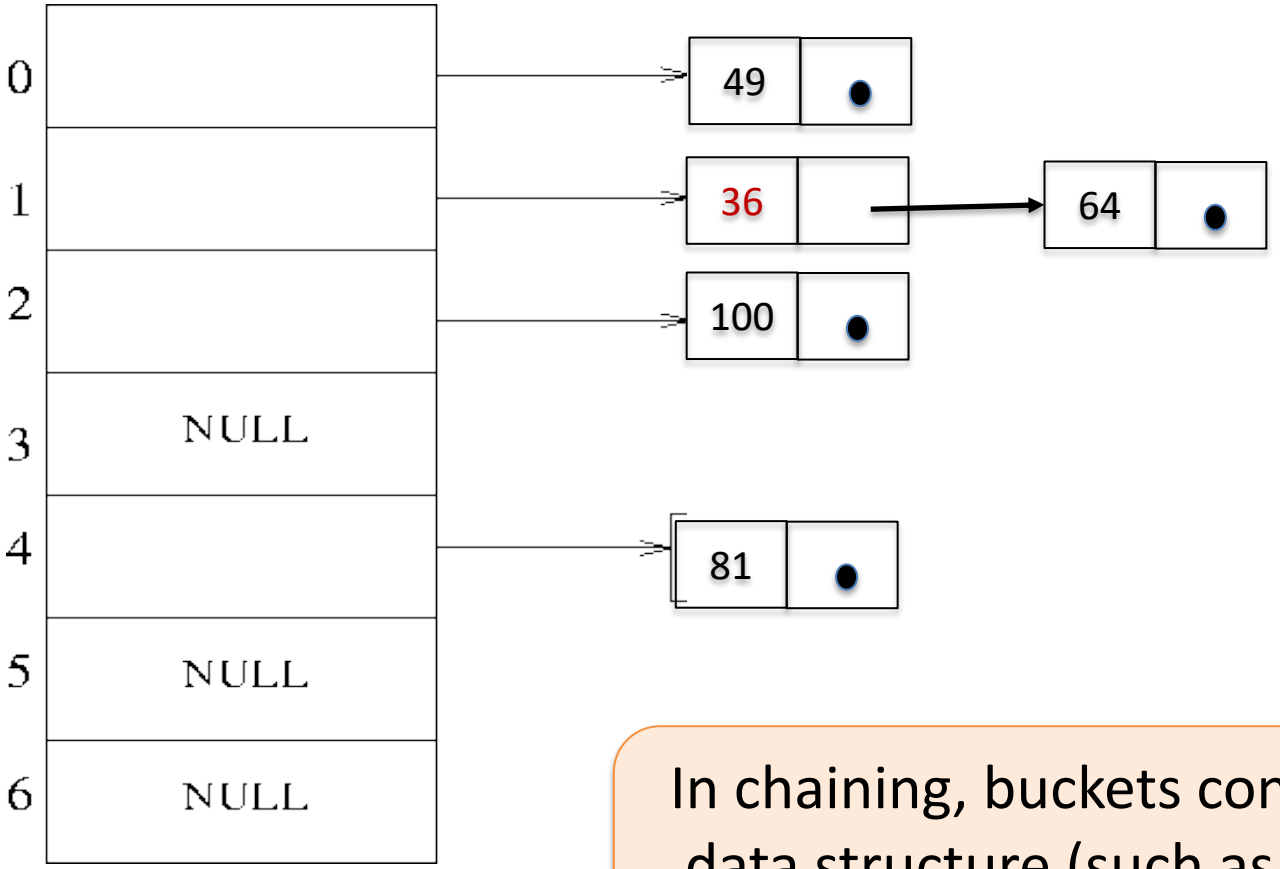


In chaining, buckets contain the link to a data structure (such as linked lists), *not the data themselves*.

Collision Solution 1: Chaining

`h(x) = x % 7,` keys (entered in decreasing order in this example):

`100, 81, 64, 49, 36, 25, 16, 4,2,1,0`

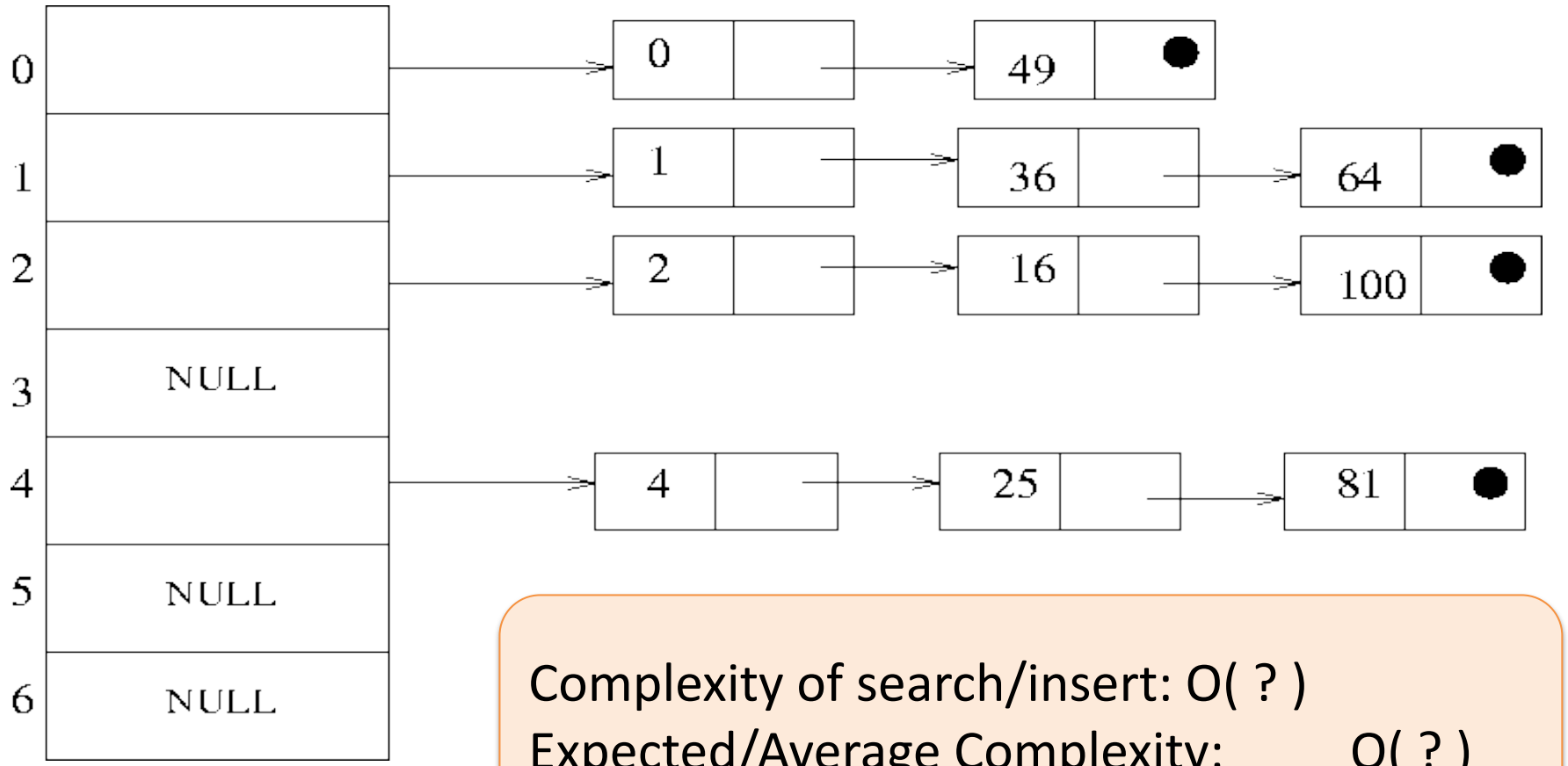


In chaining, buckets contain the link to a data structure (such as linked lists), *not the data themselves*.

Collision Solution 1: Chaining

`h(x) = x % 7,` keys entered:

`100, 81, 64, 49, 36, 25, 16, 4,2,1,0`



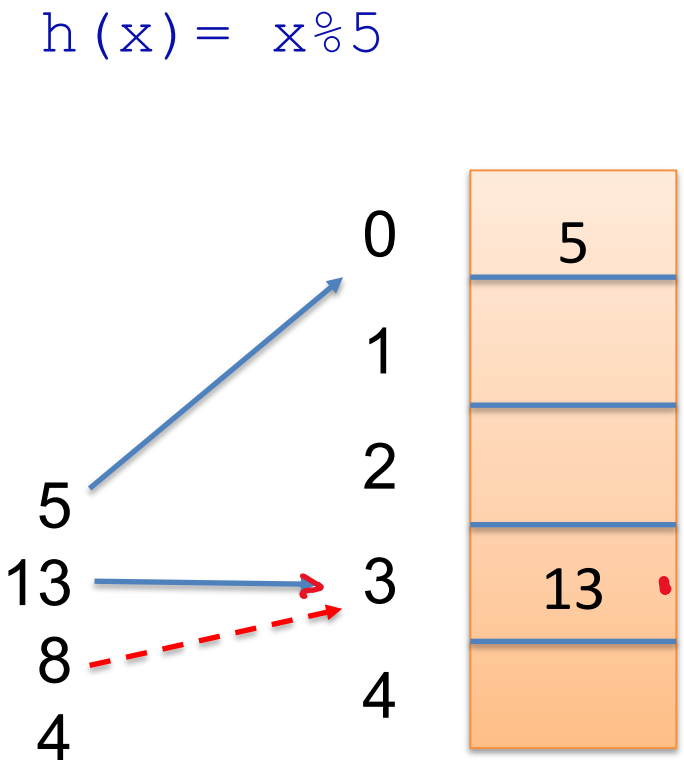
Complexity of search/insert: O(?)

Expected/Average Complexity: O(?)

Solution 2: Open Addressing
(here, *data are stored in the buckets*)

Unlike chaining, in open addressing:

- Data are stored inside the buckets
- At any point, the size of the table must be ≥ the total number of keys

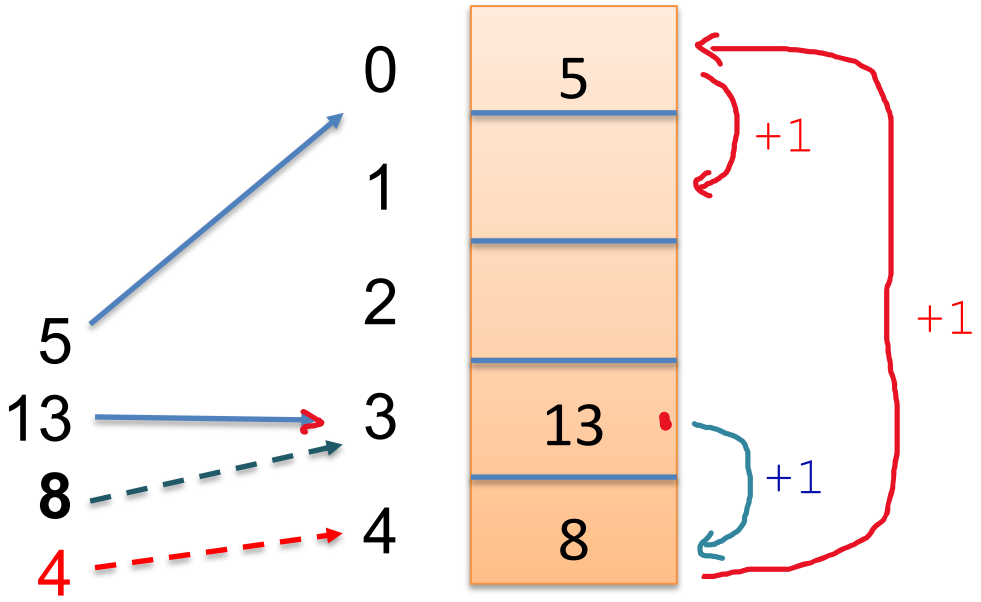


Solution 2a: Open Addressing with Linear Probing

That is, when inserting we do some *probes* until getting a vacant slot.

- Start at position h(x)
- If collided at position i, we try position (i+1)%m (and continue like that to the subsequent +1 until reaching a vacant)

h(x)= x%5



Complexity of search/insert : O(?)
Expected/Average Complexity: O(?)
Deletion is problematic (try insert 3, del 8, search 3)

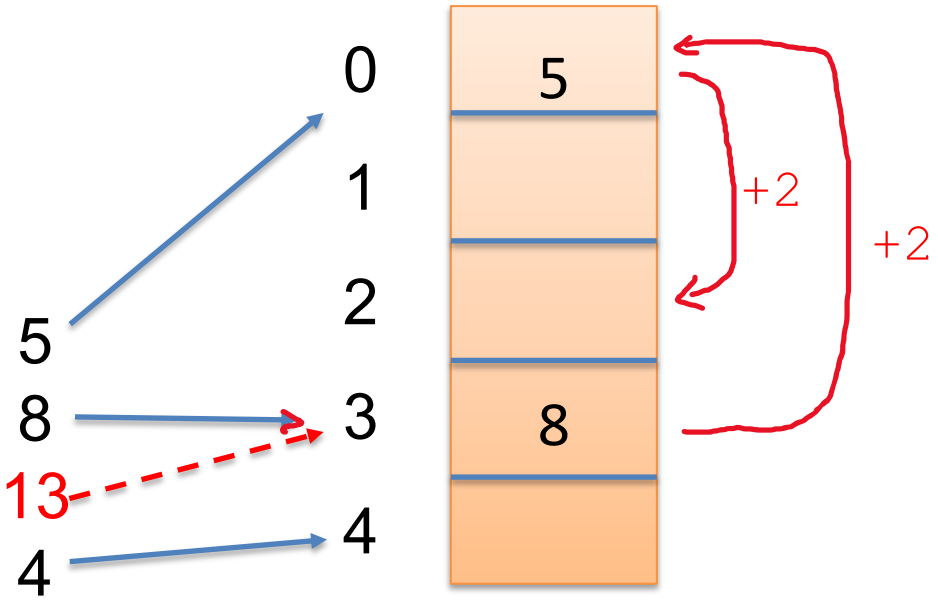
Solution 2b: Open Addressing with Double Hashing

Here, in addition to the hash function `h(x)`, we have a second hash function `h2(x)`.

- Start at position `h(x)`
- If collided at position `i`, we employ `k= h2(x)` and try position `(i+k)%m` (and continue like that with the subsequent `+k` until reaching a vacant)

`h(x)= x%5`

`h2(x)= x%3+1`



Linear Probing is a special case of Double Hashing, when h2(x)= 1

Hashing summary

Hashing= *hash table* `T` + *hash function* `h(x)`
key `x is` stored at `T[h(x)]`
To be good:

- `h(x)` must distribute key to the table evenly

Pros:

- expected O(1) for search/insert/delete

Cons:

- Worst case O(n) for search/insert/delete
- Designing evenly-distributed hash function is tricky

Practice:

- Understanding linear probing & double hashing with `W6.2`
- Programming with `W6.3`:
  - hash table framework implemented
  - just implement 3 functions in `hashT.c`

Programming Notes:

- functions `insertLP` and `insertDH5` are actually invoked in function `insert`, where the application of the principal hash function already done
- so, the `key` in `insertLP`, `insertDH5, insertDH` is actually the first mapping position, not the original key (a different name like `firstHashPosition` would be better than `key`)

- [here or at home:] use W6.3 to learn more about function pointers

Assignment 2: General Information

- Being developed further from ASS1 code
- You can use your ASS1 code, or the supplied solution, or even your friend‘s ASS1 code
- Remember to make references if you used ASS1 code (even if it’s yours)

- You don’t have to keep dict1 for ASS2, you just need to have dict2 and dict3.
- But it’s OK to keep, and it’s might be better to keep!

ASS2: Q&A

Work Smarter!

How?

What to keep in the array? the tree?

What does the empty tree look like?

For each DS (array, tree):

How to do the insert:

- what to insert?
- what the function header looks like?
- what happen when insert key already exists?

How to do the search:

- what the function header looks like
- is there a NOTFOUND case
- what are required output

Requirements (Code)

- modularity, but not strict on object-oriented
- extensible to multiple dictionaries
- Space Efficiency:
  - numbers should be stored as int or double
  - strings with exact size
  - index should not contain unneeded data
- Time efficiency should be reasonable:
  - Time = Time for processing 2 dictionaries

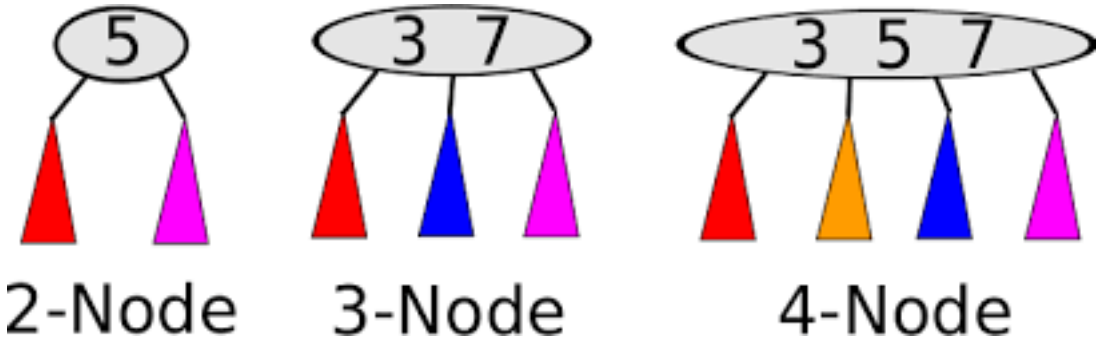
Requirements (Report)

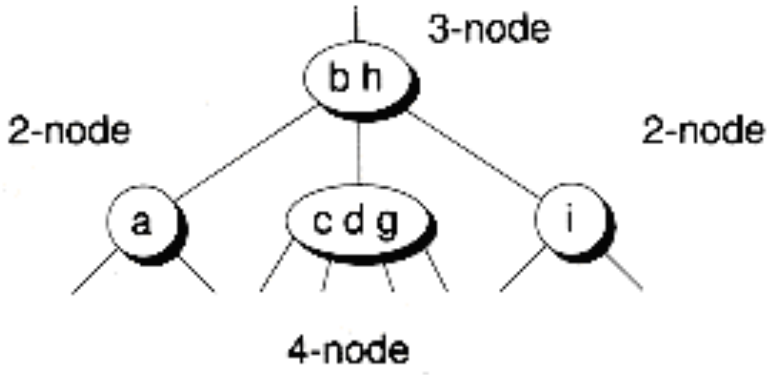
- ? perhaps next week
- what to consider when writing the code?

Non-examined Materials

2-3-4 Tree [a self-balancing search tree]

Each node might have 1, 2 or 3 data, and 2, 3, or 4 pointers to children, respectively

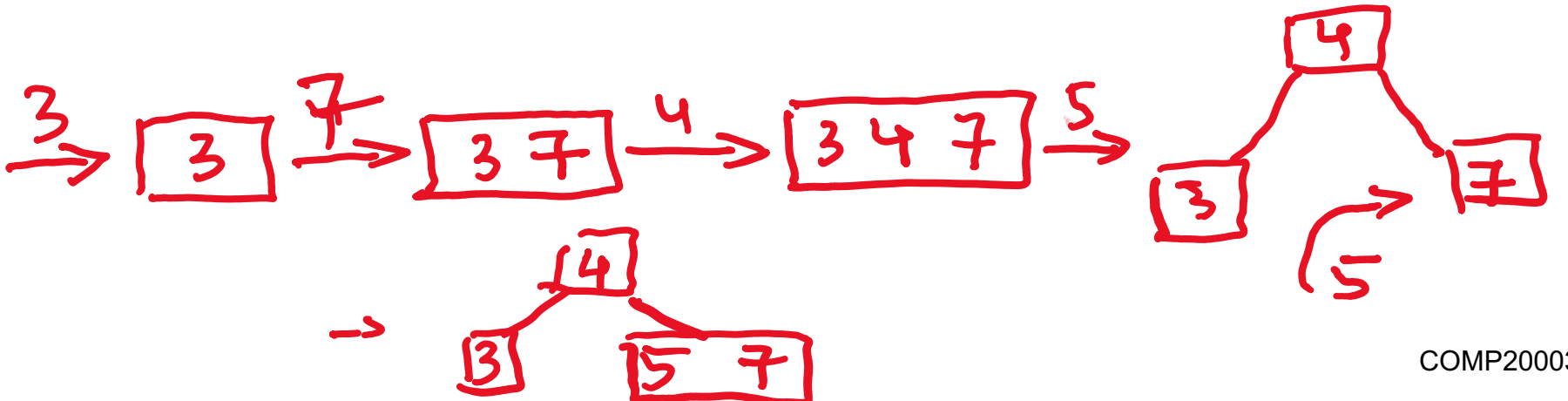




2-3-4 Insertion: inserts a key x into a non-empty tree

- from root, walks down by comparing x with keys in nodes until arriving to a *leaf node* (ie. *node with no children*)
- if the leaf is not full (ie. <3 key), insert x into the leaf, otherwise:
  - splits the leaf by promoting the middle key to be the parent of 2 splitted leaves
  - then, insert x into the appropriate one of the 2 new leaves

Example of splitting leaf: insert `i`nto an empty 2-3-4 tree: `3 7 4 5`



Example: insert EXAMPLETRES into an empty tree

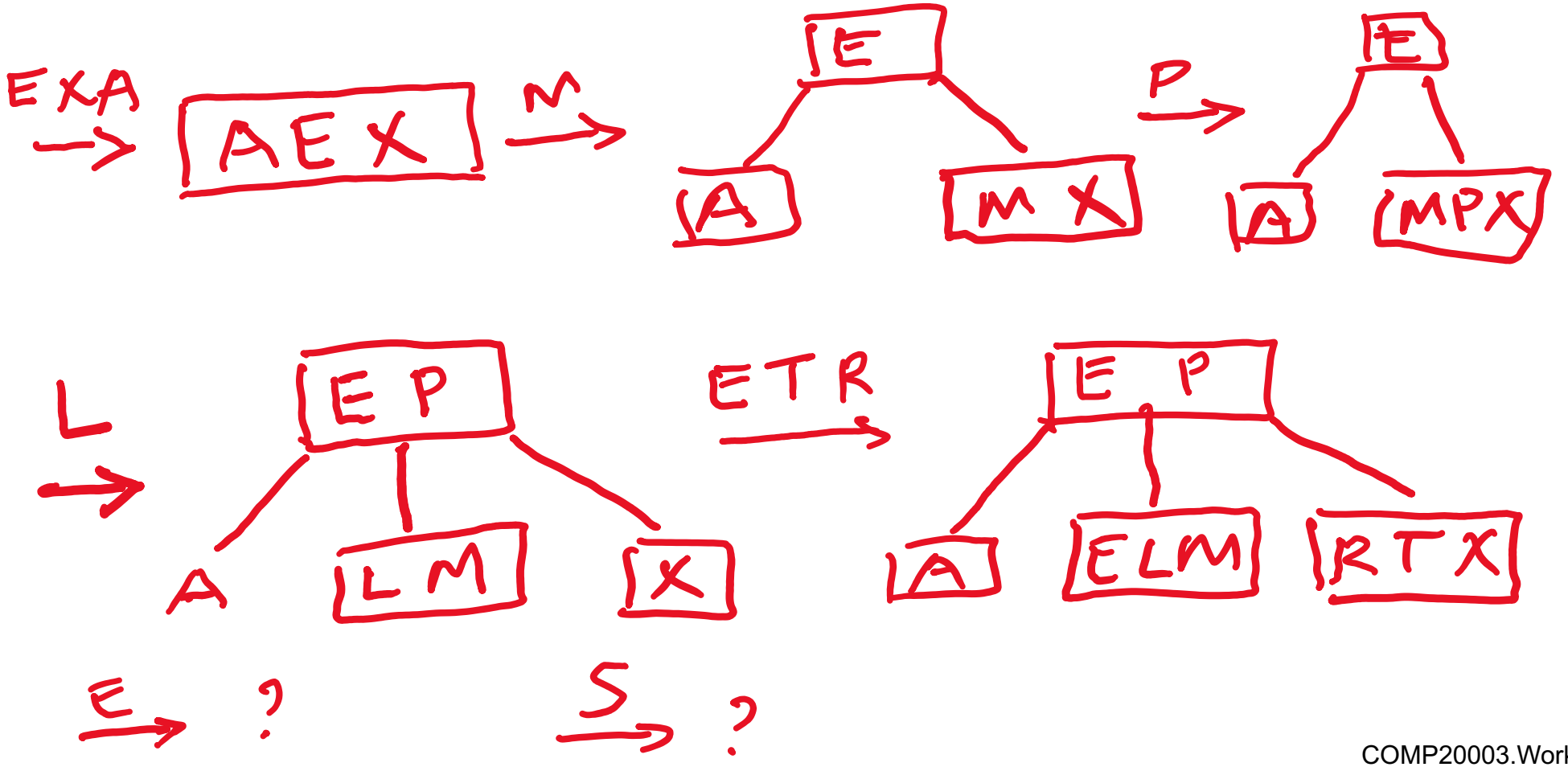
Supposing:

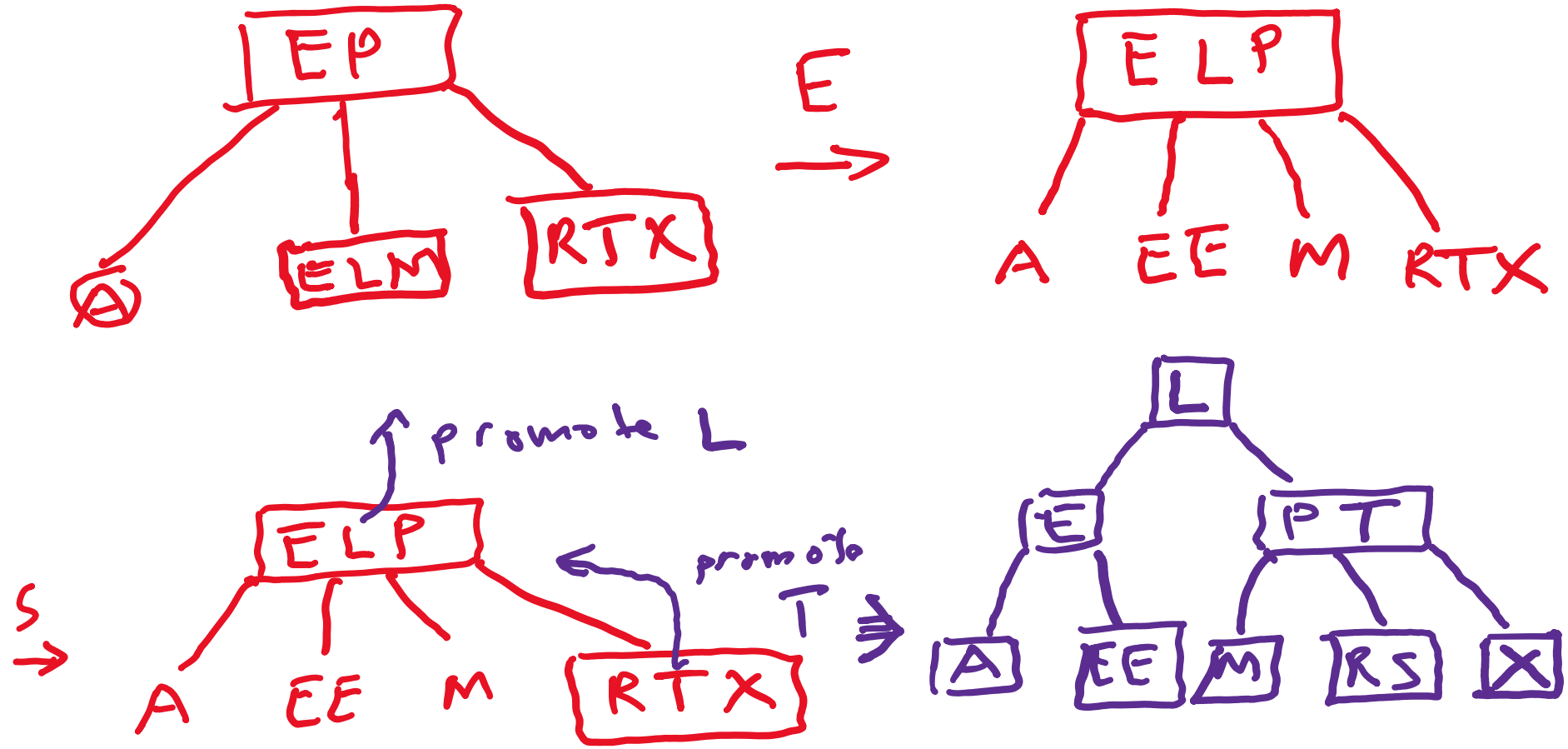
- an equal key will be placed in the right children

Example: insert EXAMPLETRES into an empty tree

Supposing:

- an equal key will be placed in the right children




EP
ELM
RTX
E
ELP
A
EE
M
RTX
promote L
ELP
S
promote T
A
EE
M
RTX
L
E
PT
A
EE
M
RS
X

2-3-4 Tree: Time Complexity

The height of 234 tree= ?

Insert

- O(?)

Lookup

- O(?)

2-3-4 Tree: Time Complexity

The height of 234 tree= ?

$\log_4 n \leq \text{height} \leq \log_2 n$

so, height is O(logn)

Insert

- Θ(log n)

Lookup

- O(log n)

Still wondering about hashing and/or 2-3-4 trees?

Revisit lecture and/or lecture slide

See a detailed workshop .pdf in W6.1.