1	Hashing
2	Modular Programming in ASS1
3	Assignment 2: Introduction
4	LAB: - Implementing hashtables - Questions on Ass1 & Ass1 feedbacks (if available) - Ass2 Q&A
5	Non-examined Materials: 2-3-4 Trees ***k-D(2-D) Trees

Hashing: Introduction

Task: Build a dictionary where keys are unique and in the range of 0..799. Insert and Search are major operations.

Q: What concrete data type is the best?

Hashing: Introduction

Task: Build a dictionary where keys are unique and in the range of 0..799. Insert and Search are major operations.

Q: What concrete data type is the best?

Q: What if the keys are in range 1200..1999, or 0..1600?

Concepts: hash function, hash table, bucket, collision

Hashing

Hashing= hash tables + hash functions

Hash table is an array of `m` *buckets*.

Hash function `h` is to map key `x` to `h(x)`

`h(x)=` index into the hash table,

ie. mapping `x` to the bucket where `x` will be likely stored.

Example: `m= 7`, `h(x) = x%m`

Potentially, hashing gives us a dictionary with O(1) for both insertion and search!

Collisions

`h(x1) = h(x2)` for some `x1` $\neq$ `x2`.

Collisions are normally unavoidable.

One method *to reduce collisions* using a prime number for hash table size `m`. (remember the lecture on this point?)

Another method is to make the table size `m` big enough (but that affects space efficiency).

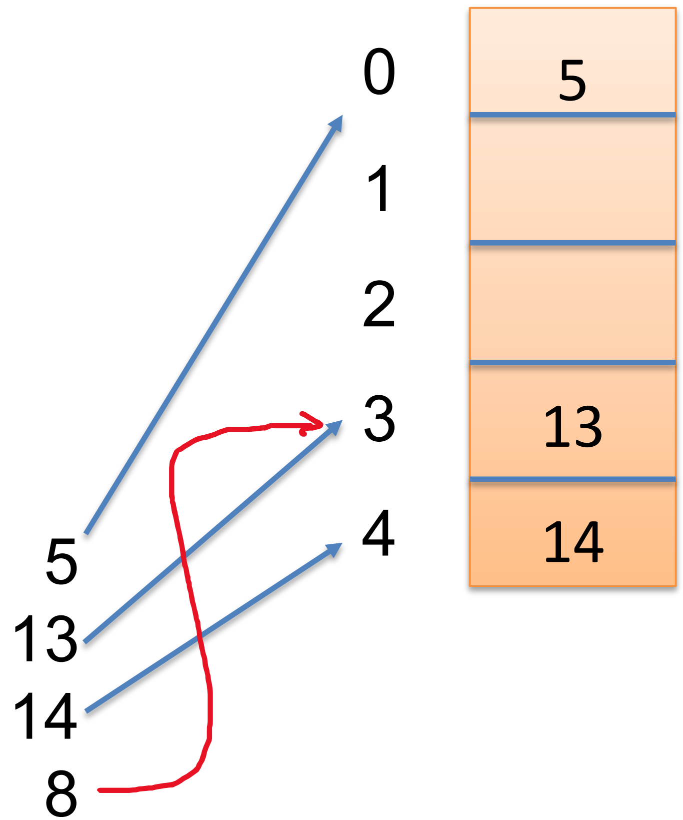
Colisions

`h(x1) = h(x2)` for some `x1≠ x2`.

Example:

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

Here: `h(8)= h(5)`



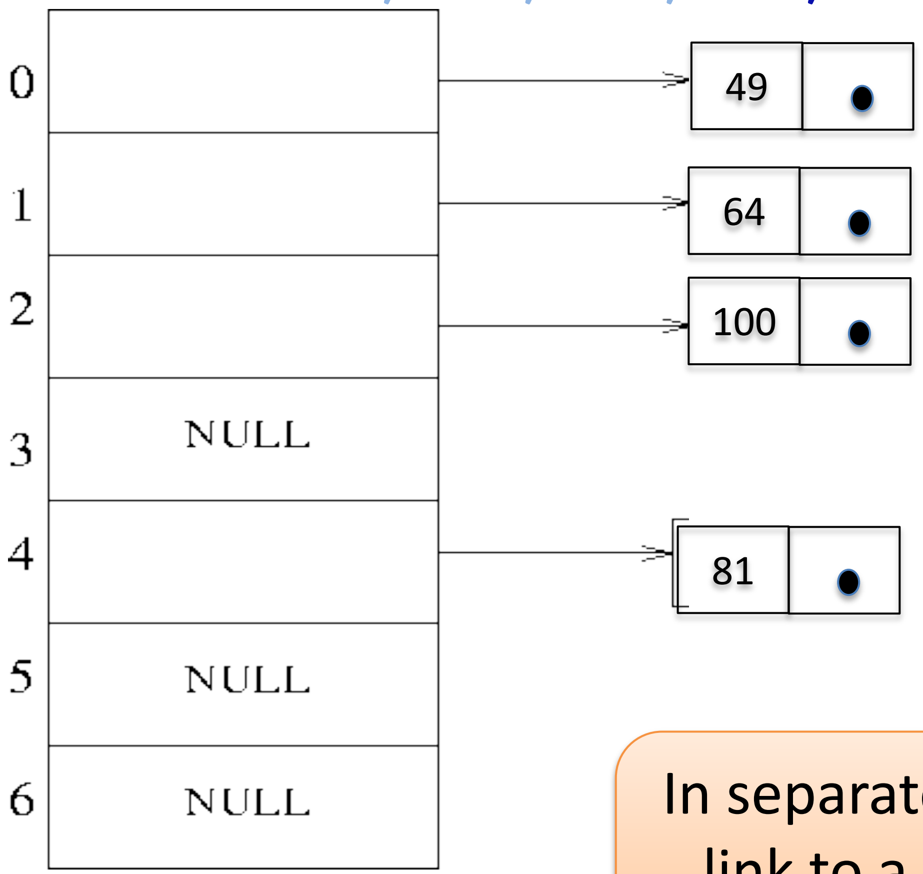
One method *to **reduce** collisions* using a prime number for hash table size `m`. (remember the lecture on this point?)

Another method is to make the table size `m` big enough (but that affects space efficiency).

Collision Solution 1: 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`

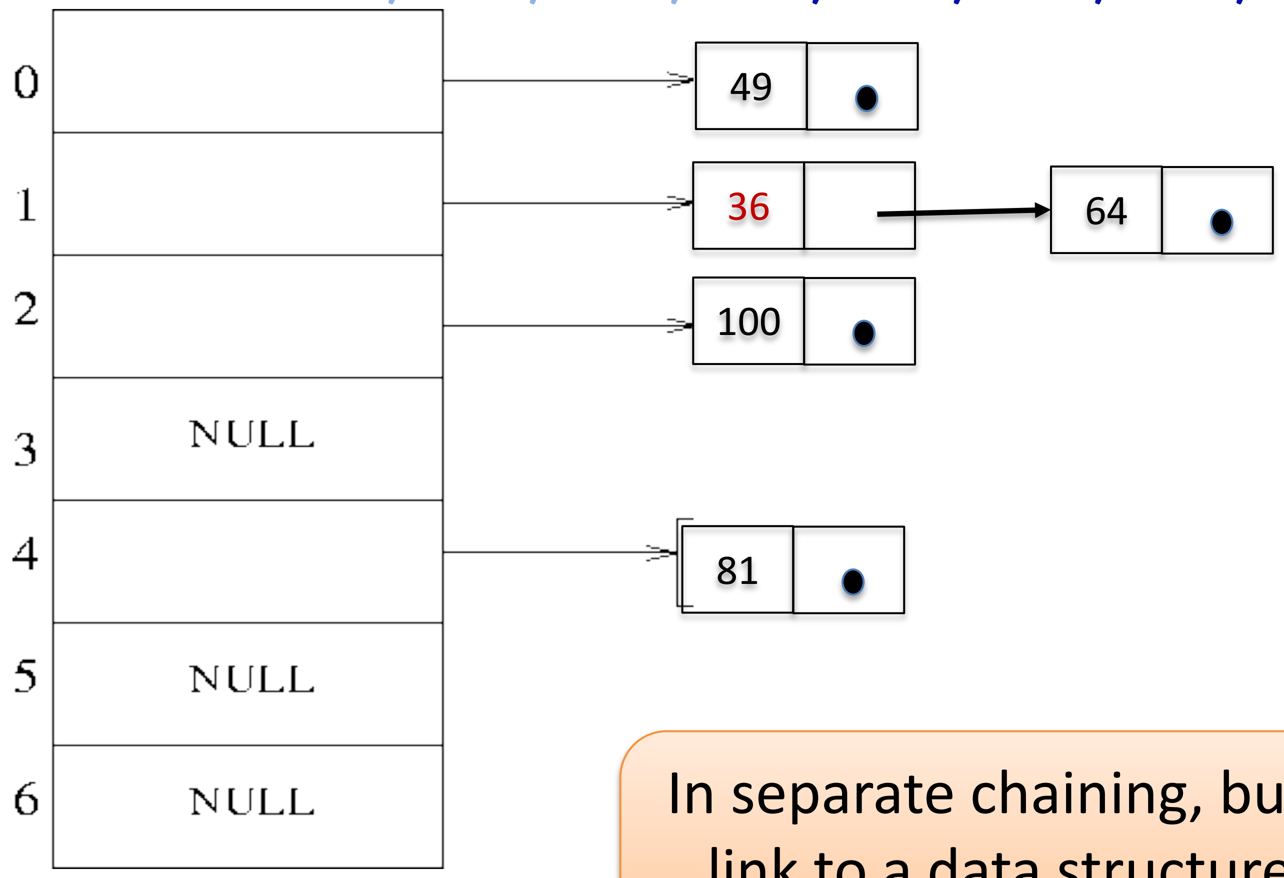


In separate chaining, buckets contain the link to a data structure (such as linked lists), *not the data themselves*.

Collision Solution 1: 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`

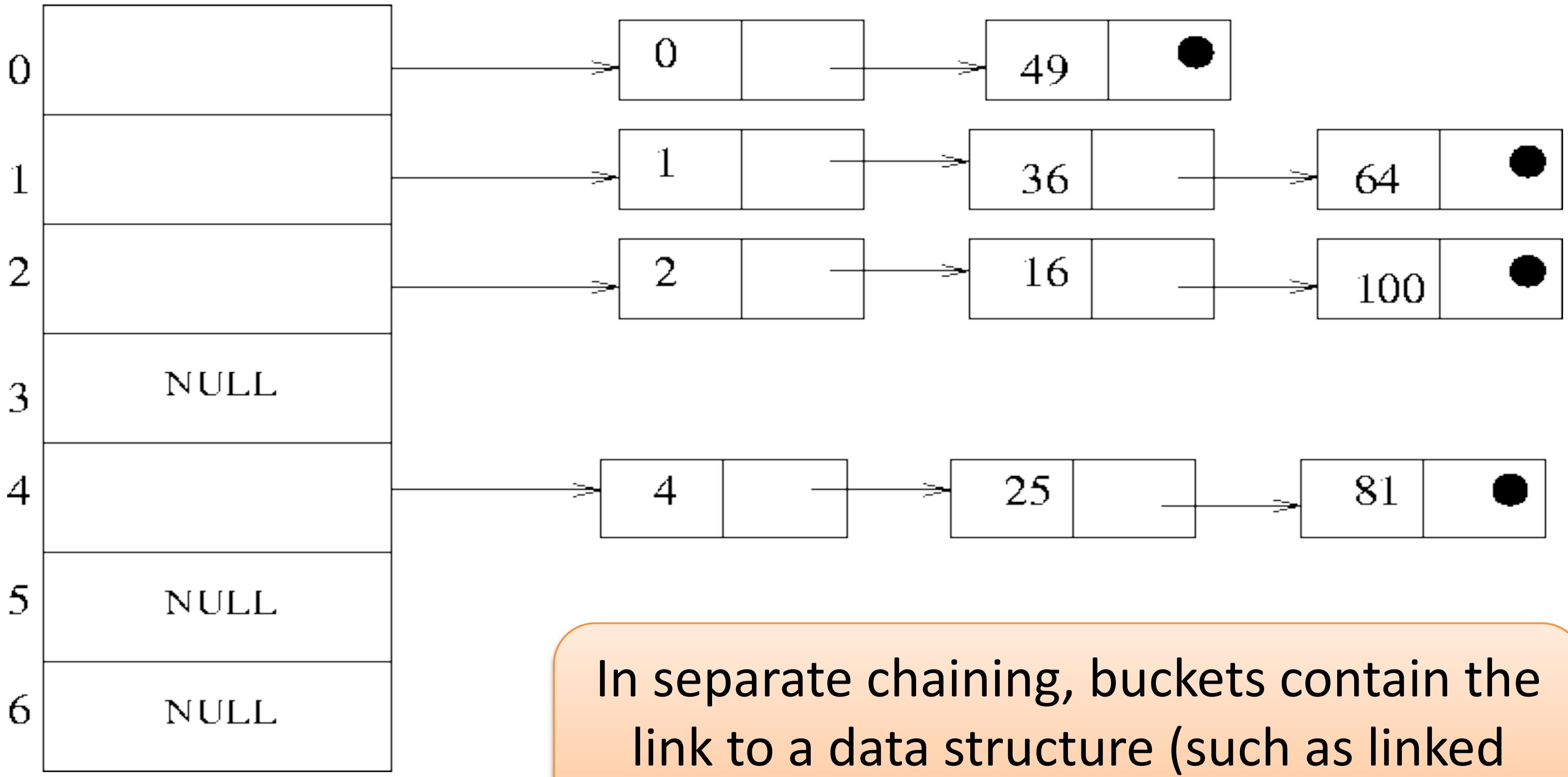


In separate chaining, buckets contain the link to a data structure (such as linked lists), *not the data themselves*.

Collision Solution 1: Separate Chaining

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

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

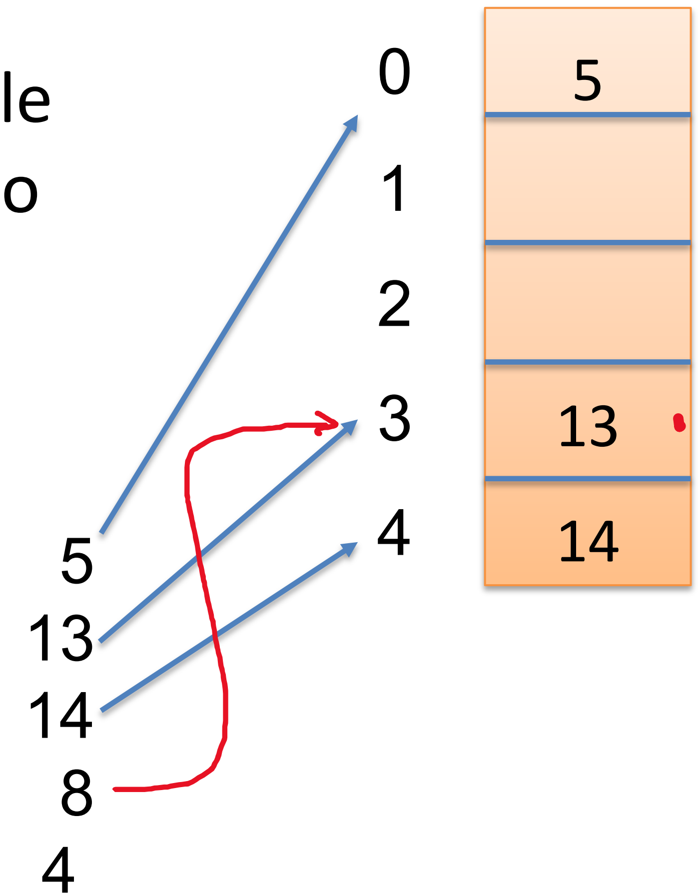


In separate chaining, buckets contain the link to a data structure (such as linked lists), *not the data themselves*.

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

Unlike separate chaining:

- Data are stored inside the buckets
- At any point, the size of the table must be greater than or equal to the total number of keys

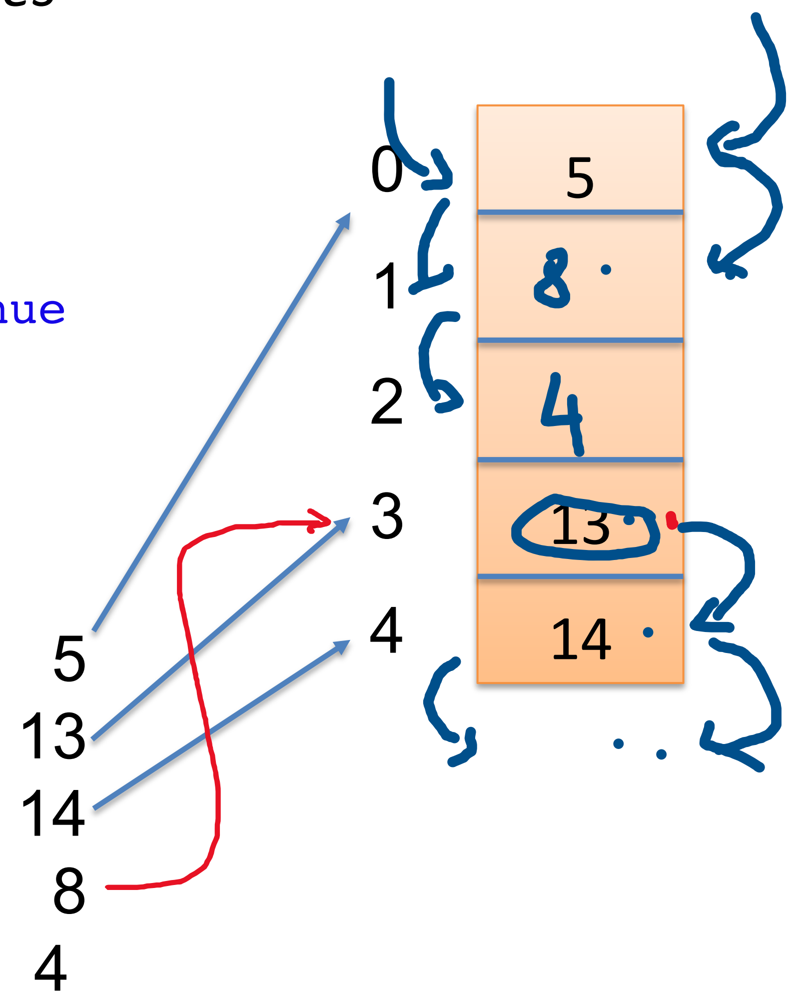


That is, when inserting we do some *probes* until getting a vacant slot.

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

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

and inserting



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 try position (i+h2(x))%m (and continue like that until reaching a vacant)`

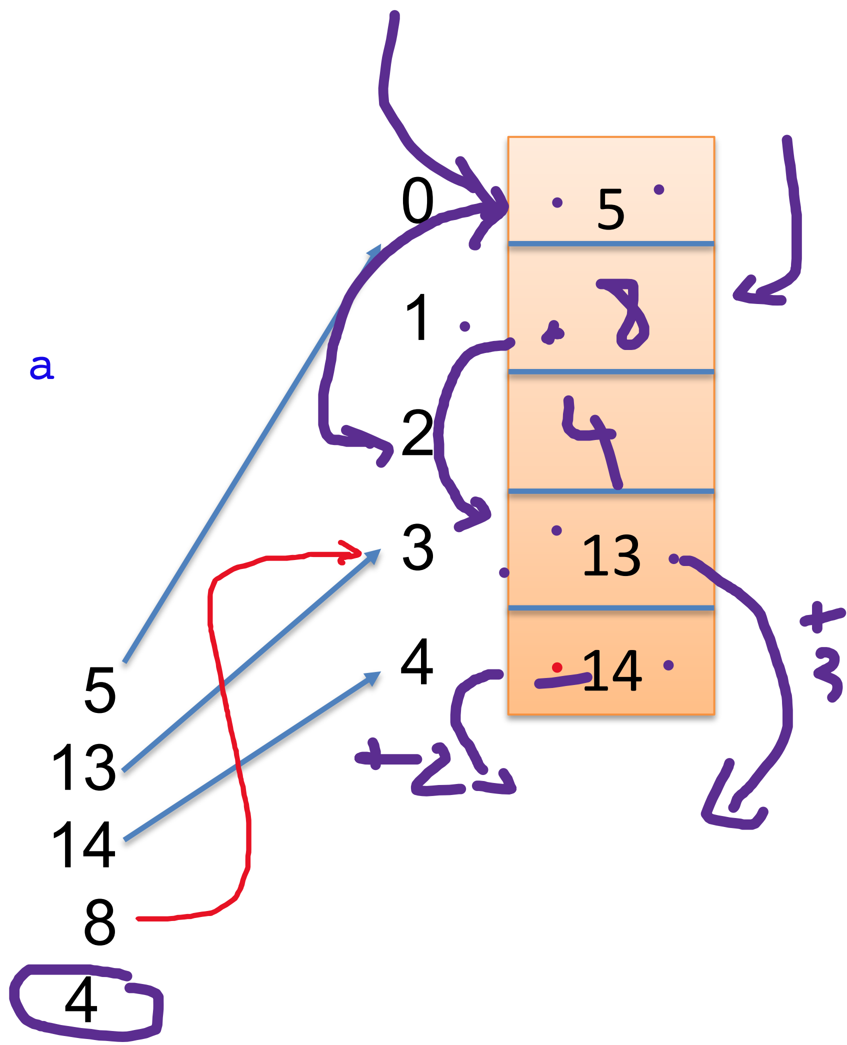
Example: `m=5, h(x)= x mod m,`

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

and inserting

2

$h2(8) = 3$



Double hashing summary

`In addition to h(x), use a second hashing function h2(x):`

- `Start at position h(x)`
- `If collided at position i, we try position i+h2(x) (and continue like that until reaching a vacant)`

Note that:

`h2(x)` $\neq$ `0 (must be!) for all x`,

to be good, `h2(x)` should be co-prime with `m` (see example in lecture),

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

Example (Lesson→Quiz→1)

You are given a hash table of size 13 and a hash function hash(key) = key % 13. Insert the following keys in the table, one-by-one, using linear probing for collision resolution:

14, 30, 17, 55, 31, 29, 16

index:	0	1	2	3	4	5	6	7	8	9	10	11	12
hash table		**14**		**55**	**30**	**17**	**31**	**29**	**16**				

Example double hashing(Lesson→Quiz→2)

Keys to insert: 14, 30, 17, 55, 31, 29, 16

Now insert the same keys into an (initially empty) table of the same size (13), using double hashing for collision resolution, with hash2(key) = (key % 5) + 1

h2(29)= 5 h(31)= 5

h(16)=3 → h2(16)=2 3 3+2 +2+2

0	1	2	3	4	5	6	7	8	9	10	11	12
	14		**55**	**30**	**31**		**17**	**29**	**16**			

Quiz 1

What is the big-O complexity to search for an element in a hash table if there are no collisions?

A. *O(1).*

B. O(n).

C. O(n^2).

D. O(*log n)*

Quiz 2

What is the big-O complexity to search for an element in a hash table?

A. *O(1).*

B. $O(n)$.

C. $O(n^2)$.

D. $O(\log n)$

Quiz 3

The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function `h(k) = k mod 10` and linear probing. What could be the resultant hash table?

0	
1	
2	2
3	23
4	
5	15
6	
7	
8	18
9	

(A)

0	
1	
2	12
3	13
4	
5	5
6	
7	
8	18
9	

(B)

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

(C)

0	
1	
2	12, 2
3	13, 3, 23
4	
5	5, 15
6	
7	
8	18
9	

(D)

Quiz 4

The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using separate chaining with hash function `h(k) = k mod 10`. What could be the resultant hash table?

0	
1	
2	2
3	23
4	
5	15
6	
7	
8	18
9	

(A)

0	
1	
2	12
3	13
4	
5	5
6	
7	
8	18
9	

(B)

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

(C)

0	
1	
2	12, 2
3	13, 3, 23
4	
5	5, 15
6	
7	
8	18
9	

(D)

Prep. for ASS2

- ASS1 review on Modular Programming
- Initial information/preparation for ASS2

Remember:

- Start ASS2 ASAP
- Seek help early
- You can also check WorkSpace → base voronoi2.c

  I might put more on that workspace if receive some demands by this Saturday.

Assignment 1: Modular Programming

A simple module has 2 files: one header file .h and one (or more) implementation file .c.

A module normally supply all facilities to declare and manipulate a single datatype, or a collection of mutually-related datatypes.

Assignment 1: Modular Programming

Suggested Modules: `tower`, `dcel`, `utils`

Example Module for ASS1:
`tower`: datatypes and function for working with towers, including:

Datatype: `typedef … tower_t;`

Reading csv file:
- `tower_t **read_tower(tower_t **T, int *n, int *size, char *fname)`, or
- `read_tower(tower_t ***pT, int **n, int *size, char *fname)`

Print a tower:
`print_tower(tower_t *t, FILE *f)`

simple `Makefile` has the format:

```
all: voronoi1

voronoi1: voronoi1.o dcel.o tower.o utils.o
    gcc -Wall –o voronoi1 voronoi1.o dcel.o tower.o utils.o

dcel.o: dcel.h dcel.c
    gcc –Wall –g –c dcel.c
...

clean:
    rm –f *.o voronoi1
```

Makefile: a better and *easier* version

```
CC = gcc
CFLAGS = -Wall -g
HDR = dcel.h tower.h utils.h
SRC = voronoi1.c dcel.c tower.c utils.c
EXE = voronoi2
OBJ = $(SRC: .c=.o)

...
all: $(EXE)
$(EXE): $(HDR) $(OBJ)
	$(CC) $(CFLAGS) –o $(EXE) $(OBJ)

clean:
	rm –f $(EXE) *.o

$(OBJ): $(HDR)
```

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 [with acknowledgment)
- Having 4 small tasks
- BAD NEWS: you probably can get the easy 4 marks of stage 4 only after finishing the hard 7 marks of stage 3

Ass2 (Voronoi Diagram). The Main Task

The Task: Computing the Voronoi diagram iteratively.

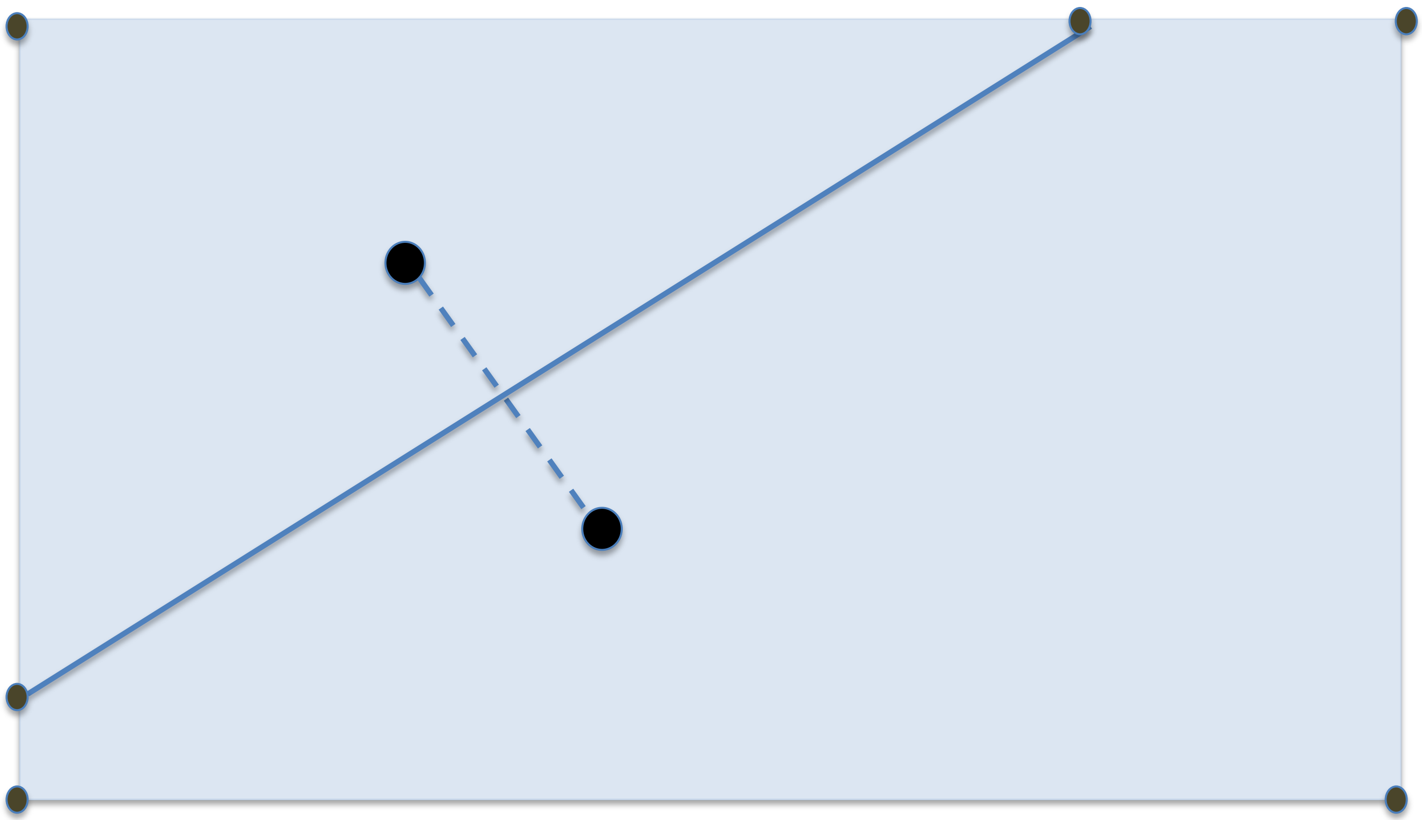
Input: filenames in argv[]

- A region file for the region's boundary (= the initial polygon in assignment 1)
- A watchtower: here, important data are the list of 2D points

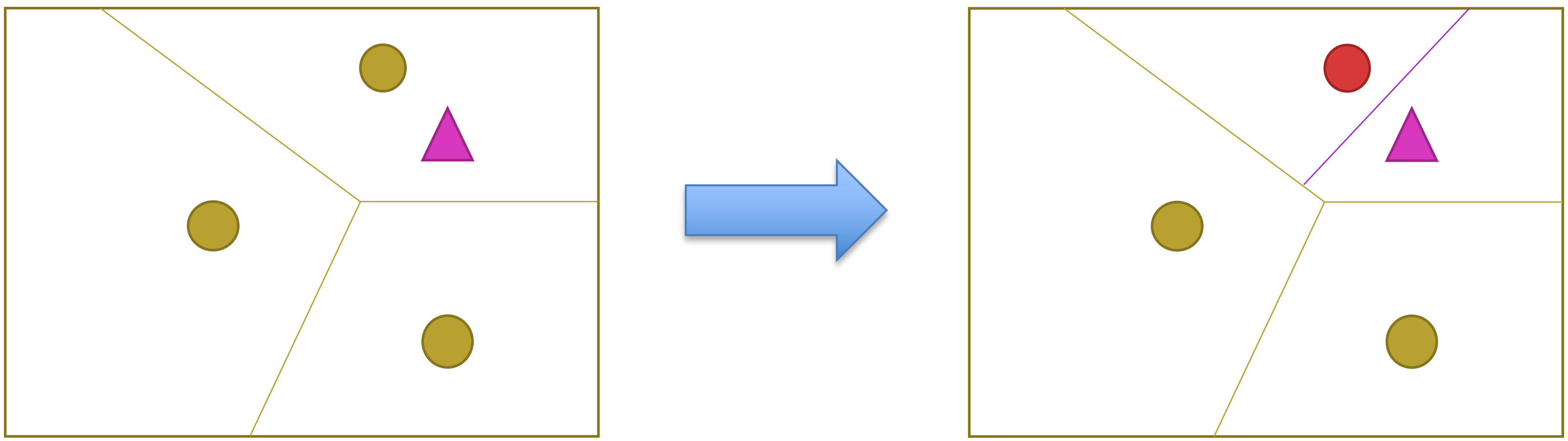
Main Task [7 marks + 2 marks for the follow-up stage]

- After processing the region file, we have the first face. That face and the first watchtower forms the first Voronoi cell
- From the second input point, each will add one Voronoi cell (ie. one face) to the space.

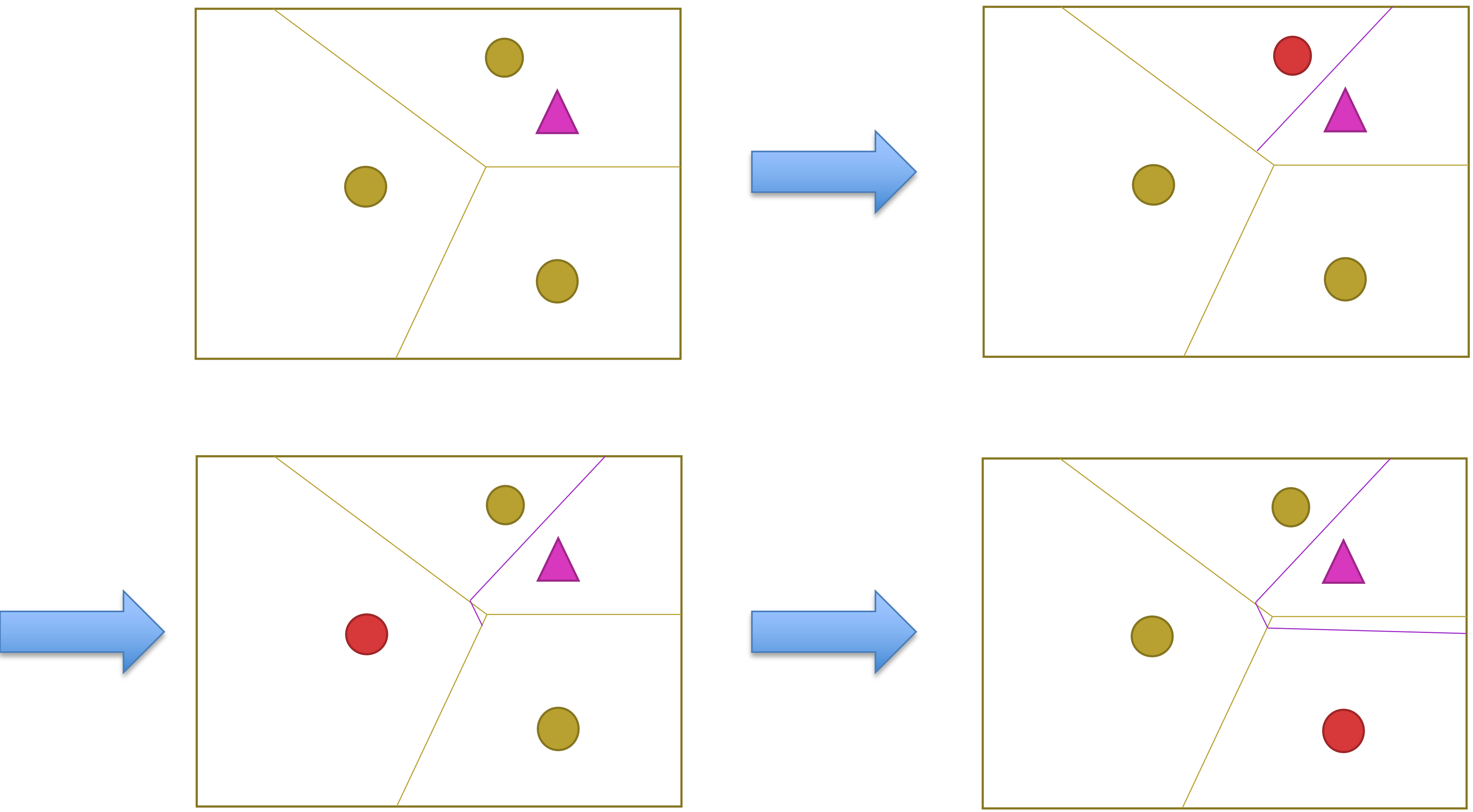
Region, points, and bisector. Computing the bisector of 2 points



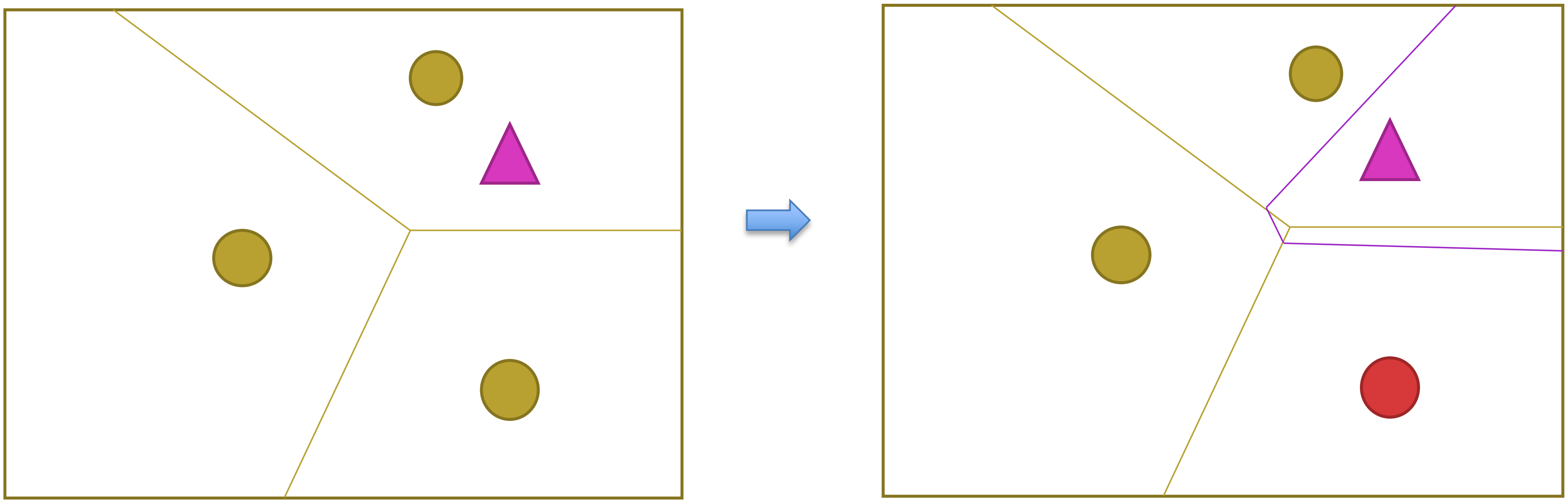
Adding new points (fig fom Grady's slides)



Adding new points (fig fom Grady's slides)



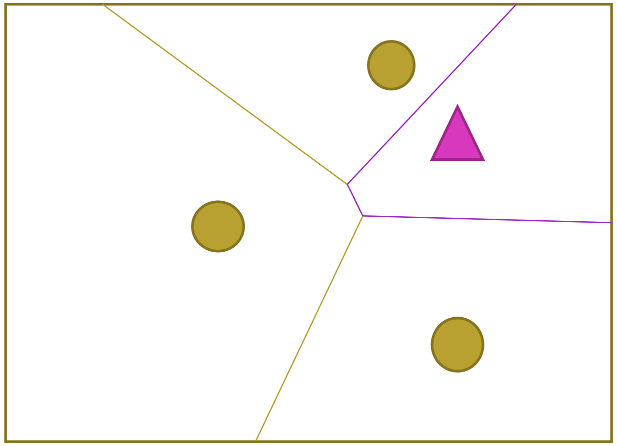
Adding new points (fig fom Grady's slides)



- You can start to think about the steps for adding a new points (draw a more complicated situation, when the cell being added to share edges with more than 2 cells).
- The spec will briefly introduce the steps

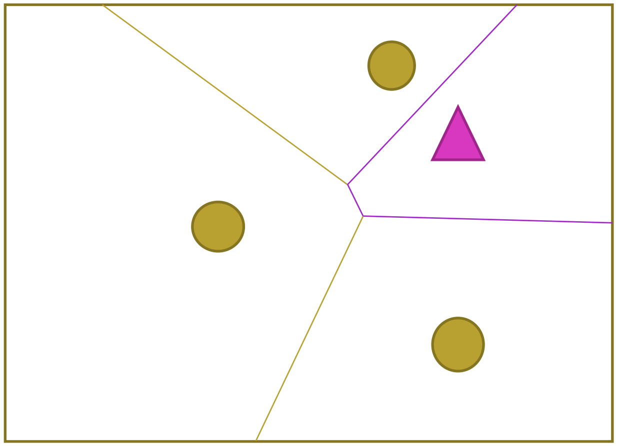
The tasks, as currently introduced in Grady's slides

- Assignment 4 parts
  - Calculate Bisectors
  - Intersection edge + points (code given)
  - Incrementally Construct Voronoi Diagram
  - Sort Faces by Largest Distance Between Vertices in Face



More details, as currently provided by Grady

- Incrementally Construct Voronoi Diagram
  - Find Face (Assignment 1, provided)
  - Add Bisector
  - Find Intersection (Provided)
  - Perform Split (Assignment 1, provided)
  - Set Edges Inside New Face to -1
  - Find furthest vertices (diameter) by looping through all pairs



Grady's Slides

pp 4-7 : example insert 14,30,17,55,31,29,16 into hashtable with h(x)= x%13

pp8 collision

pp9-13: chaining, continue 14,30,17,55,31,29,16

pp14-25 : linear probing for the above 14,30,17,55,31,29,16

pp26-37 : doiuble hashinh h2(x) (x%5)+1 for 14,30,17,55,31,29,16

pp 38- 70: step-by-step 234 tree example on 1 2 3 4 5 6 7 7 7 8

71-74 efficiency

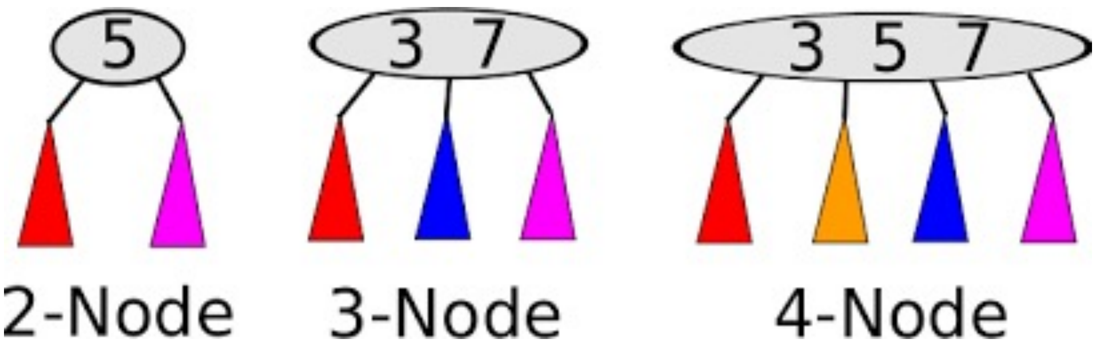
75: B++ trees

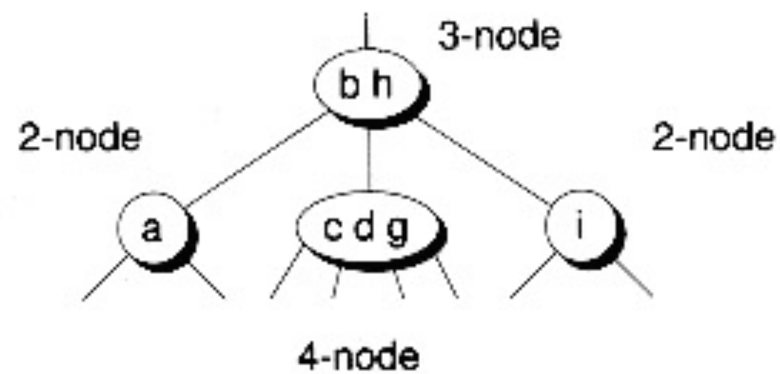
76-77 Incremental Voronoi Construction

Non-examine 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

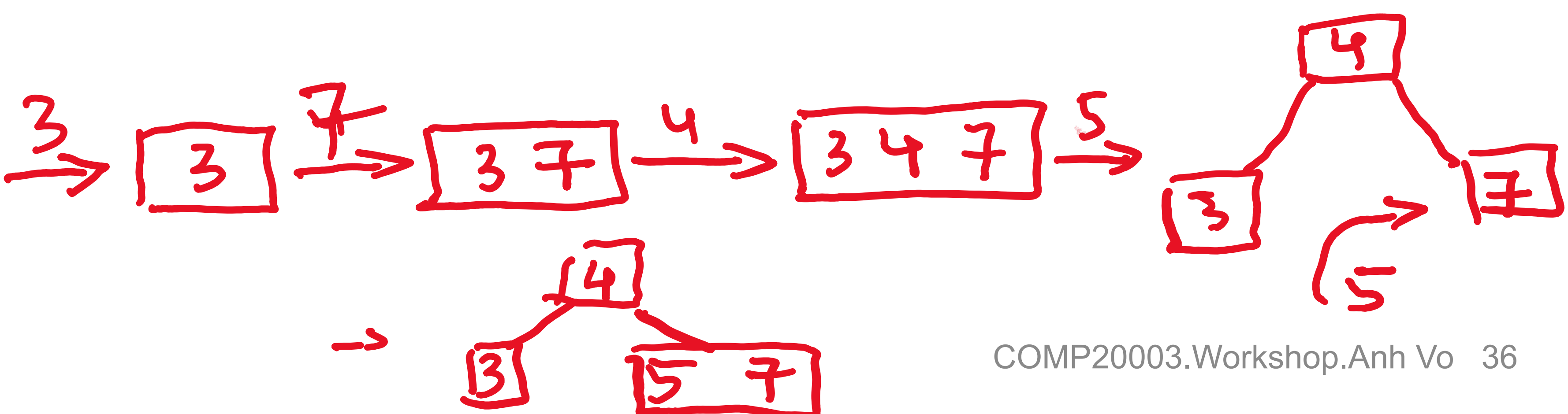




2-3-4 Insertion: inserts a key x into a non-empty tree

- 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 into an empty 2-3-4 tree: `3 7 4 5`



Example: insert `EXAMPLETRES` into an empty tree

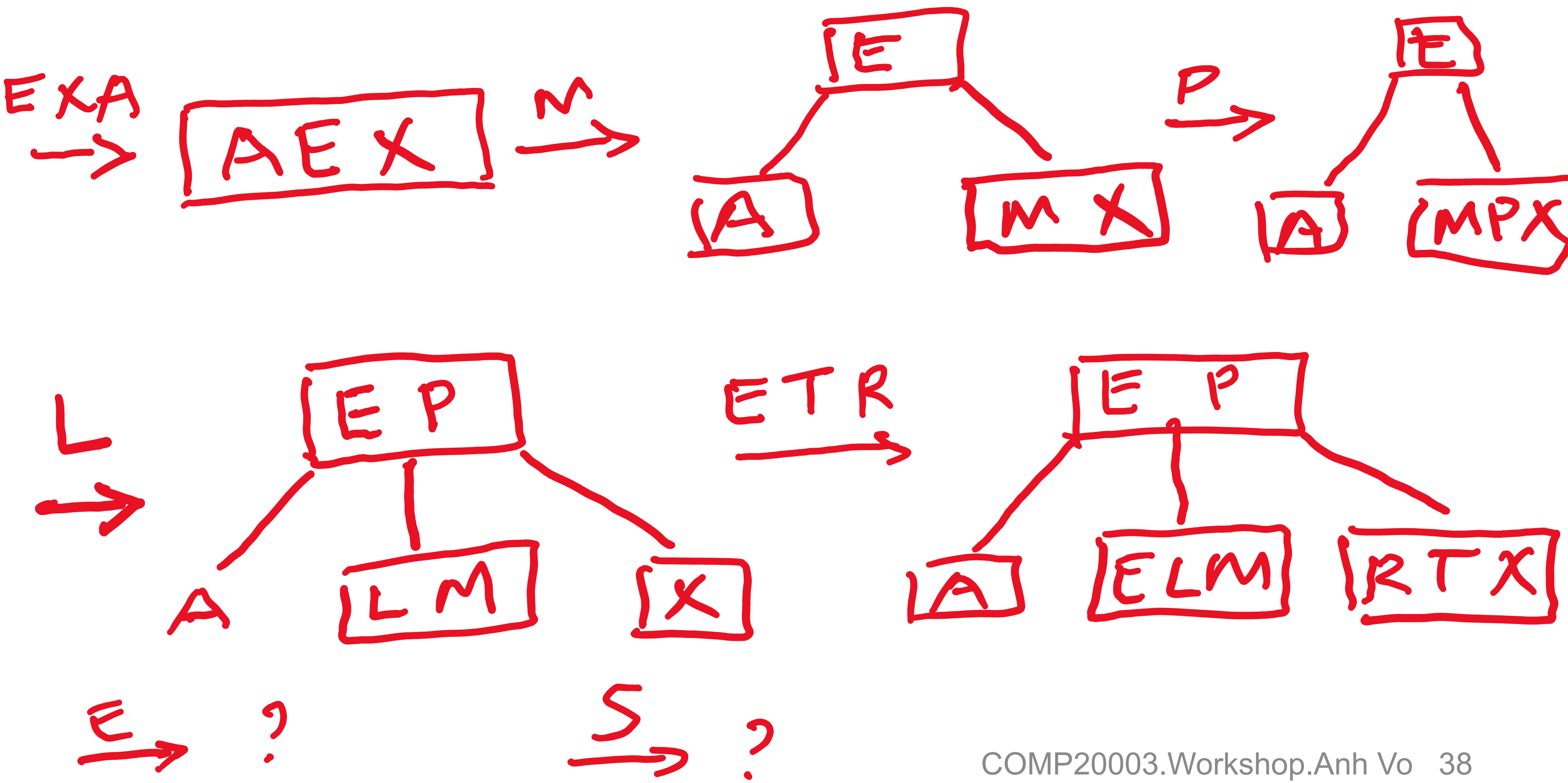
Supposing:

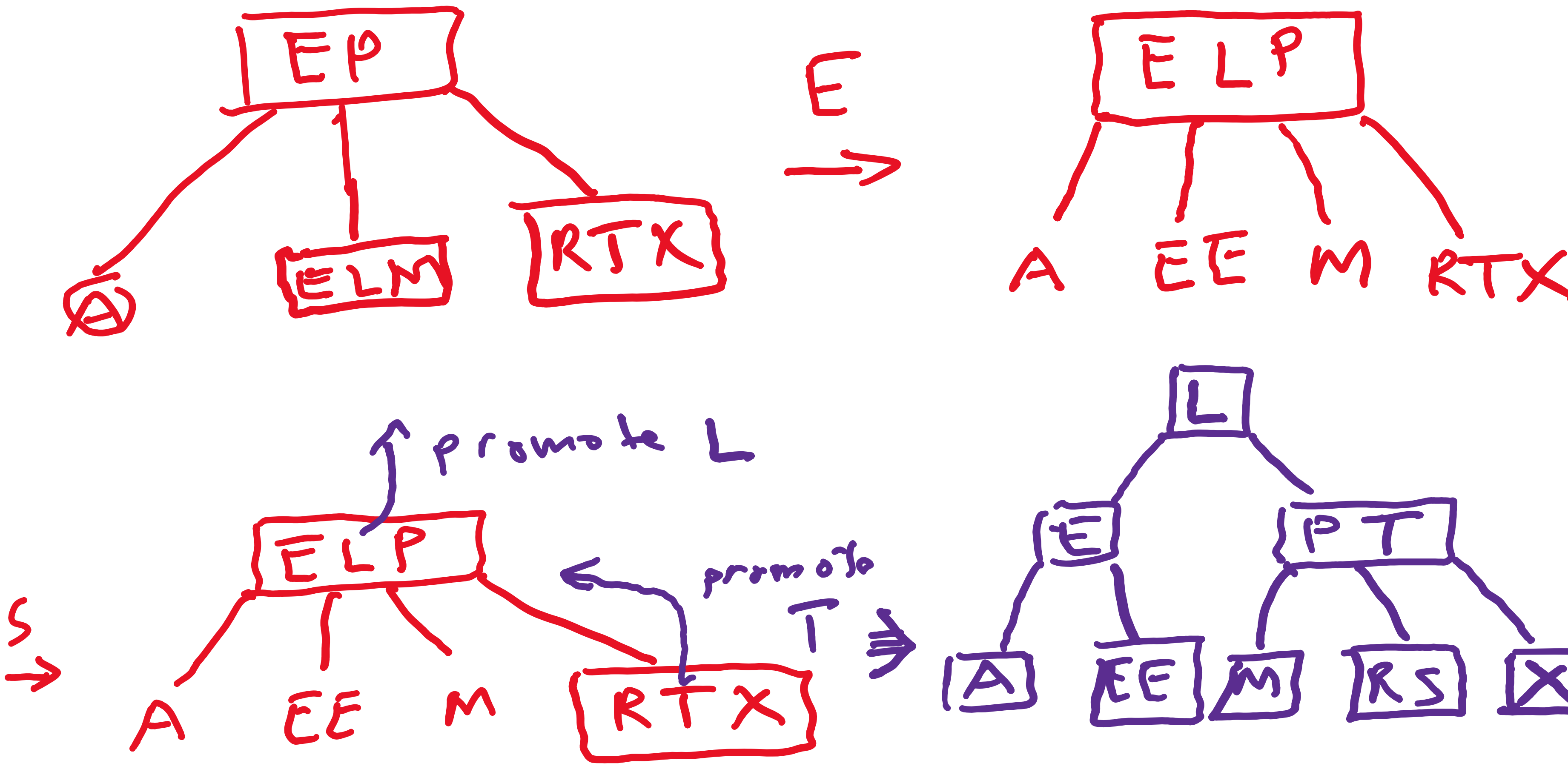
- an equal key will be placed in the right children

Example: insert EXAMPLETREE into an empty tree

Supposing:

- an equal key will be placed in the right children




EP
A
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

Insert

O(?)

Lookup

O(?)

2-3-4 Tree: Time Complexity

Insert

$\Theta(\log n)$

Lookup

$O(\log n)$

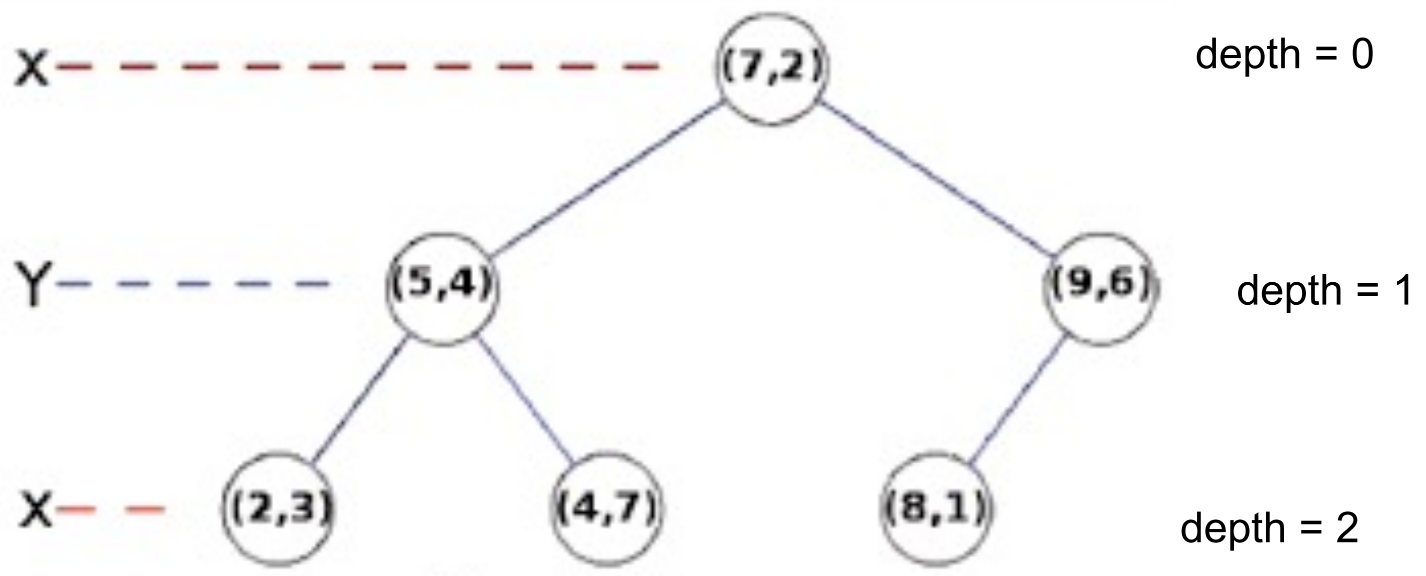
Still wondering about hashing and/or 2-3-4 trees?

Sea a very detailed workshop .ppt for Week 6 in Canvas.

Also note that this presentation is available for download at github.com/anhvir/c203

2D trees: BST tree for 2-component keys

- is a BST tree (not necessarily balanced!)
- but each key has 2 components: `X` (or `key[0]`) and `Y` (or `key[1]`)
- at node with depth `d`, compare/switch/split using `key[d%2]`



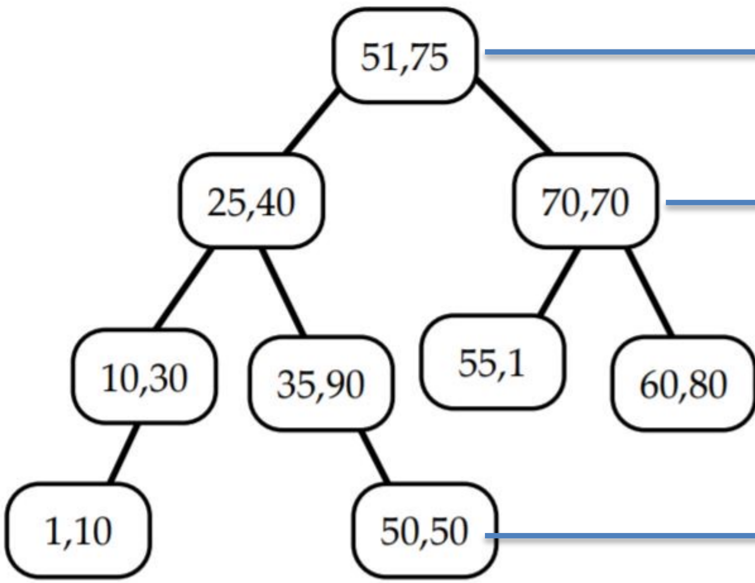
2D tree: Example

Insert the following keys into an initially empty tree:

```
(51,75)
(25,40)
(70,70)
(10,30)
(1,10)
(35,90)
(55,1)
(50,50)
(60,80)
```

2D tree: Example

Insert the following keys into an empty tree



depth 0, switch by `key[0]`

depth 1, switch by `key[1]`

depth 2, switch by `key[0]`

depth 3, switch by `key[1]`

→ depth `d`, switch by `key[d % 2]`

→ in `ass2` you might want to keep `d` in nodes for easy debugging

2D tree: Example

Visualisation in the 2D map: Nodes A, D, E *divide their respective areas into left and right parts; node* B *and* D *– into top and bottom parts.*