COMP20003 Workshop Week 6

Note: Please have draft papers and pens/pencils ready

Hashing
2-3-4 Trees
***k-D(2-D) Trees

Assignment 2: Understanding the requirements

LAB:
- Implementing hashtables
- Assignment 2

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)=` 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`.

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

Colisions

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

Example:

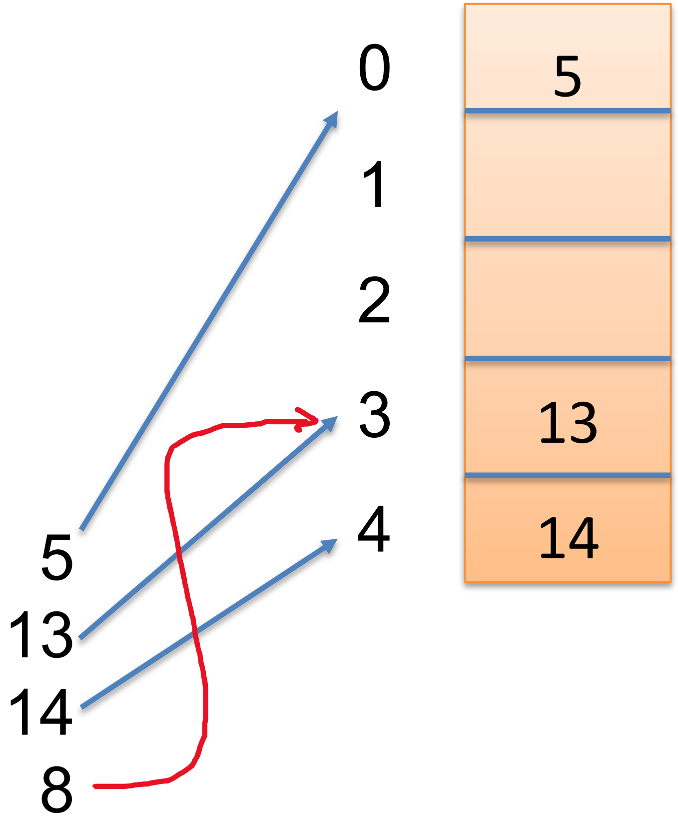
$m=5$, $h(x) = x\%\ m$

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

Collisions are normally unavoidable.

One method *to reduce collisions* using a prime number for hash table size m.

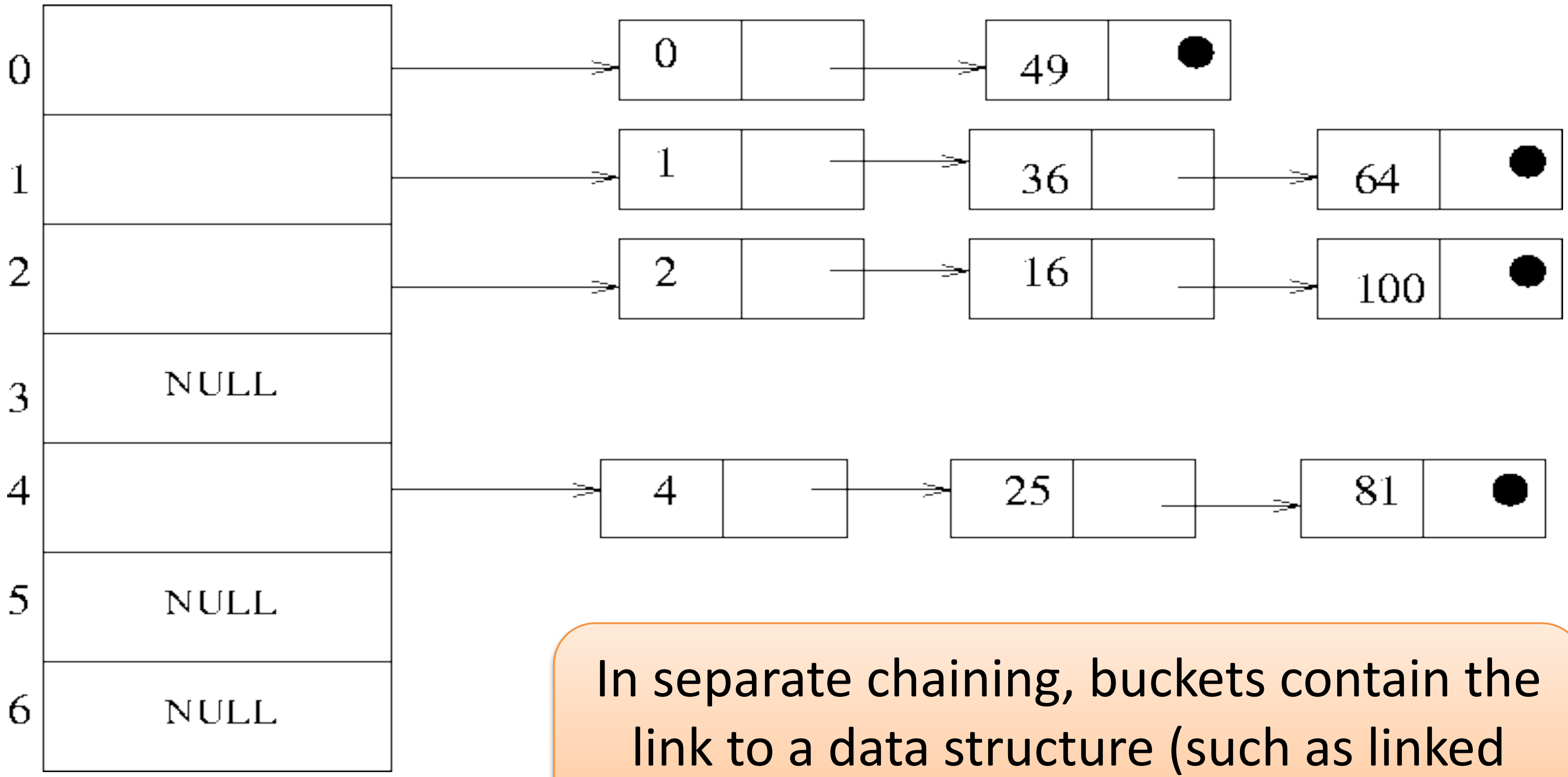
Another method is to make the table size m big enough (but that affects space efficiency).



Collision Solution 1: Separate Chaining

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

`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 buckets)

```
while (HT[index] != NULL)
      index= (index+1)%TABLESIZE
```

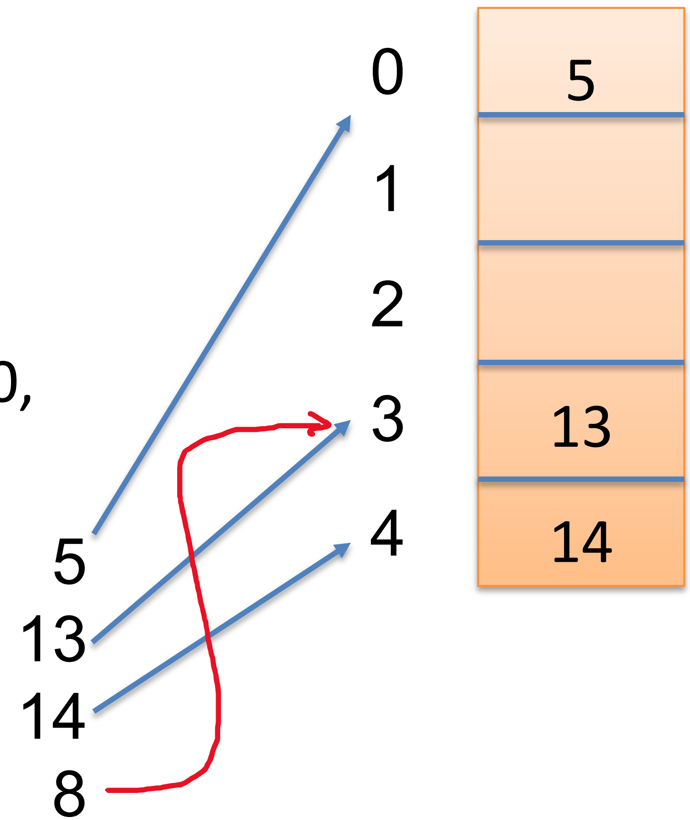
That is, when inserting we do some probes until getting a vacant slot. H(x,probe) can be summarized as:

```
H(x, probe) =
      ( h(x) + probe ) % m
```

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

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

and inserting



Double hashing

```
jumpnum = hash2(key);
while (HT[index] != NULL)
  index=(index+jumpnum)%TABLESIZE
```

Example **hash2** function:

→ `hash2(key) = key%SMALLNUMBER + 1;`

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

where i= 0, 1, 2, … (until reaching a vacant slot). Note that:

`h2(x) ≠ 0 for all x`,

to be good, `h2(x)` should be co-prime with `m`,

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

Q 5.1 a)

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

0	1	2	3	4	5	6	7	8	9	10	11	12

Q 5.1 b)

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

0	1	2	3	4	5	6	7	8	9	10	11	12

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

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 is 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

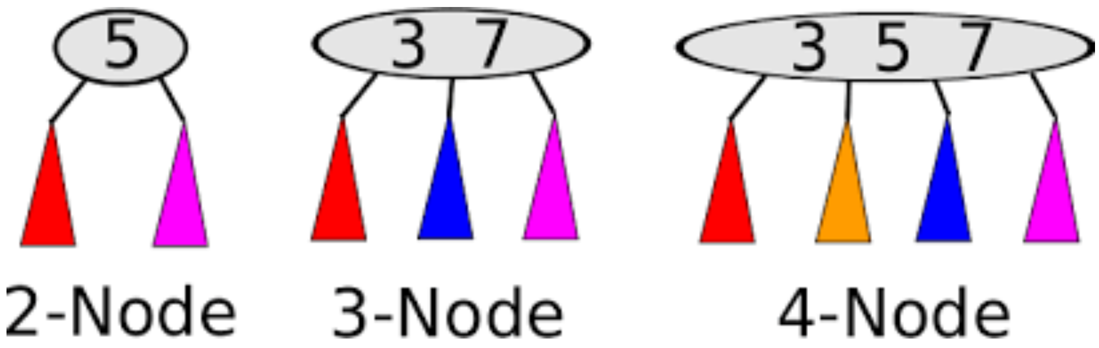
(C)

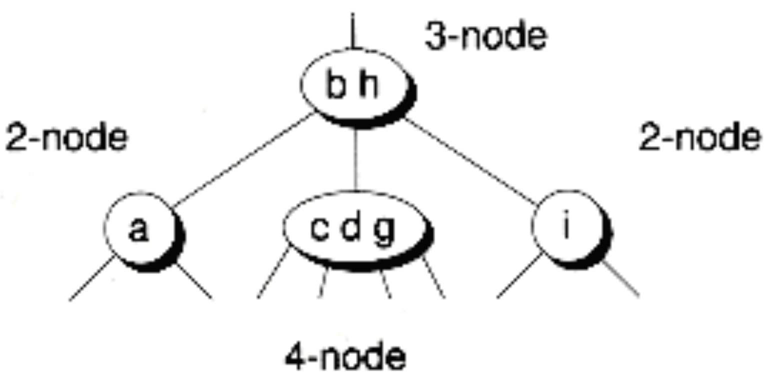
0	
1	
2	12, 2
3	13, 3, 23
4	
5	5, 15
6	
7	
8	18
9	

(D)

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

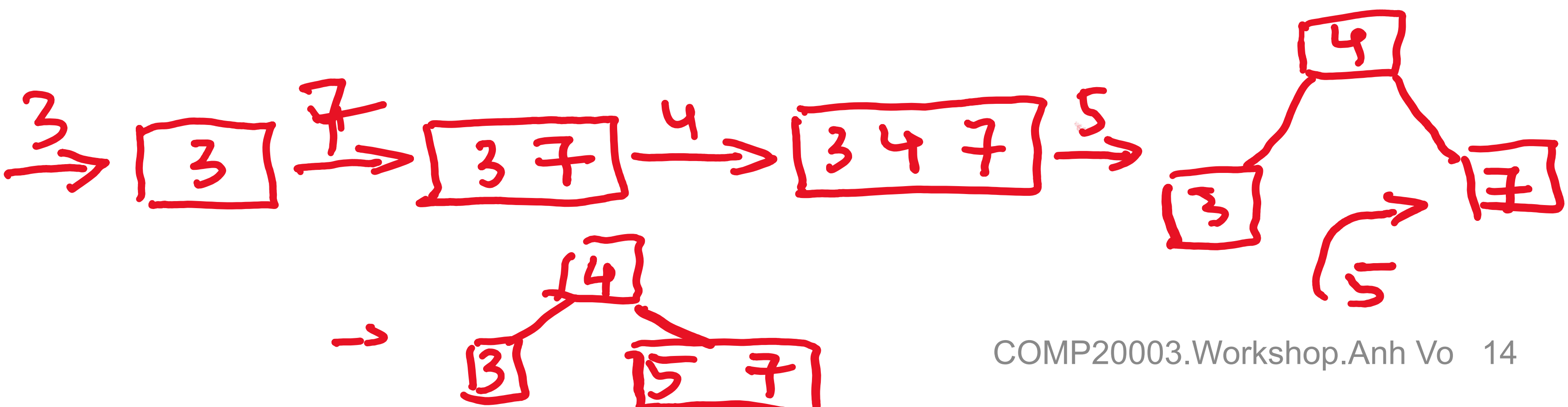




2-3-4 Insertion: inserts a key x into a non-empty tree

- 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

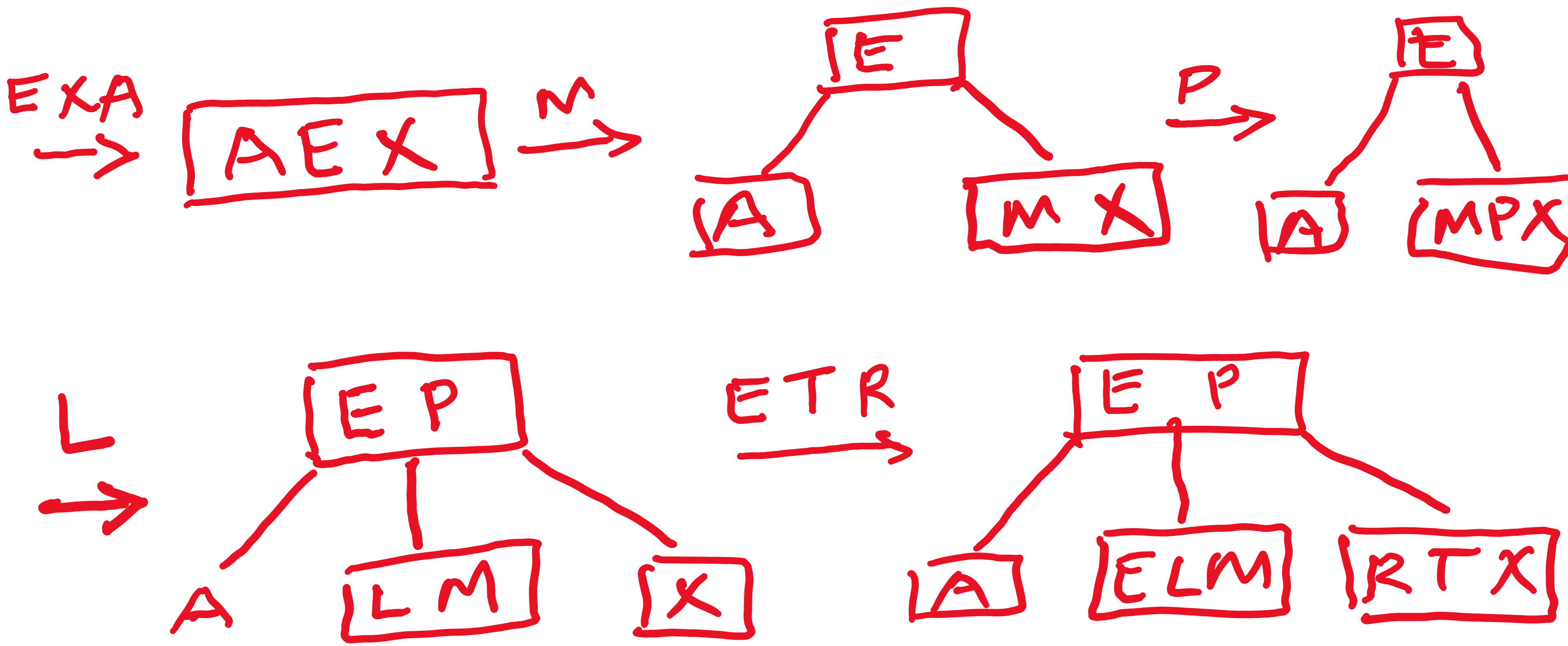
Supposing:

- an equal key will be placed in the right children

Example: insert EXAMPLETREE into an empty tree

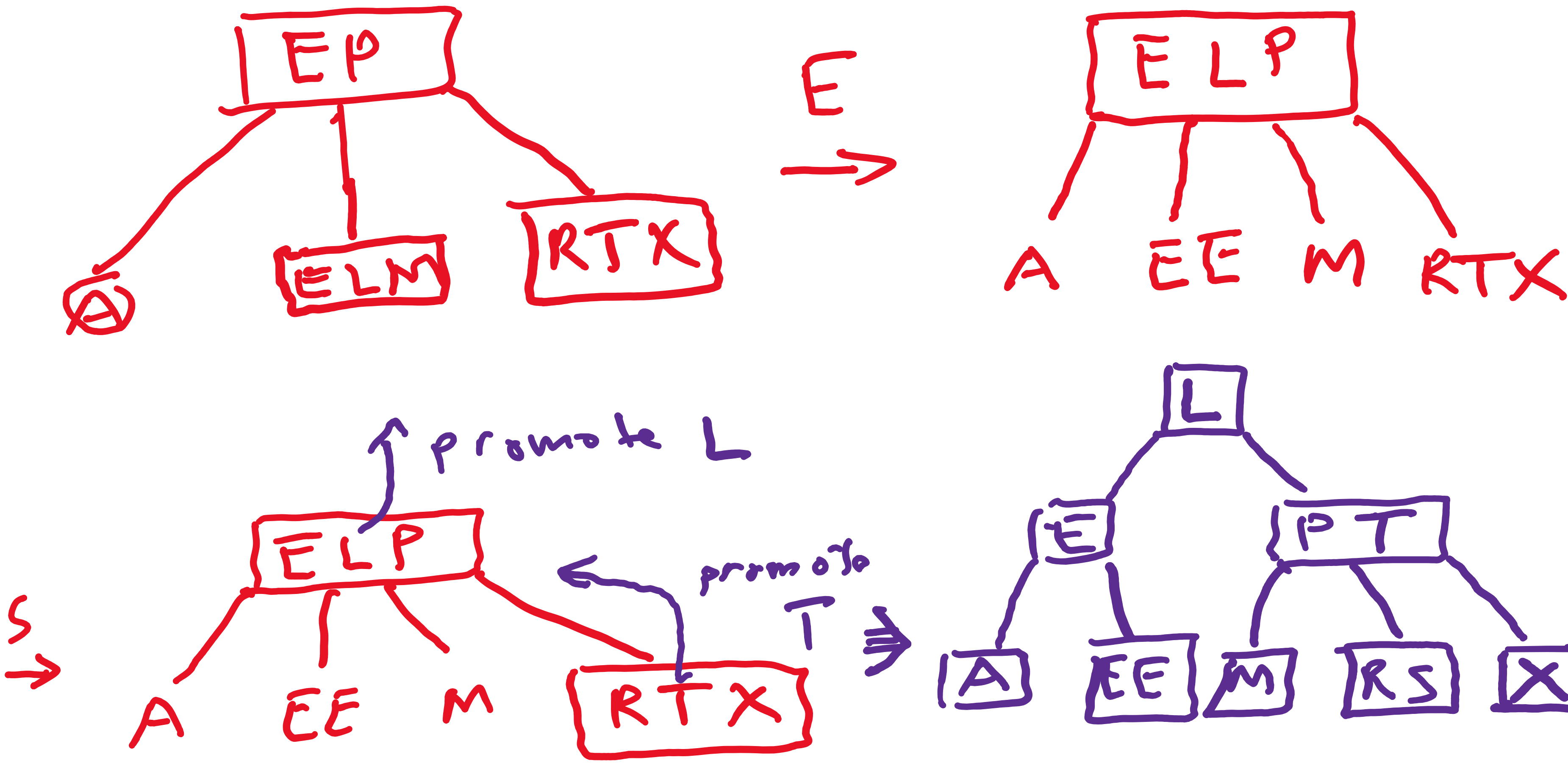
Supposing:

- an equal key will be placed in the right children



E → ?

S → ?


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

$O(\log n)$

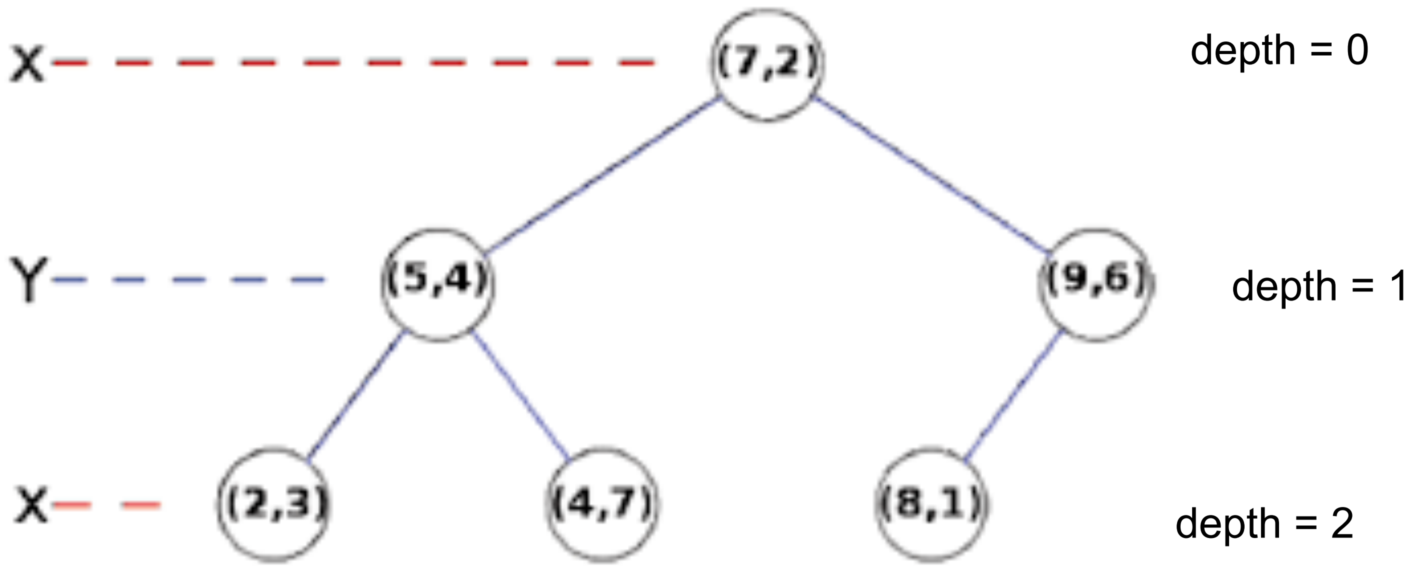
Still wondering about hashing and/or 2-3-4 trees?

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]`



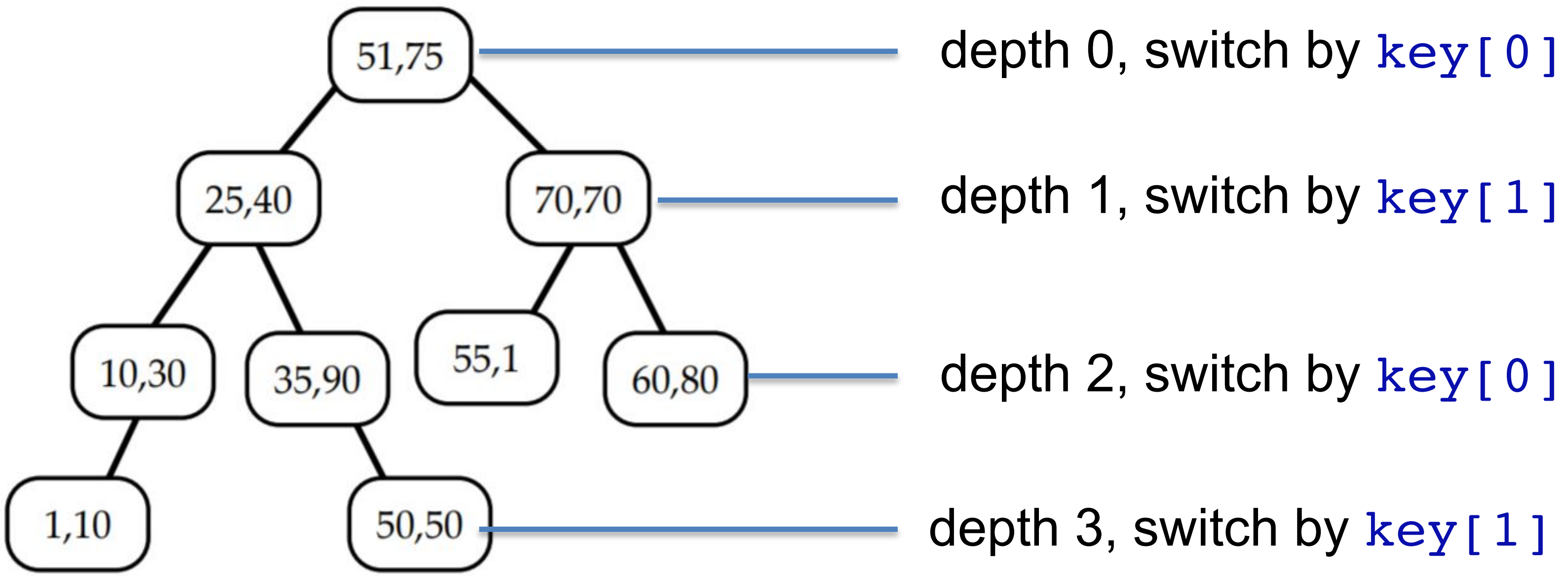
2D tree: Example

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

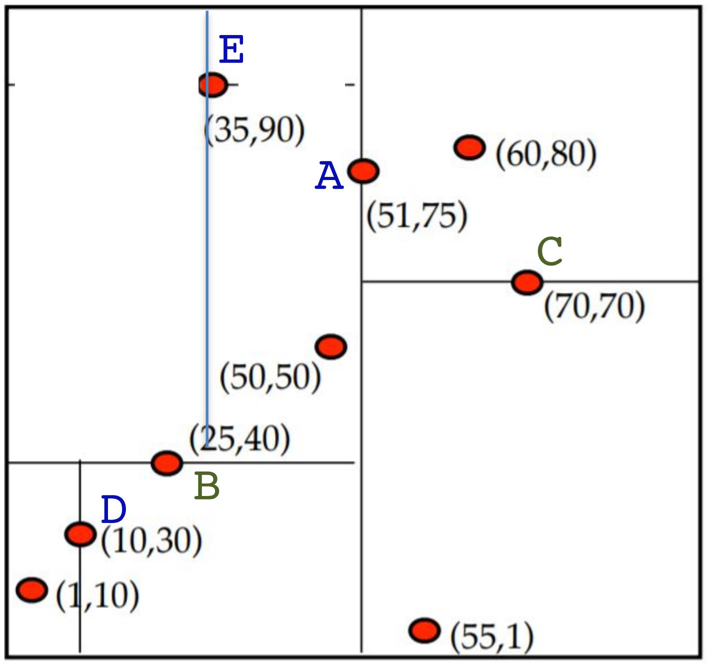


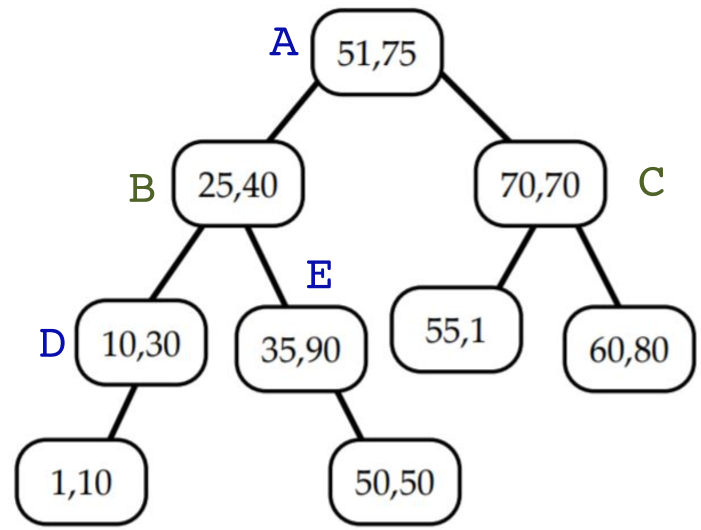
→ depth `d`, switch by `key[d % 2]`

→ 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.*





Assignment 2: Programming Task (Delivery 1)

Build 2 executable files `map1` and `map2`. So you probably will have at least:

- `data` module
- `list` module
- `2dtree` module
- `map1.c` that contains `main()` for stage 1
- `map2.c` that contains `main()` for stage 2

Note that facilities in the `data`, `list`, and `2dtree` modules are used by both `map1.c` and `map2.c`.

simple `Makefile` has the format:

```
all: map1 map2
map1: map1.o 2dtree.o llist.o data.o
   gcc -Wall –o map1 map1.o llist.o data.o
map2: ...
   gcc ... –o map2 ...

...

clean:
    rm –f map1 map2 *.o
```

Makefile: a better version

```
CC = gcc
CFLAGS = -Wall
HDR = data.h llist.h 2dtree.h
SRC1 = map1.c data.c llist.c 2dtree.c
OBJ1 = $(SRC1: .c=.o)
EXE1 = map1
...
all: $(EXE1) $(EXE2)
$(EXE1): $(HDR) $(OBJ1)
	$(CC) $(CFLAGS) –o $(EXE1) $(OBJ1)

clean:
	rm –f $(EXE1) $(EXE2) *.o

$(OBJ1): $(HDR)
$(OBJ2): $(HDR)
```

Repeat for SRC2, OBJ2, EXE2

ass2: Discussions

- why 3 data files supplied?
- reuse our code for the list and data modules? or use the solution code?
- what the tree node looks like?
- what should be the header for the function `insert`?

ass2: some advices

- first, focus on `Stage 1` (by making the `main()` of `map2.c` empty)
- make sure that the module `data` and `list` are ok
- design the node for the 2D tree, if you like you can add a field depth for (and only for) the debugging purpose
- implement `insert,` think of a good header first
- have a good design for the header of function `search`, as we need to get back the number of comparisons
- *use random data file to create a tiny data file of around 10 records (but with at least one duplicate key) for testing (you can obviously draw the tree for this case).*

Assignment 2: Next Week

How to organize experiments and write the Report?

Lab: JH Week 6: use Terminal and files in workshops/week6

Group work:

- *Implement hash table, or*
- *work on your assignment: discussion is OK, showing code is not!*