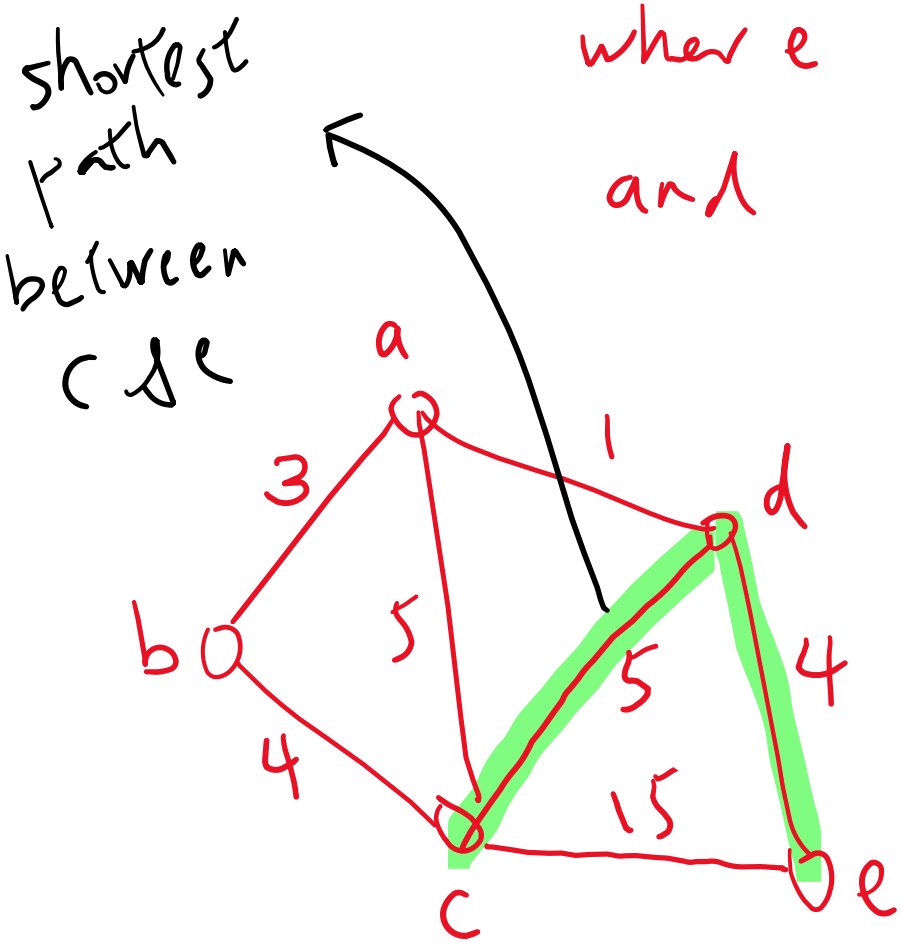
Graph $G = (V, E)$

Defn: A graph has 2 sets V and E,
where V is the set of vertices
and E is the set of edges



$V = \{a, b, c, d, e\}$

$E = \{(a, b), (a, c), (b, c), (a, d), (d, e)\}$

In CS, edges usually have weights

Some Terminology

- vertices a & b are "adjacent" in $G=(V,E)$ if $(a,b) \in E$
  - $\Rightarrow$ a is b's neighbor
  - b is a's neighbor
  - $\Rightarrow$ a is "incident to" (a,b)
  - $\Rightarrow$ the degree of a, $d(a)$, is the number of a's neighbors
- $n = |V|$, $m = |E|$

- Edge, arc

Defn: A path between src & dst in $G=(V,E)$ is a sequence of "distinct" vertices $v_1, v_2, \cdots, v_k$ in G such that:

1. $v_1 = src$, $v_k = dst$
2. $(v_i, v_{i+1}) \in E$, for every $1 \le i \le k-1$

- The "distance" of a path $v_1, \dots, v_k$ in an "edge-weighted" graph with edge weight function w

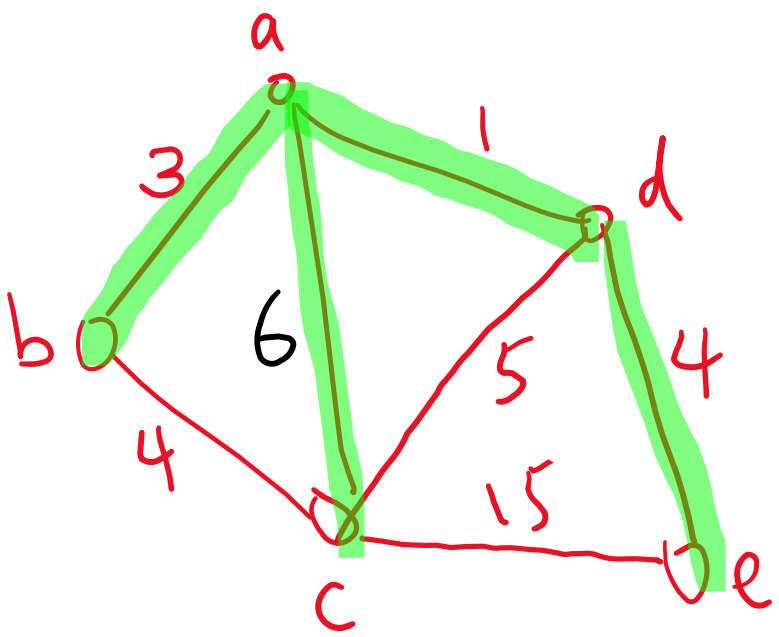
  is $w(v_1, v_2) + w(v_2, v_3) + \dots + w(v_{k-1}, v_k)$

- Shortest path between src and dst in G
  is the path between src & dst
  with the smallest distance.

Shortest Path Tree

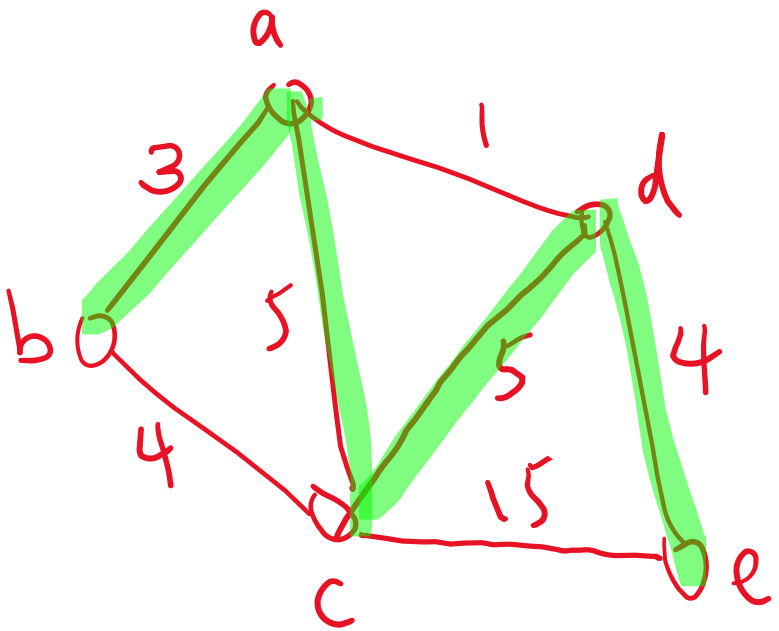
- Input: 1° an edge-weighted graph $G=(V,E)$ with edge weight w. (w is non-negative)
  2° a root src in V
- output: A shortest path T rooted at src
  (i.e., for any vertex v in G,
  the path from v to src in T
  = the shortest path between v & src in G)

root = a



Minimum Spanning Tree

- Input: an edge-weighted graph $G = (V, E)$ with edge weight w
- output: a spanning tree (i.e., a tree that contains V) with the smallest total edge weight



How to Store an Edge-Weighted Graph?

① Adjacency Matrix

$arr[i][j] = w(i,j)$ if $(i,j) \in E$

$arr[i][j] = -\infty$ otherwise

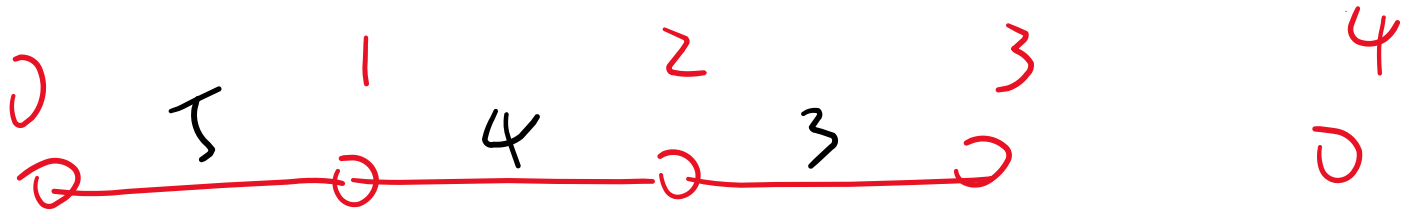
assume $V = \{0, 1, 2, \cdots, n-1\}$

space $= \Theta(n^2)$

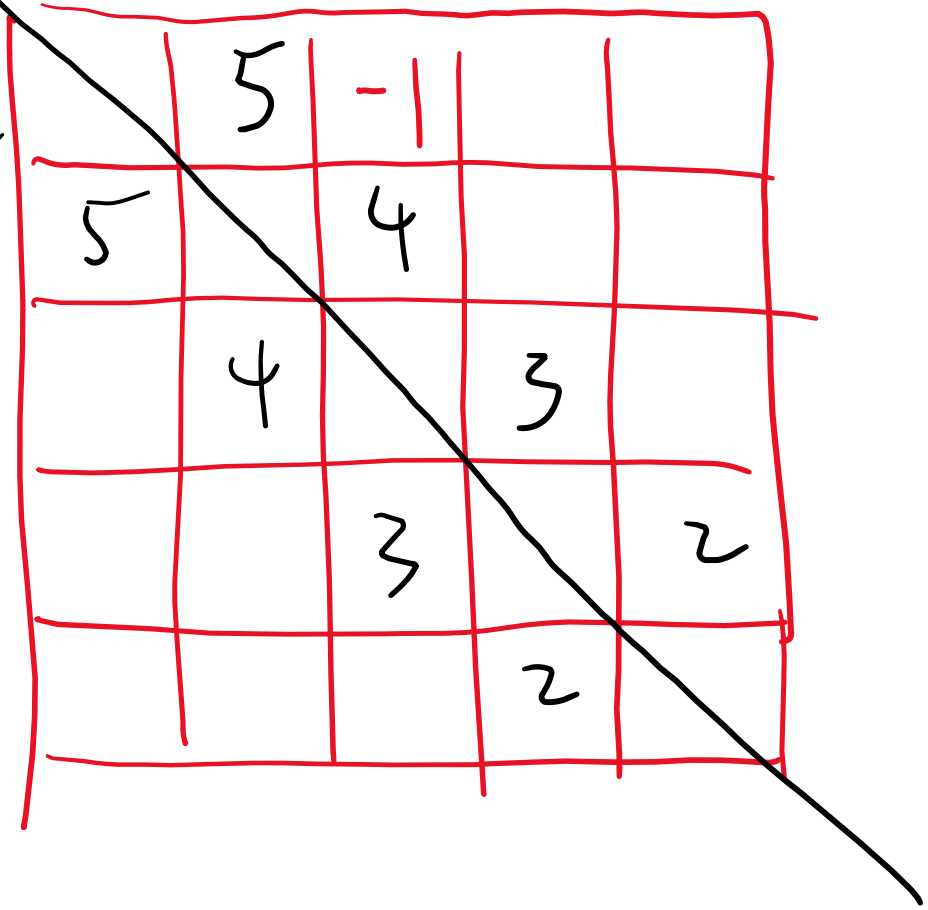
waste of space if G is "sparse"

(i.e., $|E|$ is small)

G

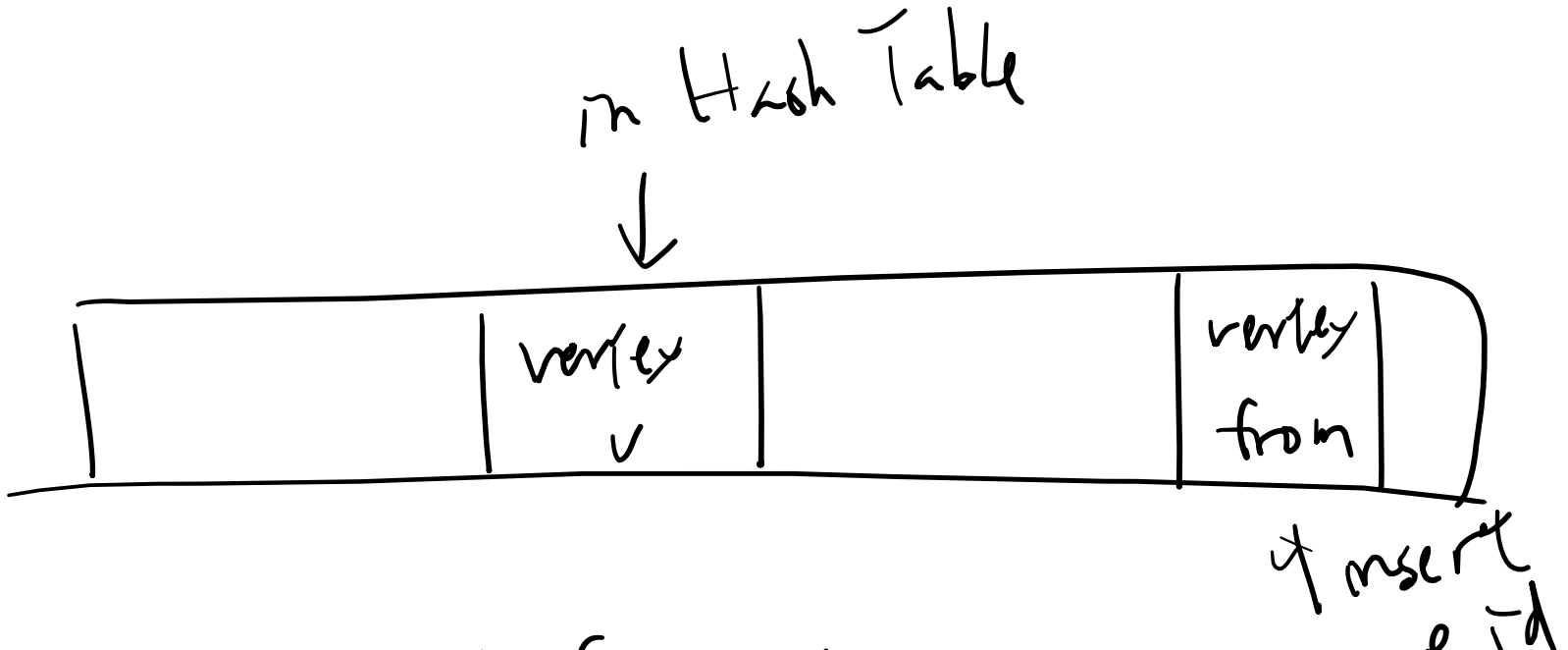


int arr[i][j]

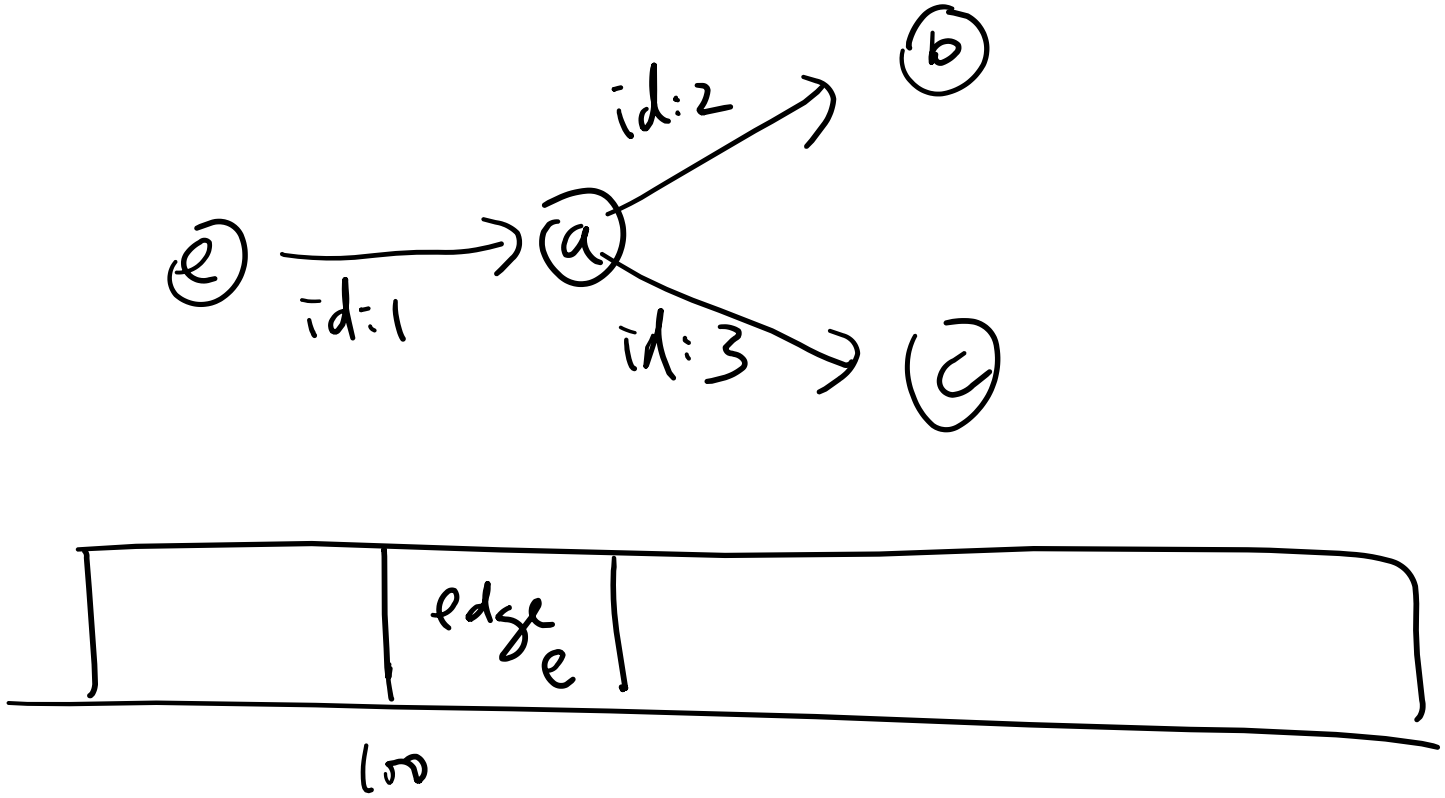


$E = \{(0,1), (1,2), (2,3), (3,4)\}$

$|E| = 4$



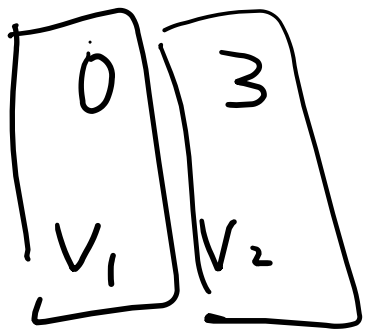
vertex_from = v;


e
a
b
c
id:1
id:2
id:3
edge e
log

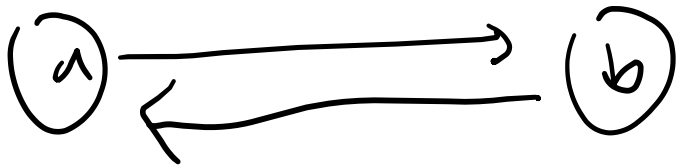
hash(id) → memory address

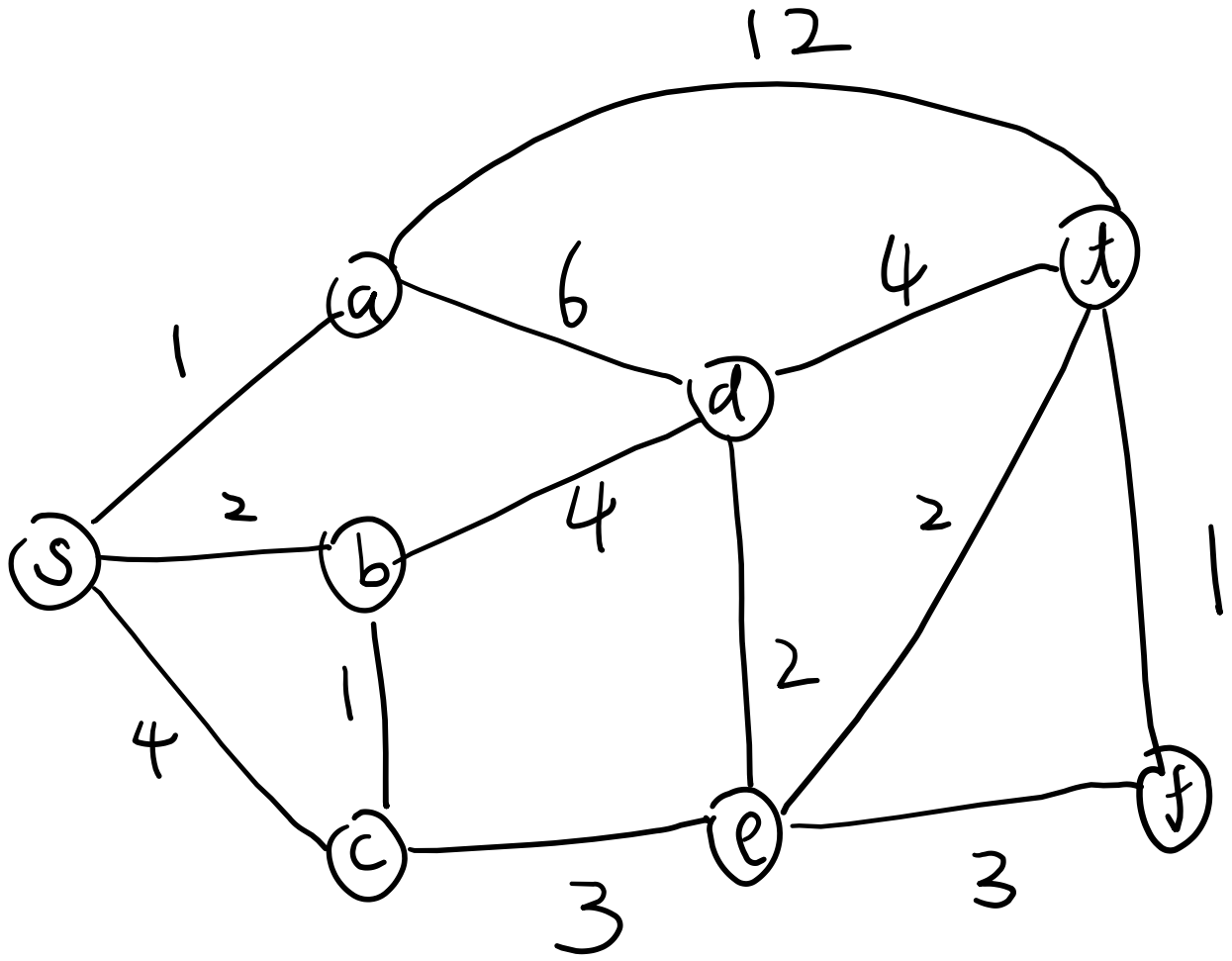
key → id

value → vertex




a
b


a
b



Key Idea:

① Gradually construct the SPT

- Let v_i be the i^{th} closest node to src

  v_0 = src

- In round i:

  a. find v_i

  b. add v_i to SPT

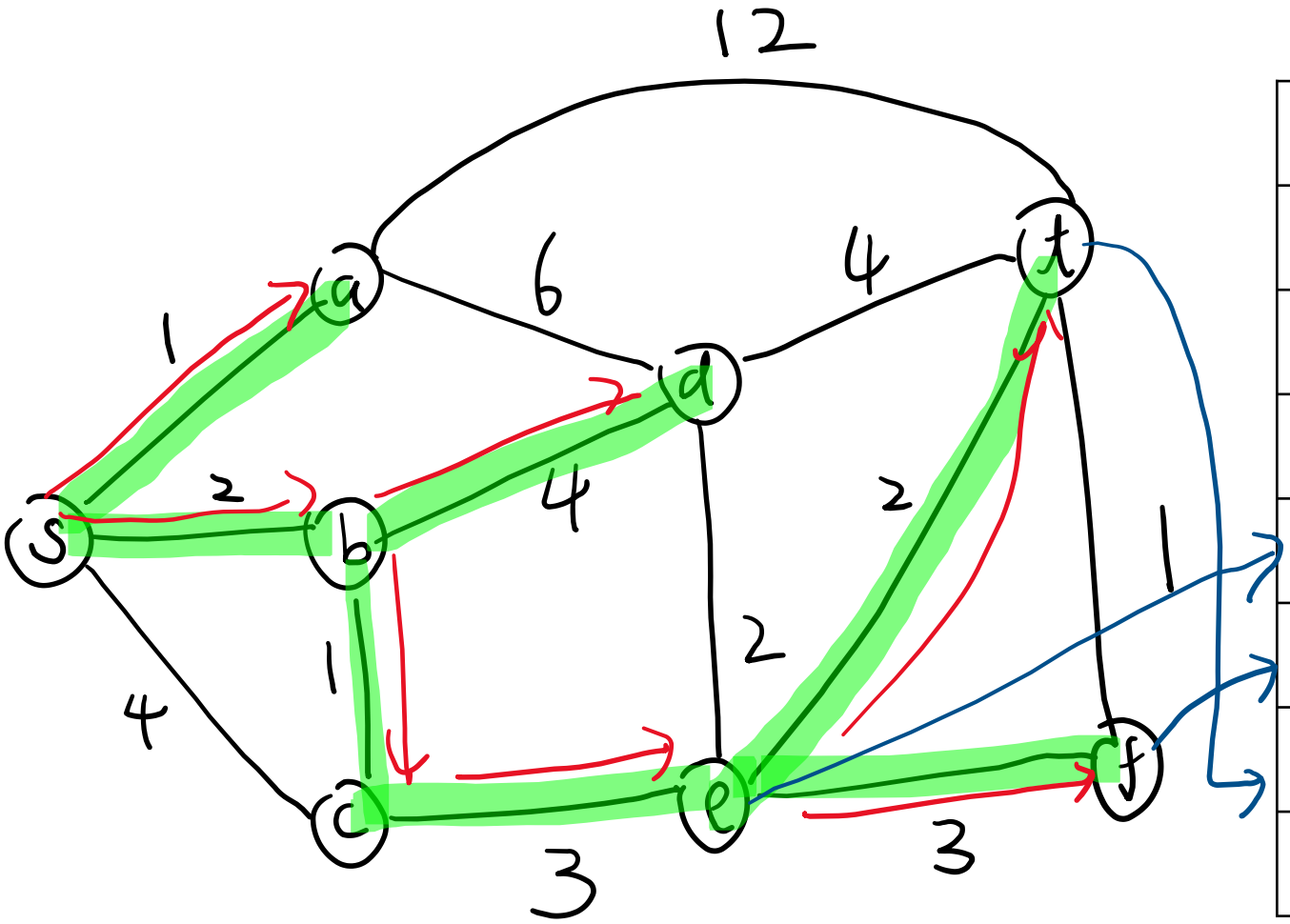
Key Observation:

v_i is adjacent to some vertex in $\{v_0, v_1, v_2, \cdots, v_{i-1}\}$

---

maintain the following info:

1. parent(v): the temporary parent of v in SPT
2. dist(v): the temporary shortest distance between v & src



id	parent	dist
a	s	1
b	s	2
c	b	3
d	b	6
e	c	6
f	e	9
t	e	8

Initialization:

for every neighbor v of src,

$parent[v] = src$

$dist[v] = w(src, v)$

for every vertex v that is not adjacent to v:

$parent[v] = nil$

$dist[v] = \infty$

In Round i: $O(n)$

$O(n)$ 1° Find the vertex v^* with the smallest dist

$O(1)$ 2° Add v^* to SPT according to parent(v^*)

$O(\#v^*\text{'s neighbors})$ 3° For every neighbor v of v^*:

if (dist[v^*] + $w(v, v^*)$ < dist[v])

dist[v] = dist[v^*] + $w(v, v^*)$

$O(1)$ 4° delete v^*

add a special tag

Check If v_i & v_j are adjacent.

① Method 1: Adjacency Matrix:

if arr[i][j] ≥ 0, then v_i & v_j are adjacent

O(1)

else, v_i & v_j are not adjacent

② Method 2: Our code (adjacency list)

① get v_i from V (a hash table)

② get v_i's adjacent edges (a vector)

③ Check if v_j is in the vector

O(# v_i's neighbors)

Output v_i's neighbors

Method 1: Adjacency Matrix

scan $arr[i][0], arr[i][1], \ldots, arr[i][n-1]$

$O(n)$

v_j is v_i's neighbor if

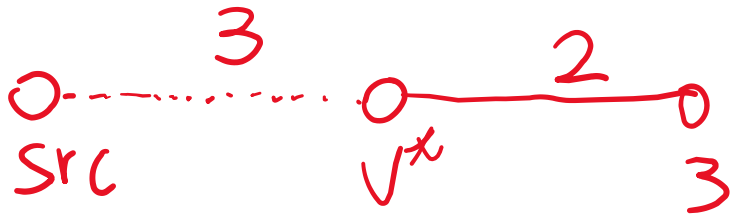
$arr[i][j] > 0$

Method 2: Adjacency list

$O(\#neighbors)$ output the edge list

Time Complexity:

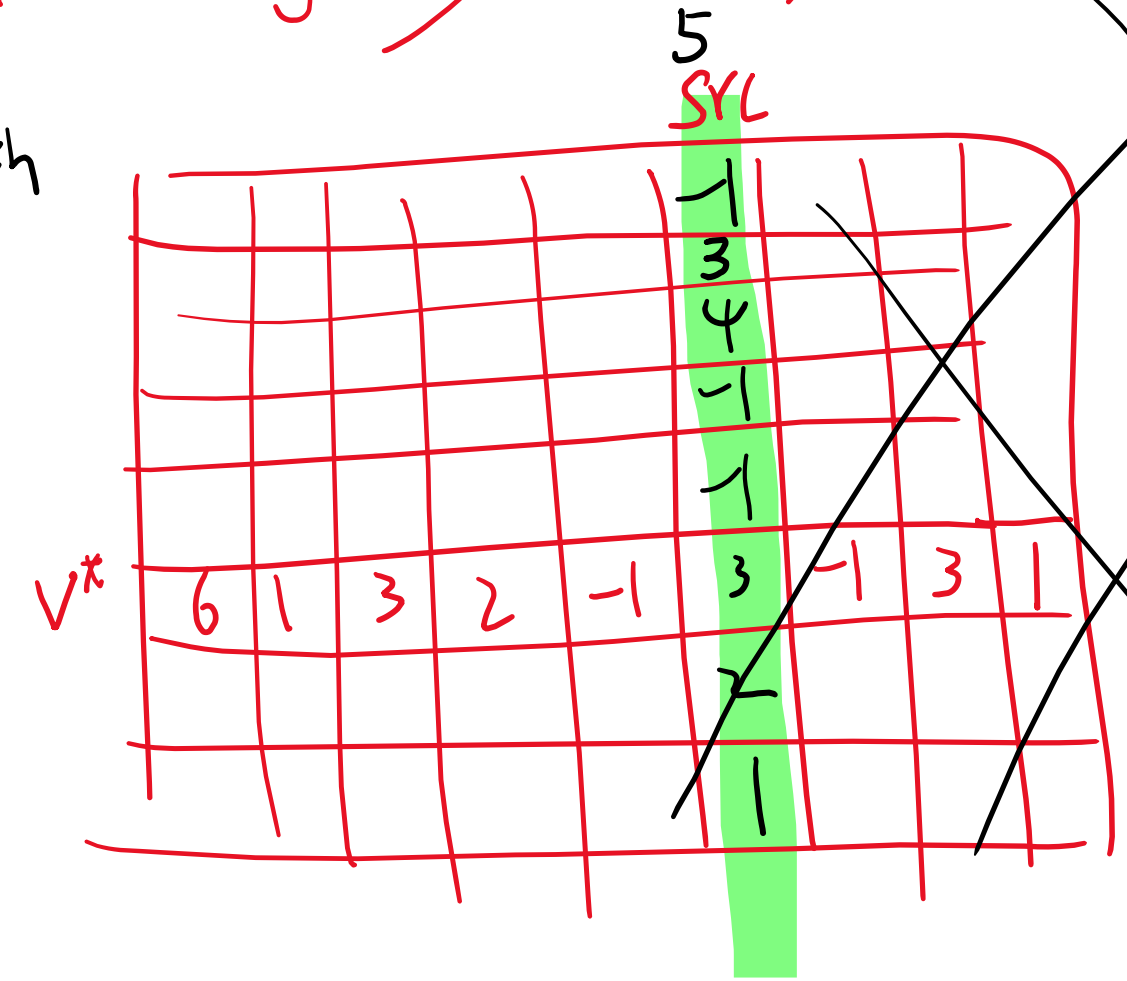
① Initialization:



Method 1: Adjacency Matrix

scan the 5^{th} row

$O(n)$



Id	parent	dist
0	nil	∞
1	5	3
2		
3	nil	~~∞~~ 5
4		
5		3

v*

Method 2: Adjacency list

① For every vertex $v_i \in V$,

$parent(v_i) = nil$

$dist(v_i) = \infty$

Time = Time to visit all vertices

$= O(n)$

② For every neighbor v_i of s

$parent(v_i) = s$

$dist(v_i) = w(s, v_i)$

Time = Time to visit all neighbors of s

$= O(n)$

Time to add an edge to a graph (input: v_i & v_j)

Assumption: the graph has all the vertices already

Method 1 (adjacency matrix)

arr[i][j] = 1

O(1)

~~arr[j][i] = 1~~

Method 2: Adjacent list

O(1)

Update dist:

Method 1: O(n)

scan v^*'s row. If $arr[v^*][j] \geq 0$ (v_j is adjacent to v^*)

& $arr[v^*][j] + dist[v^*] < dist[j]$

then $dist[j] = arr[v^*][j] + dist[v^*]$

can be obtained in O(1) time

Method 2

scan v^*'s adjacency list & update distance.

$O(\#\ v^*\text{'s neighbors})$

Total Time Complexity of Dijkstra's Algorithm

$O(n) \times O(n) = O(n^2)$

$O(n)$: n rounds

$O(n)$: Time complexity per round.

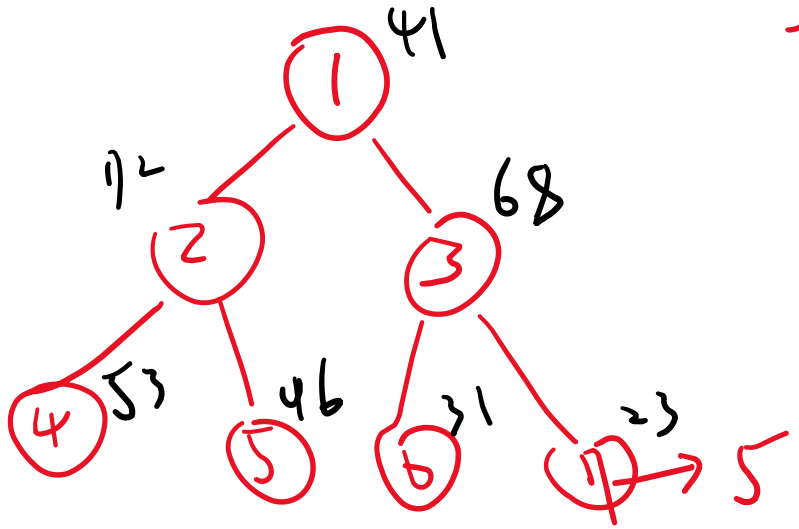
when we store dist in an array

Heap:

① ExtractMin: Find and delete the smallest data (e.g., dist)

② Insert(x): insert x to heap (x can be dist)

③ DecreaseKey(addr, newDist): update the dist stored at addr to newDist



DecreaseKey(23, 5)

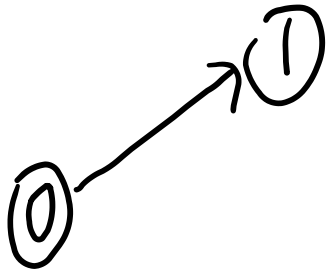
Array as a heap:

ExtractMin: $O(n)$

Insert : $O(1)$

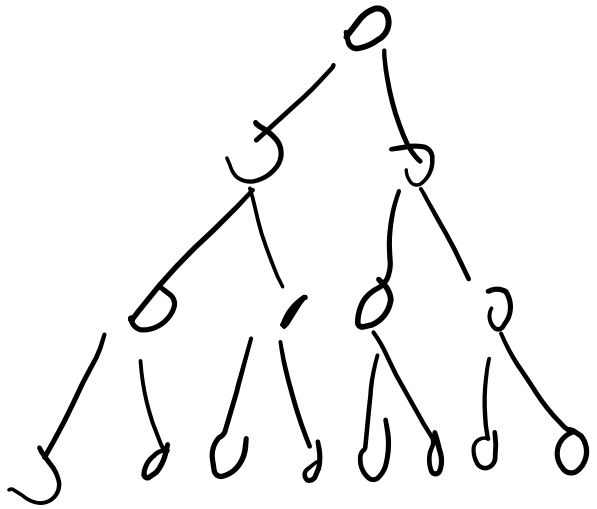
DecreaseKey : $O(1)$

	Insert	extract Min	Decrease Key	
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m \log n)$
Binomial Heap	$O(1)$ amortized	$O(\log n)$	$O(\log n)$	$O(m \log n)$
Lazy Binomial Heap	$O(1)$	$O(\log n)$ amortized	$O(\log n)$	$O(m \log n)$
Fibonacci Heap	$O(1)$	$O(\log n)$ amortized	$O(1)$ amortized.	$O(n \log n + m)$

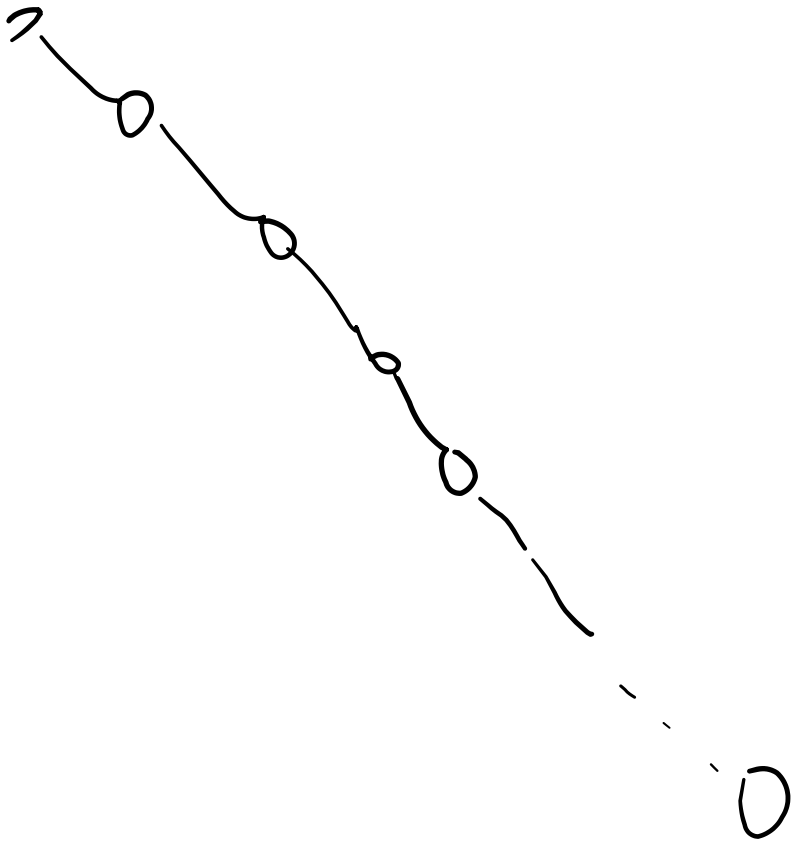


	0	1	2	3	4	5	6	7
0		1	2	4				
1	1				6			12
2	2			1	4			
3	4		1			3		
4		6	4			2		4
5				3	2		3	2
6						3		1
7		12			4	2	1	

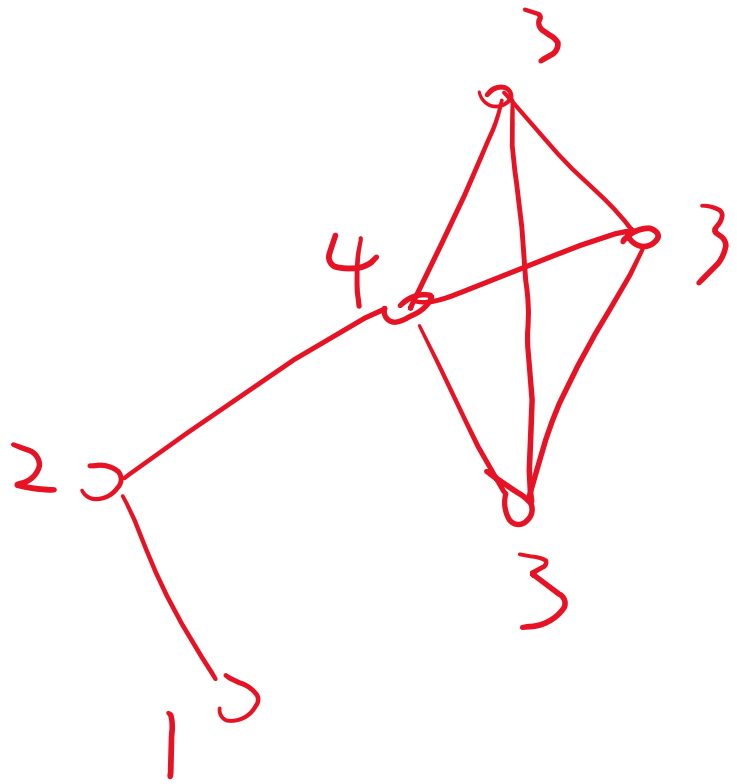
v*	id	parent	dist
	0		
	1	0	1
	2	0	2
	3	0	4
	4	1	1+6
	5	nil	∞
	6	nil	∞
	7	nil	1+12



$\log n$



n



degree of v = # v's neighbors

$$\sum_{v \in V} \#\, v\text{'s neighbors} = 16$$

$$\#\,\text{edges} = 8$$

$$\| \; m$$

$$\sum_{v \in V} \text{degree of } v = 2m$$

Time Complexity of Dijkstra's Algorithm with Heap.

T_I: time complexity of insertion $O(n)$

T_E: time complexity of extractMin $O(n)$

T_D: time complexity of decreaseKey $O(\sum_{v \in V} \#\, v\text{'s neighbors})$

$=$

$O(m)$

$O(n)(T_I + T_E) + O(m)T_D$

Time Complexity of Dijkstra's Algo

$$m \log n < n^2$$

when m is small

For example, when $m = O(n)$

$$m \log n = O(n \log n)$$

Is it possible to design a heap whose insertion & extractMin is faster than $O(\log n)$?

(e.g., $O(\sqrt{\log n})$)

NO

Sorting based on heap:

1° insert n data

2° extractMin n times

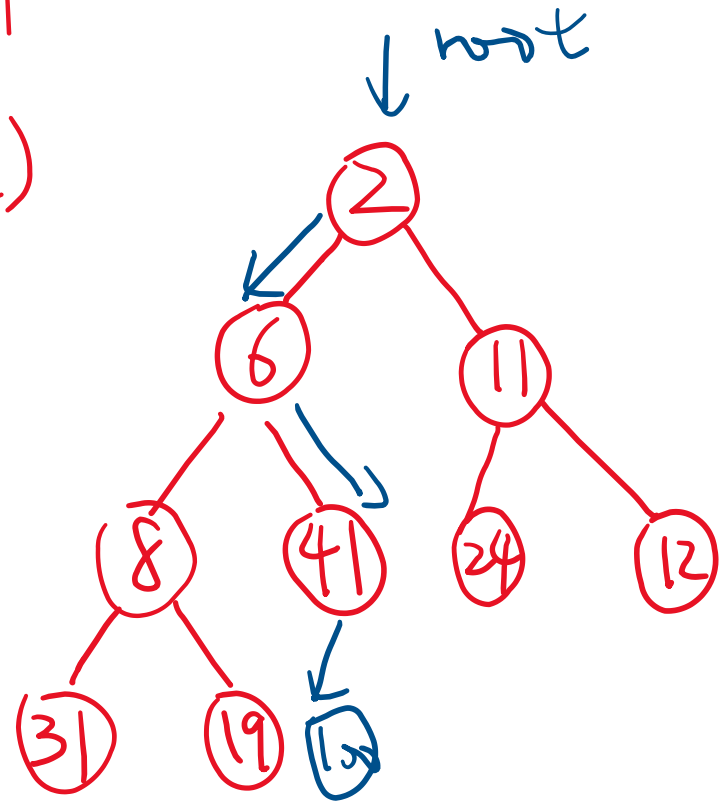
assume heap is comparison-based

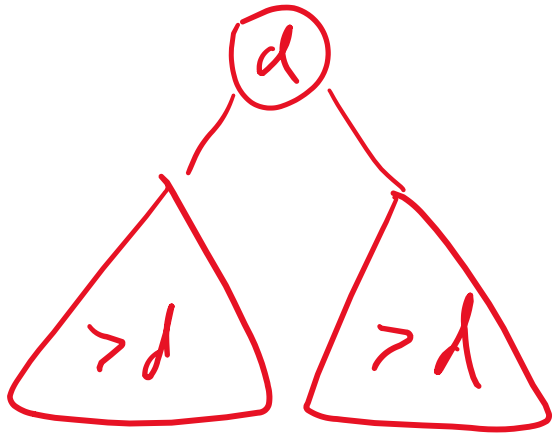
$n(T_I + T_E) \geq n \log n$

$\Rightarrow T_I + T_E \geq \log n$

Binary Heap

Insert (x)





1° insert x at the bottom leftmost node

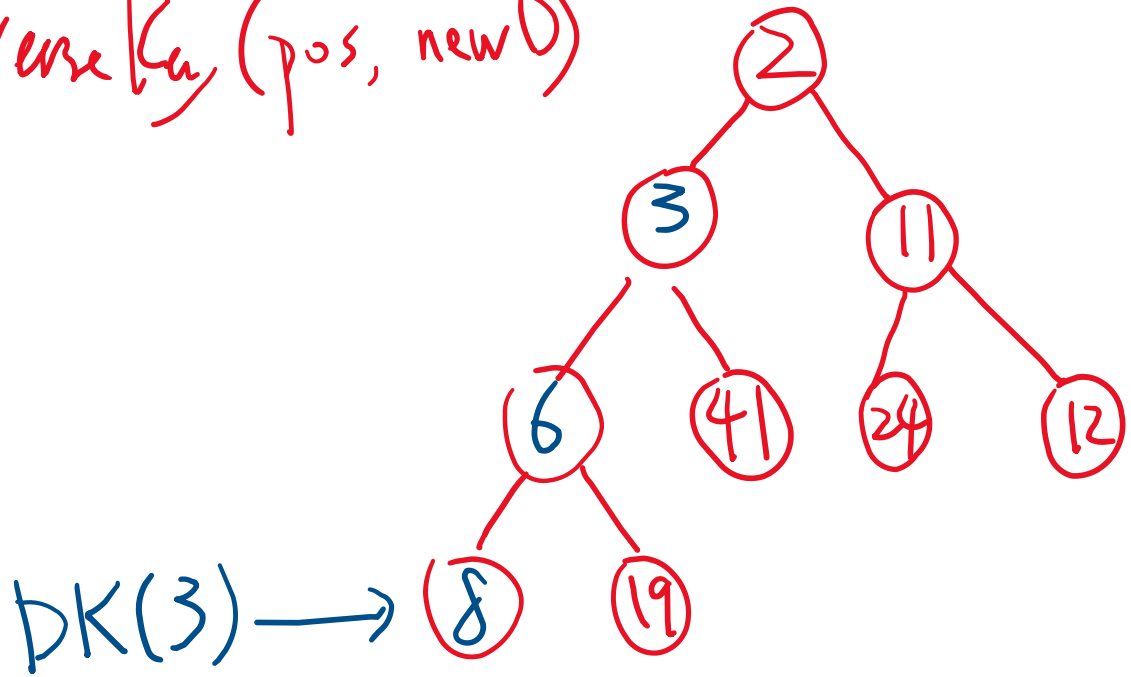
2° swap upward if parent > x

Time Complexity = $O(\log n)$

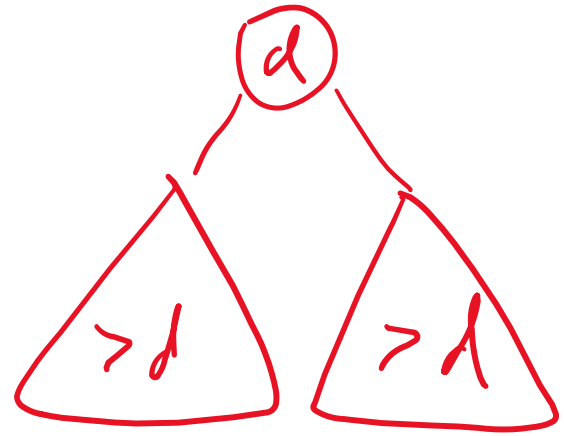
Tree height = $O(\log n)$

Binary Heap

DecreaseKey(pos, newD)



DK(3) ⟶



1° replace the data stored at pos with newD

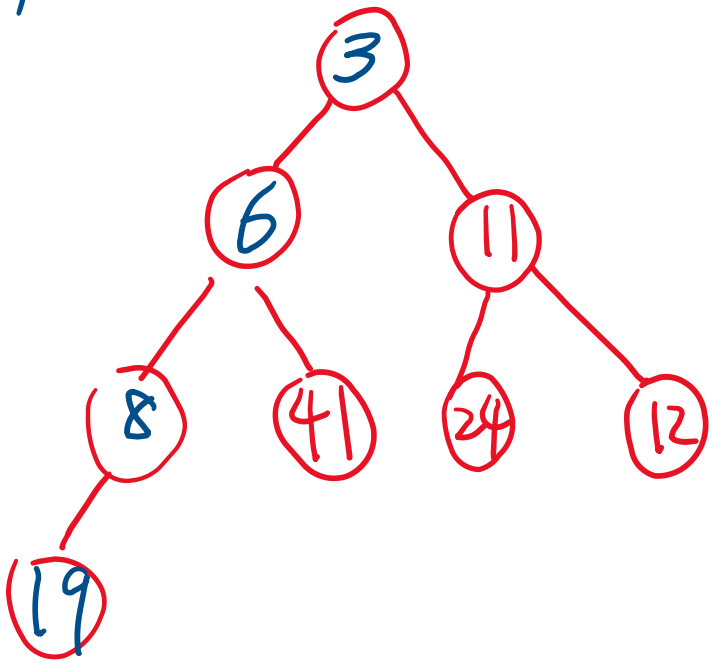
2° Swap upward if parent > newD

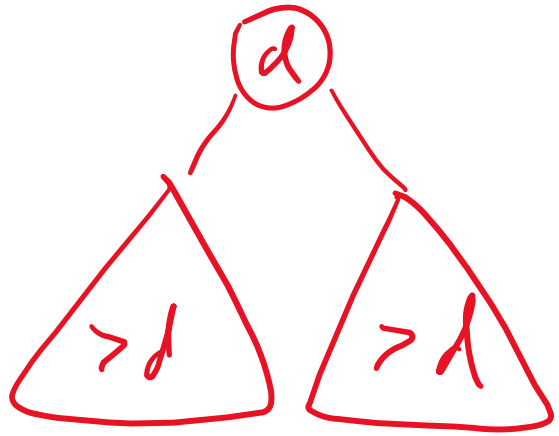
Time Complexity = O(log n)

Binary Heap

ExtractMin()

Time complexity
$= O(\log n)$

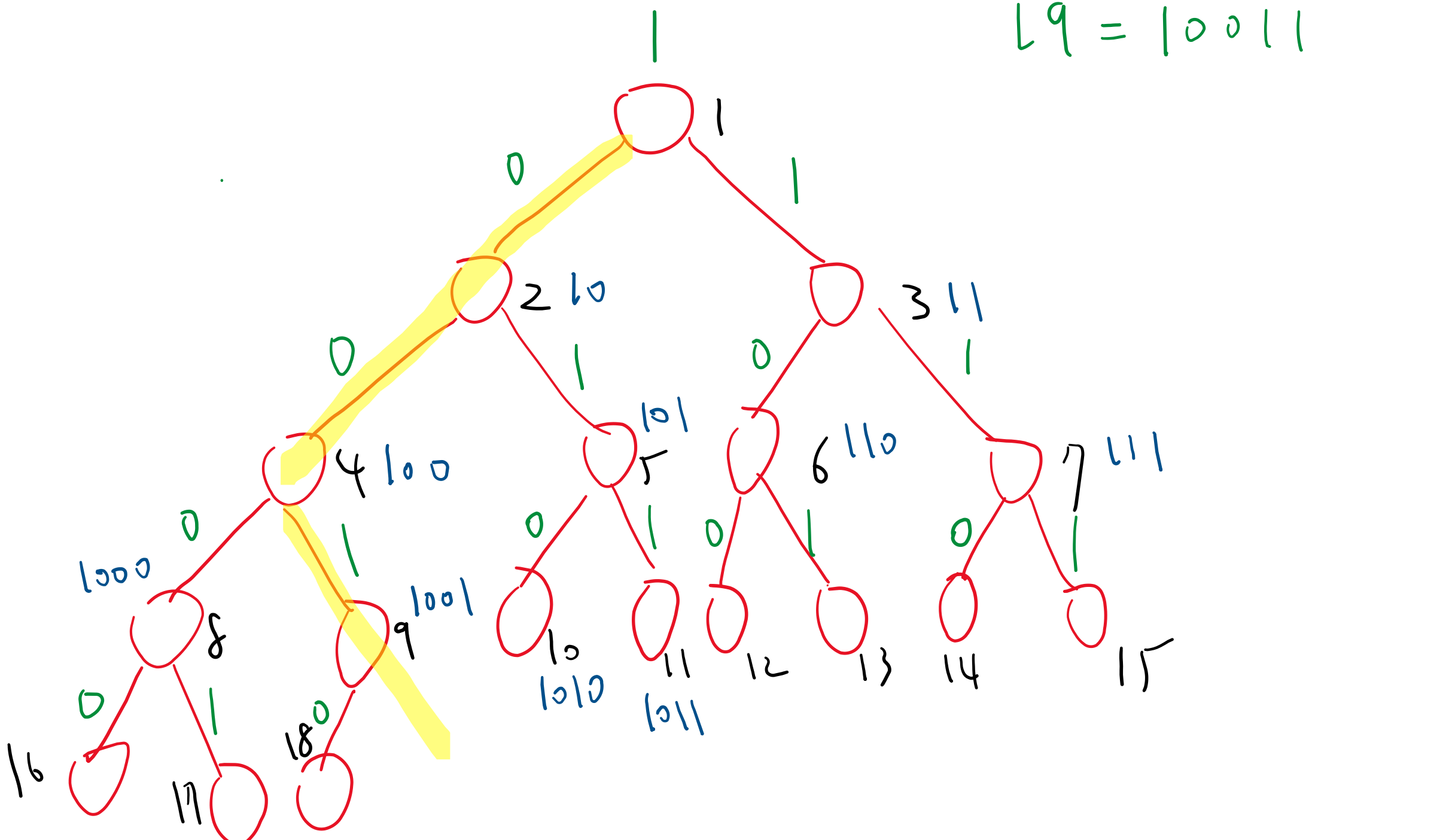




1° output the data stored at root

2° Replace root with the data stored at the bottom leftmost node

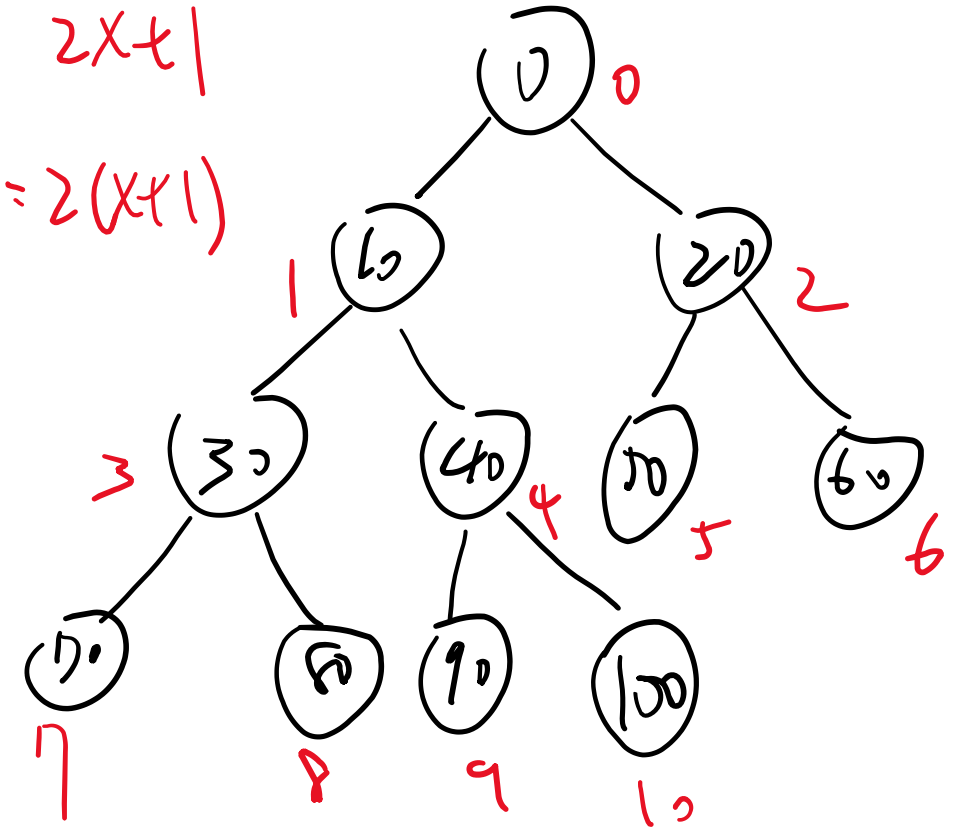
3° Swap downward from the root, if needed (swap with the smaller child)


19 = 10011
1
1
0
1
2 10
0
1
3 11
0
1
4 100
101
5
0
1
6 110
7 111
0
1
0
1
0
1
1000
8
9 1001
10
1010
11
1011
12
13
14
15
0
1
0
16
17
18

left_child(x) : 2x+1

right_child(x) = 2(x+1)

used by extractMin



parent(x) (used by insertion & decreaseKey)

if x is odd

(x-1)/2

else

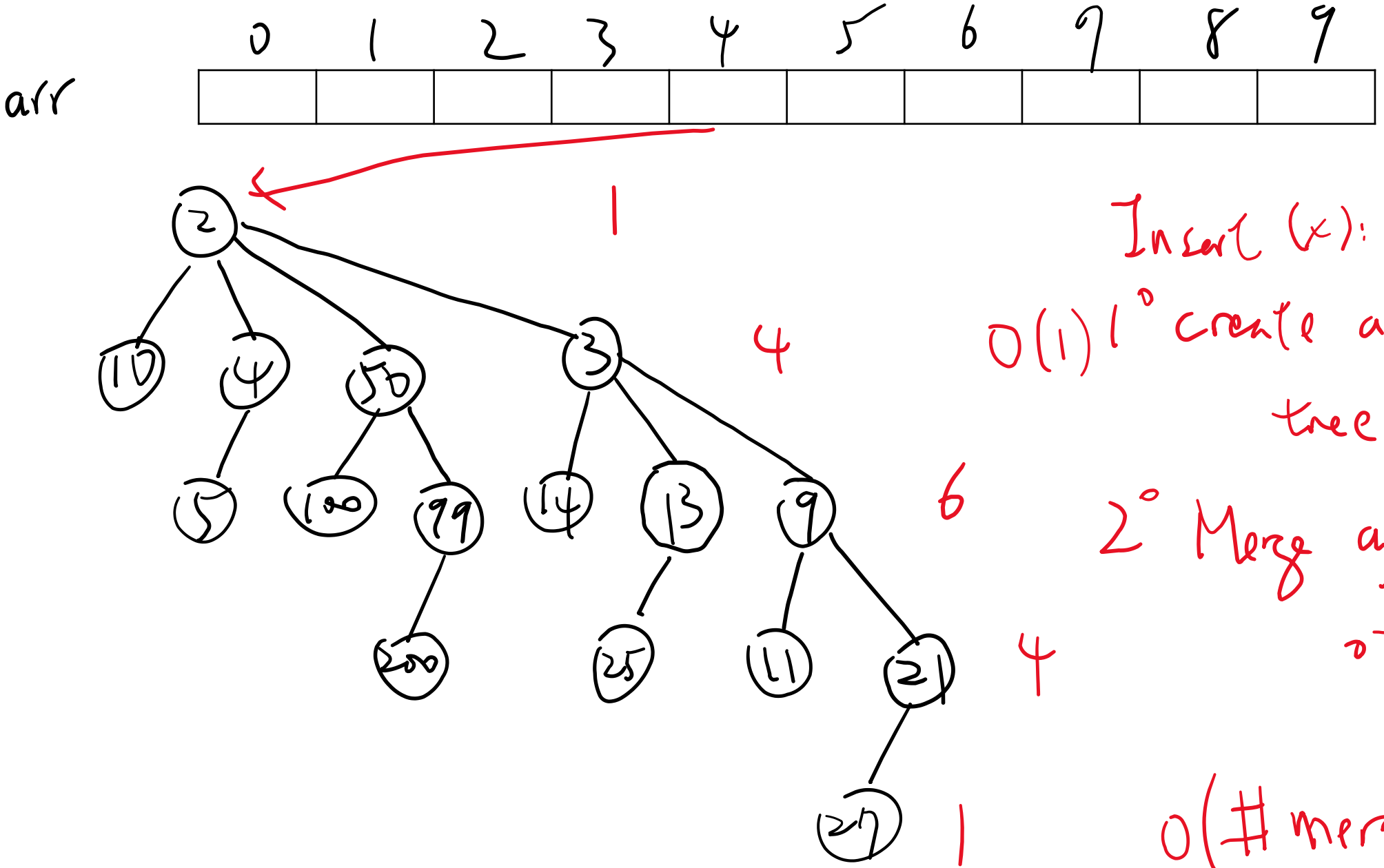
x/2 - 1

0	1	2	3	4	5	6	7	8	9	10				
10	20	30	40	50	60	70	80	90	100					
0	1	2	3	4	5	6	7	8	9	10				

Binomial Heap

- array of trees (every tree is a heap)
- tree size $\in \{1, 2, 4, 8, 16, \ldots\}$
  - tree size $= 2^i$
- No trees have the same size
- if tree size $= 2^i$, then tree height $= i$

arr[i] stores the addr of the root of the size-2^i tree

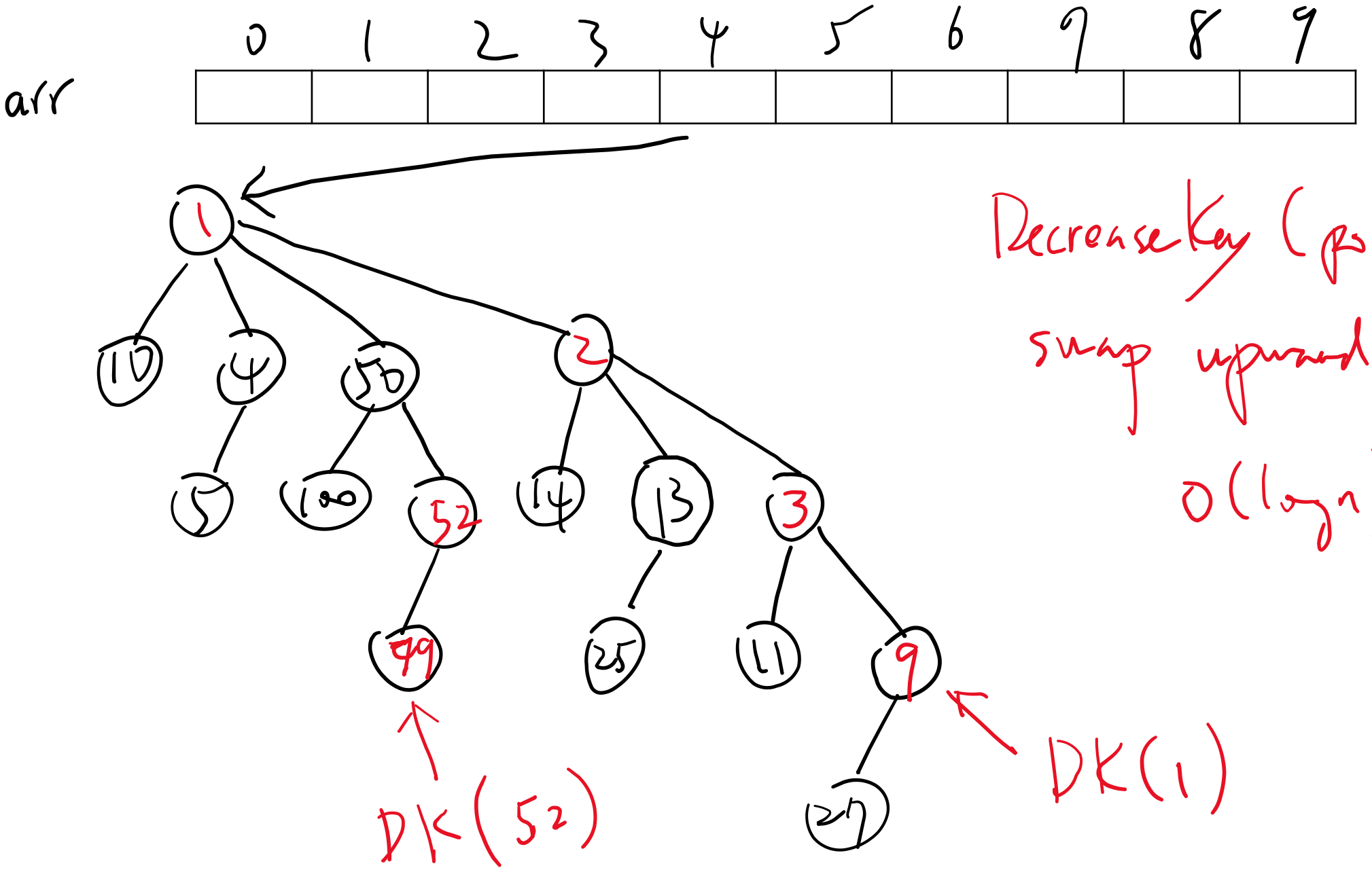


Insert (x):

$O(1)$ 1° create a single-node tree that stores x

2° Merge any 2 trees of the same size

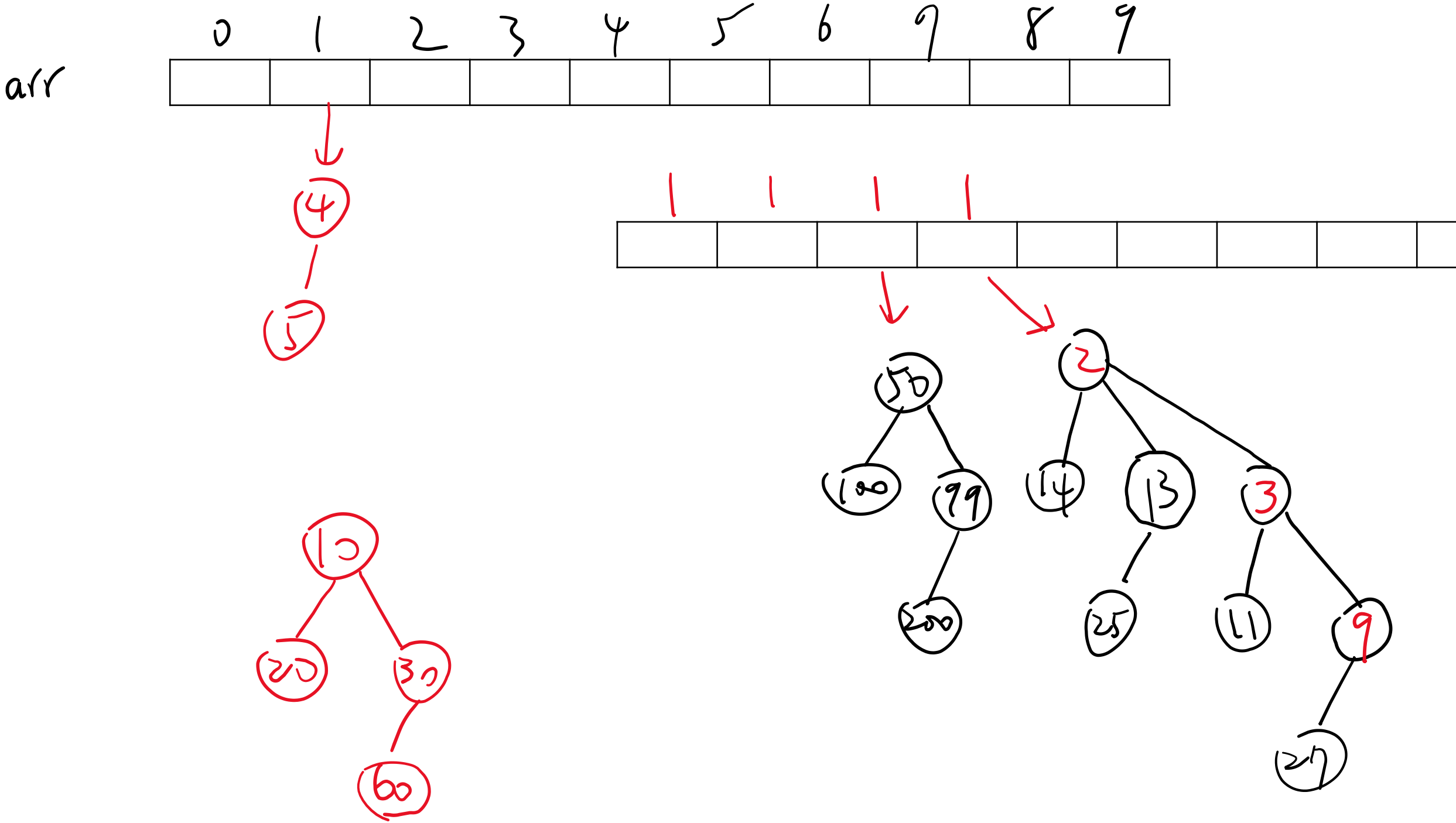
$O(\#\text{merge}) = O(\log n)$

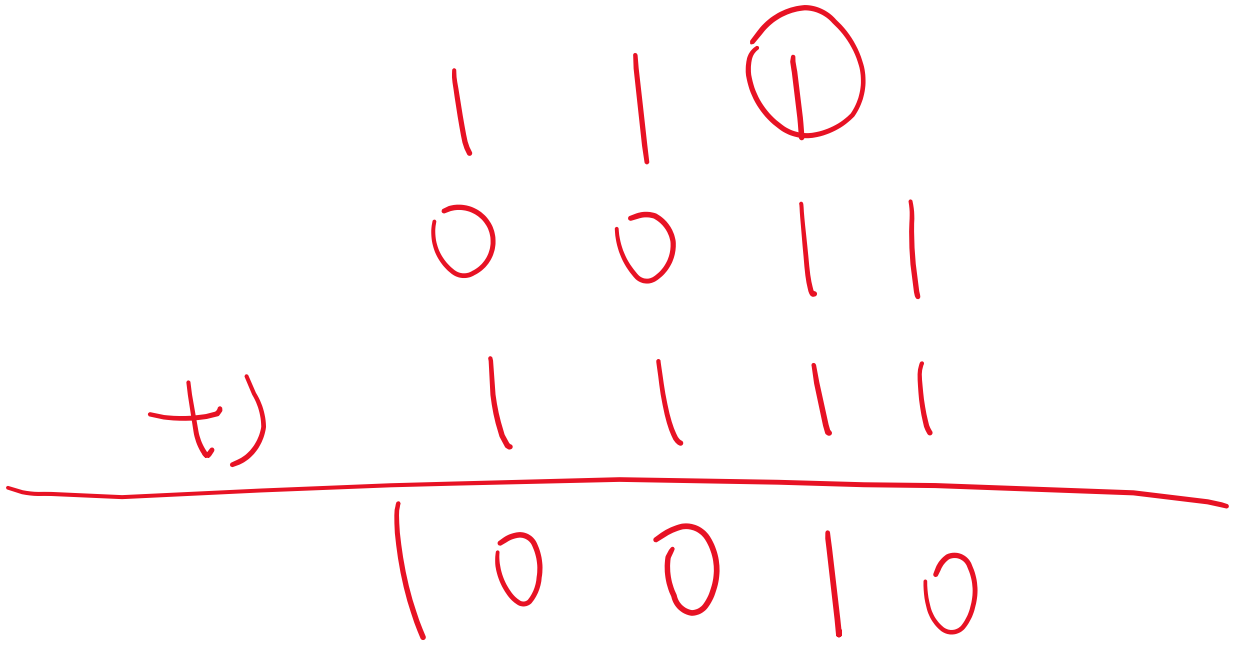


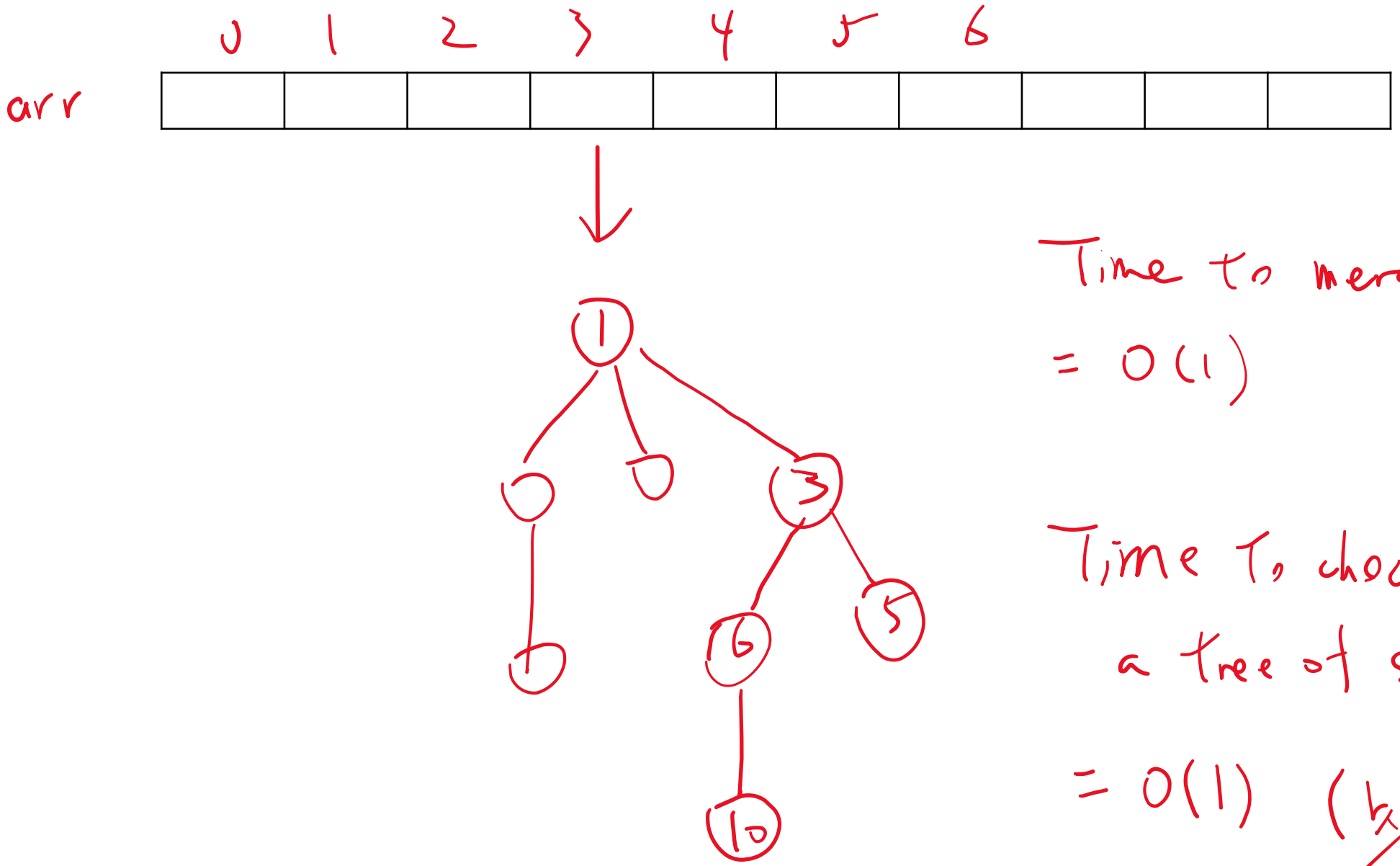
DecreaseKey (pos, newD):

swap upward if needed.

O(log n)


arr
0 1 2 3 4 5 6 7 8 9
4
5
1 1 1 1
50
100
99
200
2
14
13
25
3
11
9
27
10
20
30
60

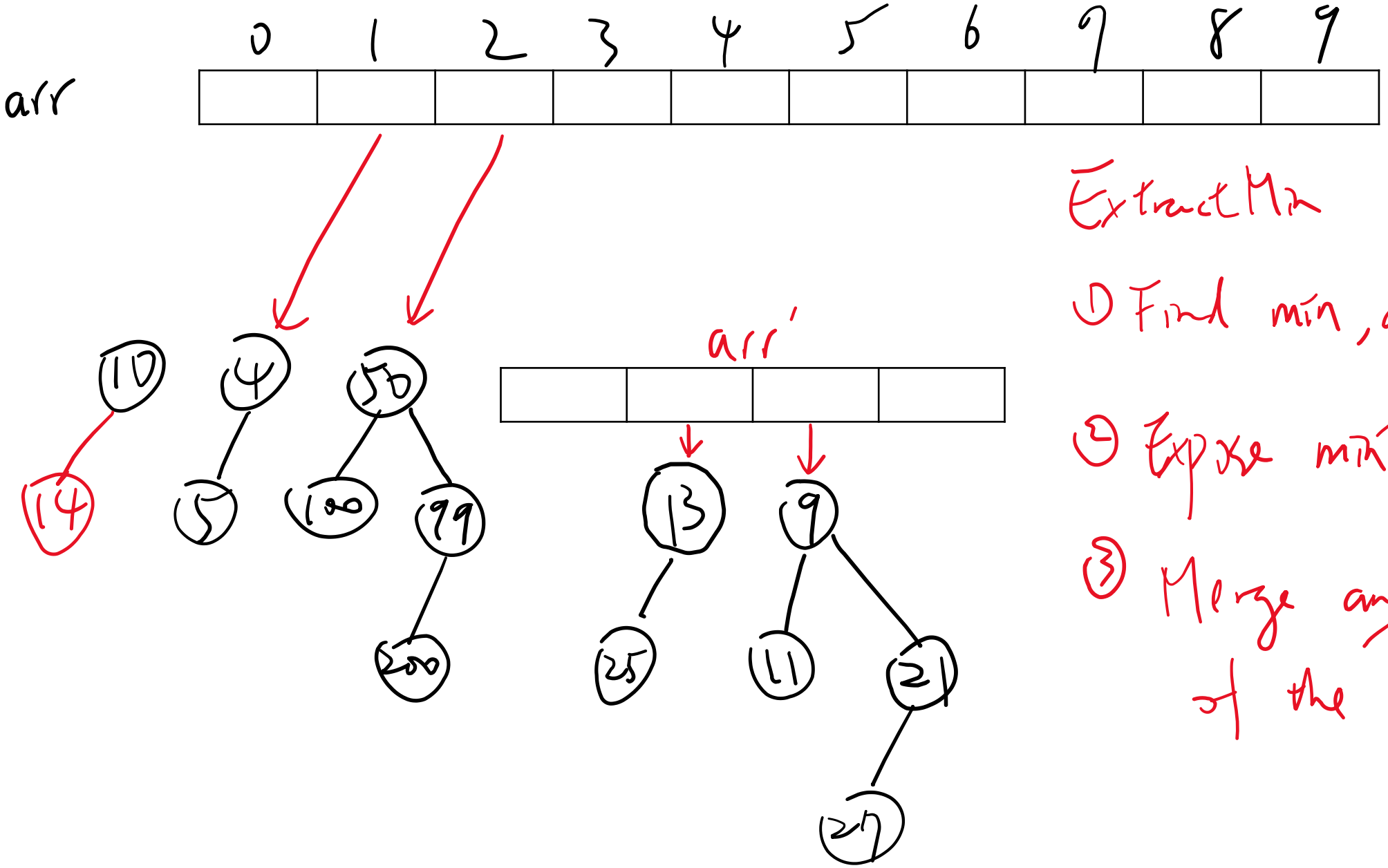




Time to merge 2 trees
$= O(1)$

Time To check whether
a tree of size 2^k exists

$= O(1)$ (by checking
arr[k])

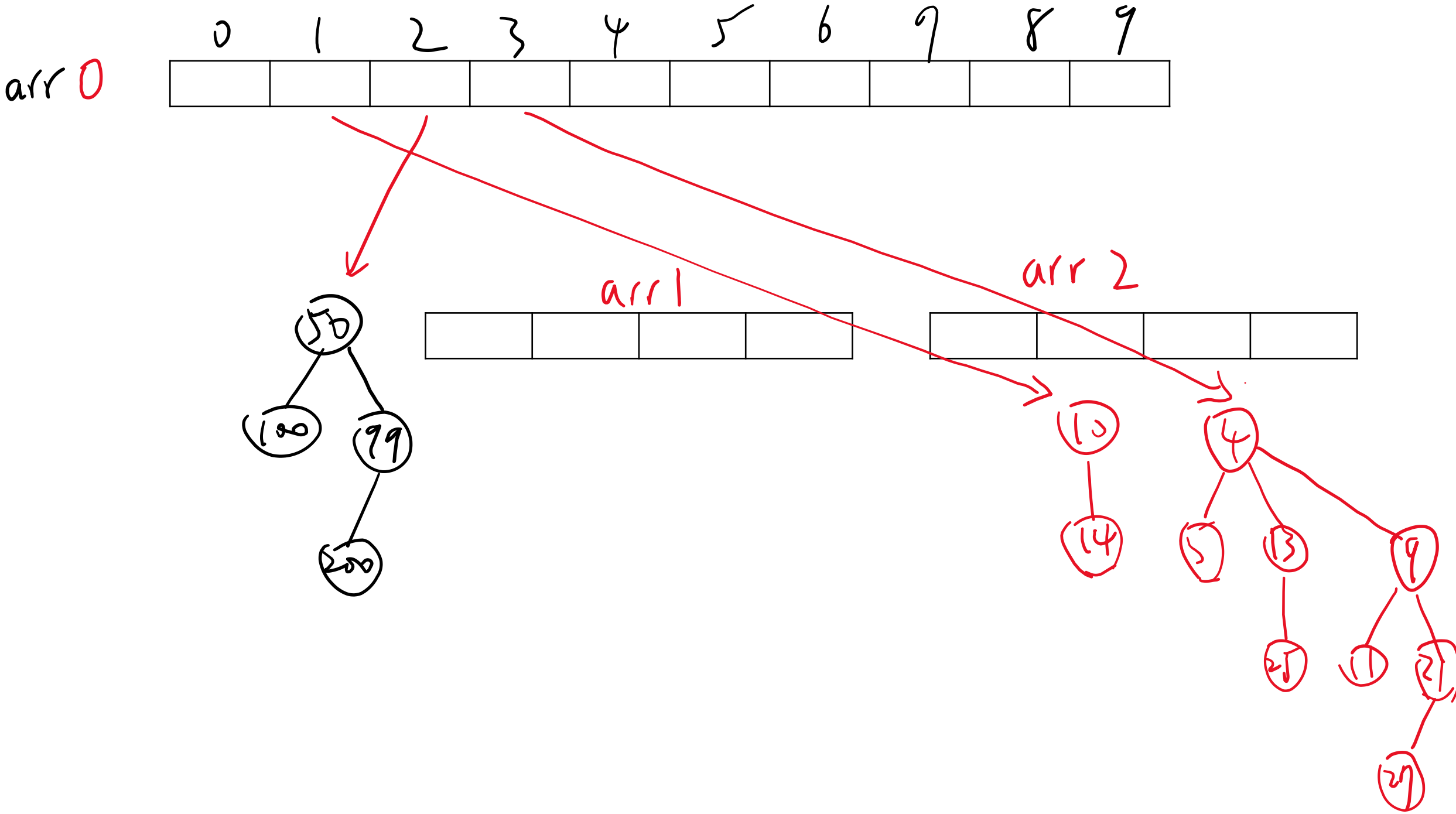


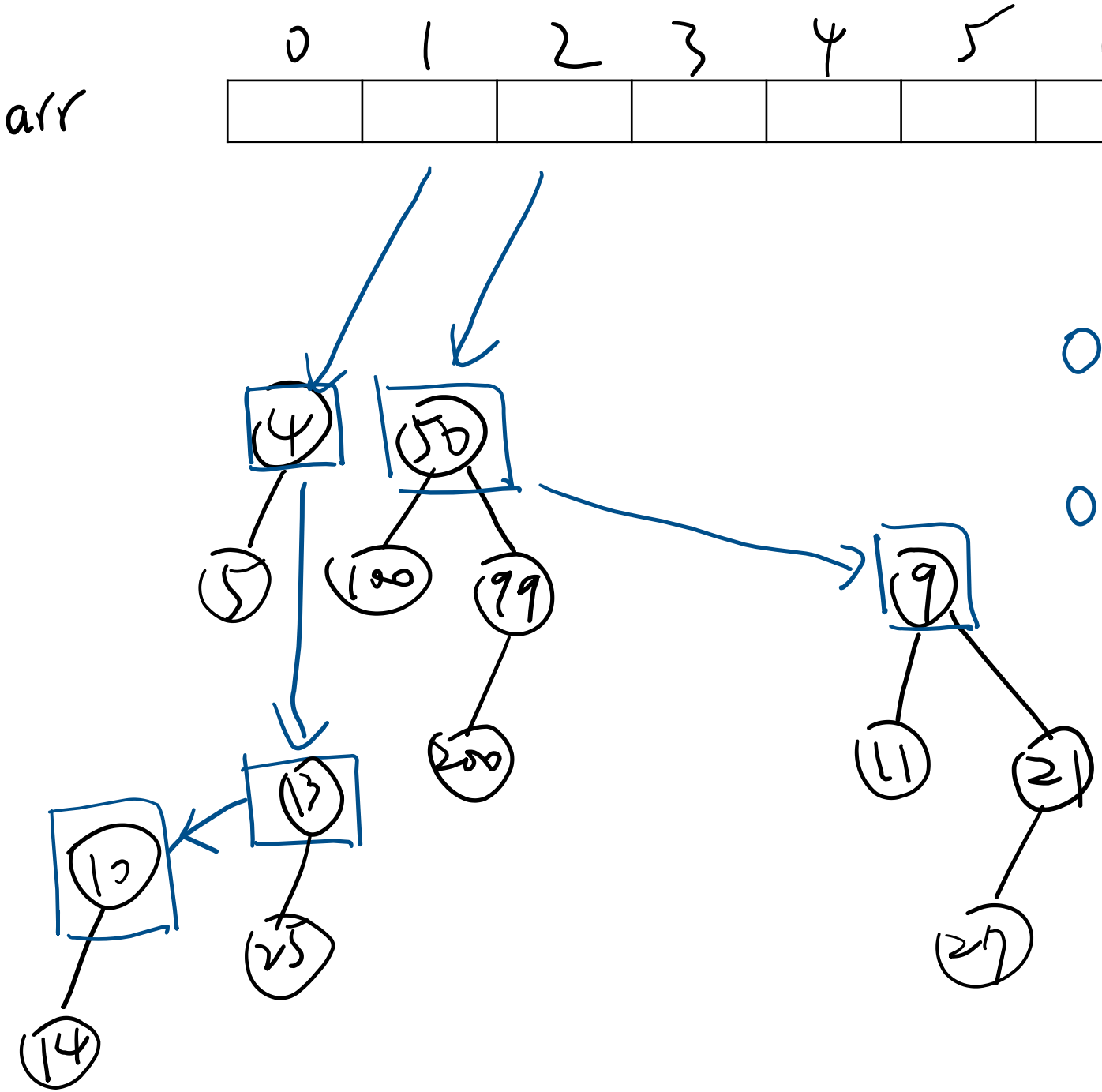
ExtractMin

① Find min, delete min,

② Expose min's children

③ Merge any 2 trees of the same size.


0 1 2 3 4 5 6 7 8 9
arr 0
arr 1
arr 2
50
100
99
200
10
4
14
5
13
9
27
1
21
27



ExtractMin

$O(\log n)$ ① Find min,

$O(\log n)$ ② Delete min'd
Expose min's children

③ Merge any 2 trees

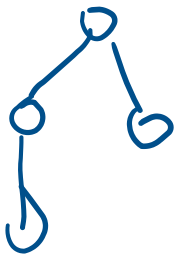
$O(\log n)$ of the same size.

000 | n=1

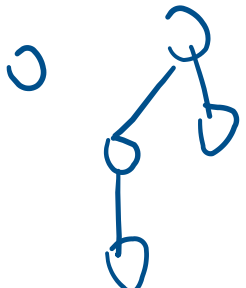
001 | n=2

010 | n=3

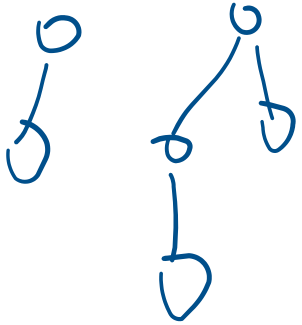
011 | n=4



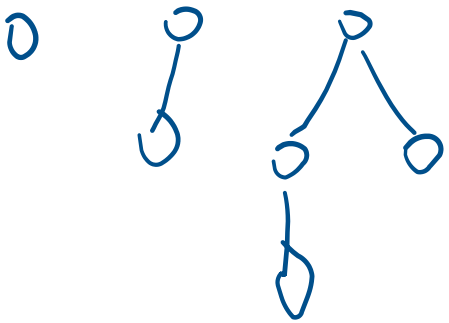
100 | n=5



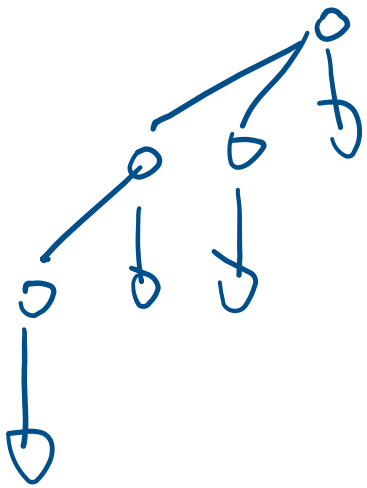
101 | n=6

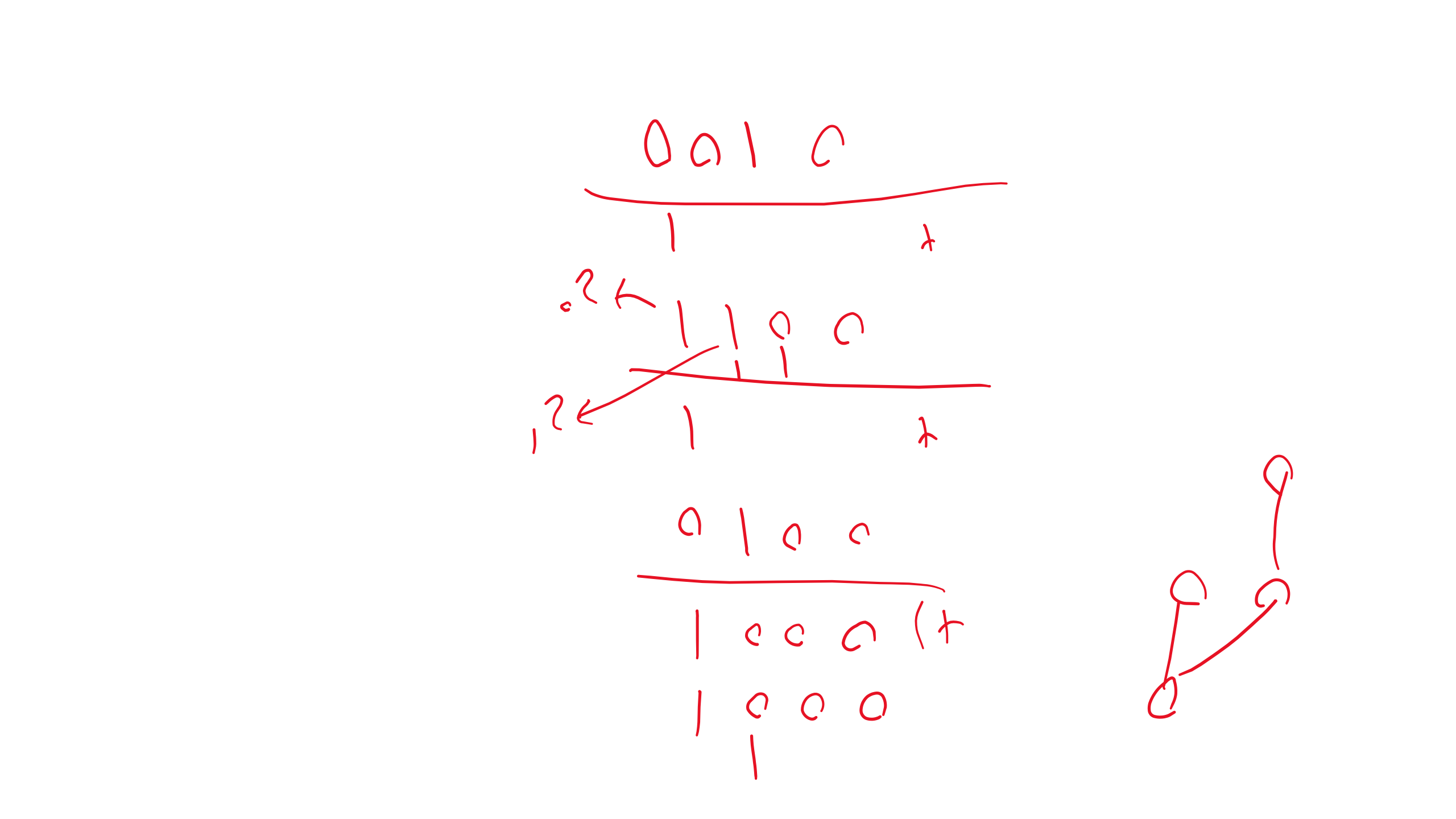


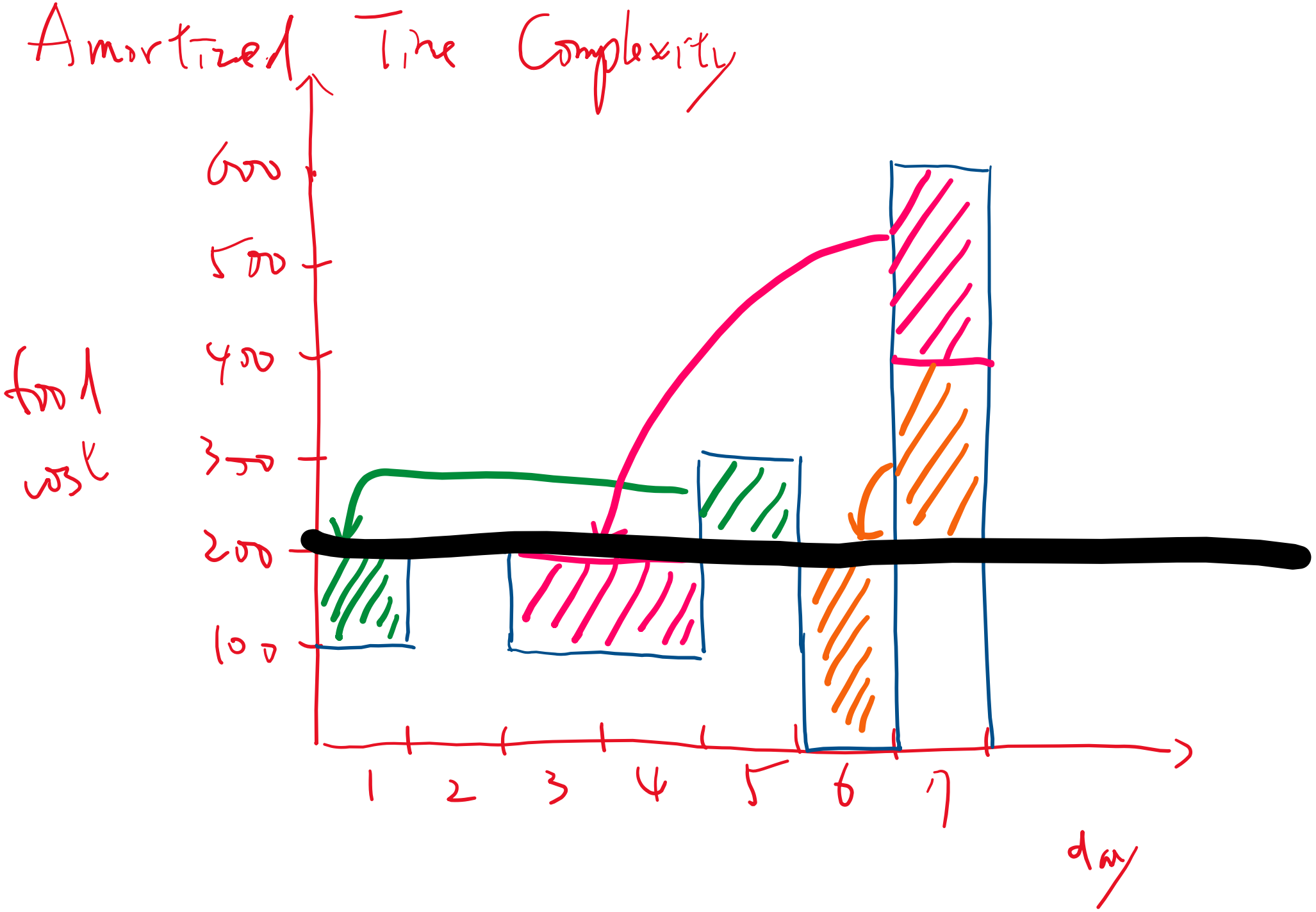
110 | n=7



111 | n=8

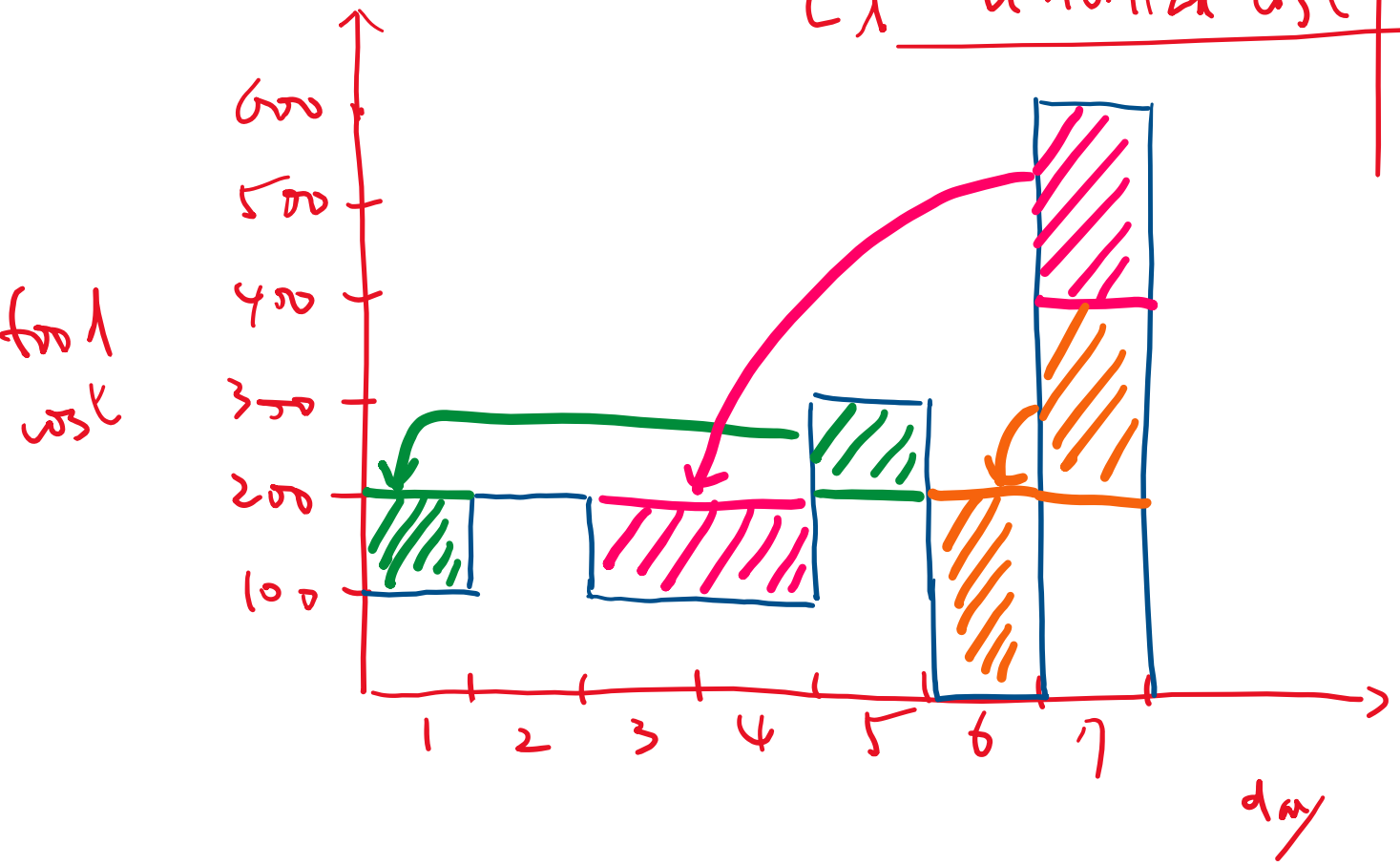





Amortized Time Complexity
600
500
400
300
200
100
1
2
3
4
5
6
7
day

$B_0 = 0$

	day	1	2	3	4	5	6	7
C_i	actual cost	100	200	100	150	300	0	600
B_i	$ in bank	100	150	200	300	200	400	0
$\overline{C_i}$	amortized cost	200	200					200



$$\overline{C_i} = C_i + (B_i - B_{i-1})$$

$$\overline{C_1} = 100 + (100 - 0)$$

$$\overline{C_2} = 200 + (100 - 100)$$

$$\overline{C_7} = 600 + (0 - 400)$$

T_i = actual time complexity of the i^{th} OP.

B_i = time in the time bank after the i^{th} OP ← potential function

$B_0 = 0$

$B_i \geq 0 \quad \forall i$

$\overline{T_i} = T_i + (B_i - B_{i-1})$

Amortized Time complexity

$= \dfrac{T_1 + T_2 + \cdots + T_n}{n}$

Goal: $\frac{T_1 + T_2 + \cdots + T_n}{n} \le c$ (e.g., $c = O(1)$ or $c = O(\log n)$)

It suffices to show $\boxed{\bar{T}_i \le c \quad \forall i}$

$\bar{T}_i = T_i + B_i - B_{i-1}$

$\because$ if $\bar{T}_i \le c \Rightarrow \bar{T}_1 \le c, \bar{T}_2 \le c, \cdots, \bar{T}_n \le c$

$B_n - B_{n-1}$

$B_{n-1} - B_{n-2}$

$B_{n-2} - B_{n-3}$

$\vdots$

$B_1 - B_0 = 0$

$\Rightarrow \frac{\bar{T}_1 + \bar{T}_2 + \cdots + \bar{T}_n}{n} \le c$

$\Rightarrow \frac{T_1 + \cdots + T_n + B_n - B_0}{n} \le c$

$$\frac{T_1 + T_2 + \cdots + T_n}{n} \leq \frac{T_1 + \cdots + T_n + B_n - \cancel{B_0}}{n} \leq C$$

if $B_n \geq 0$

Amortized Time Complexity of Insertion

B_i = # trees after the i^{th} insertion

(Make sure B_i is large befor a slow insertion)

T_i = Actual time complexity of the i^{th} insertion

= $\underbrace{O(1)}$ + #merge

↙ create a single-node tree

$$\overline{T_i} = T_i + (B_i - B_{i-1})$$

# merge	$B_i - B_{i-1}$
0	1
1	0
2	-1
3	-2
4	-3

$$\#\text{merge} + B_i - B_{i-1} = 1$$

$$\overline{T_i} = O(1) + 1 = O(1)$$

B_i = # trees after the i^{th} insertion

T_i = Actual time complexity of the i^{th} insertion

$= \underbrace{O(1)}_{\text{create a single-node tree}} + \#\text{merge}$

$$\overline{T_i} = T_i + B_i - B_{i-1}$$

$$= O(1) + \#\text{merge} + (B_i - B_{i-1})$$

$B_i = B_{i-1} + 1 - \#\text{merge}$

trees before the i^{th} insertion

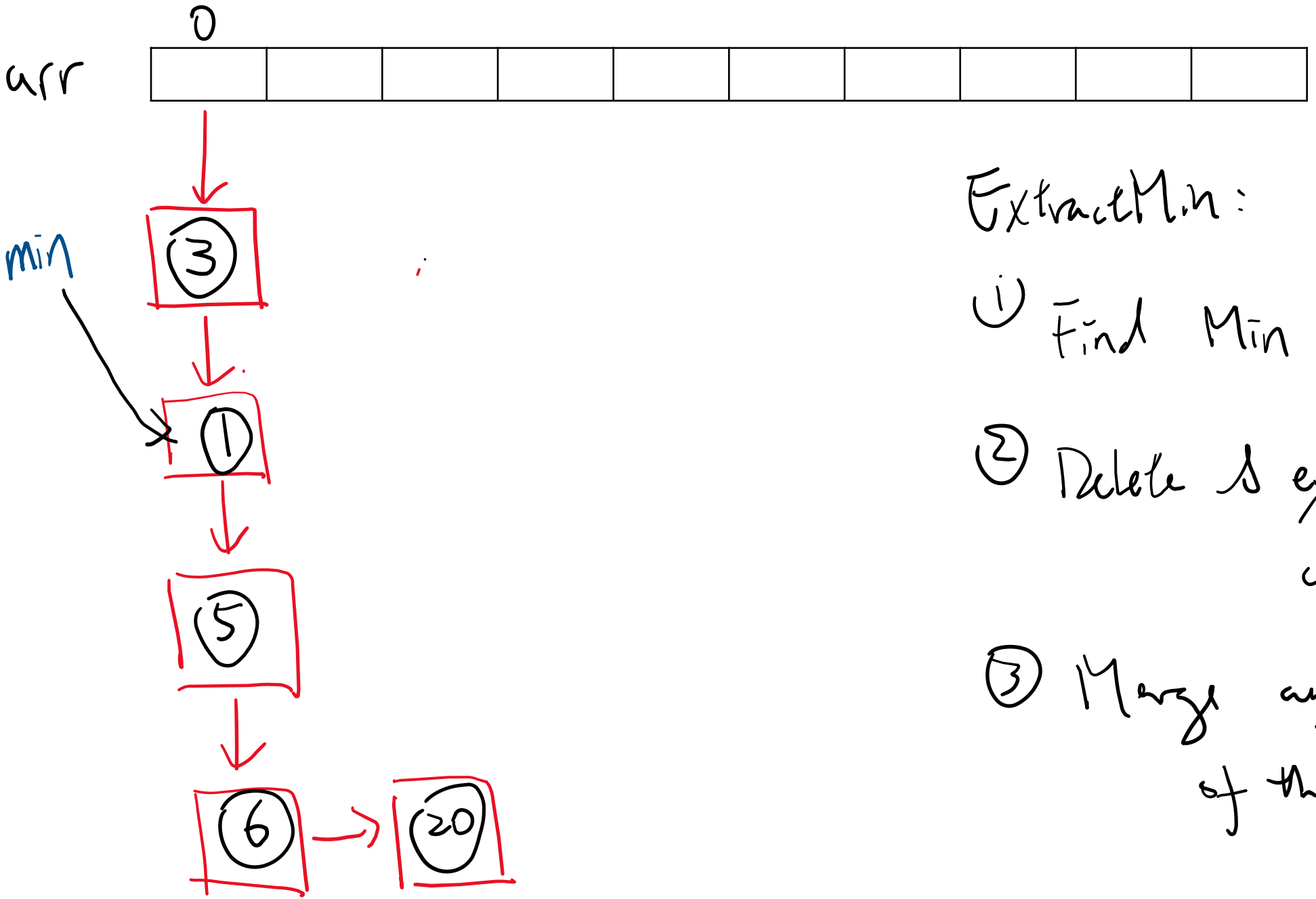
trees after the i^{th} insertion

create 1 single node tree

$$\#\text{merge} + B_i - B_{i-1} = 1$$

Lazy Binomial Heap

- No merge in insertion. Merge in ExtractMin
- Arr[i]: a "list" of roots (of size- 2^i tree)
- Maintain a min pointer after insertion, decreaseKey, extractMin

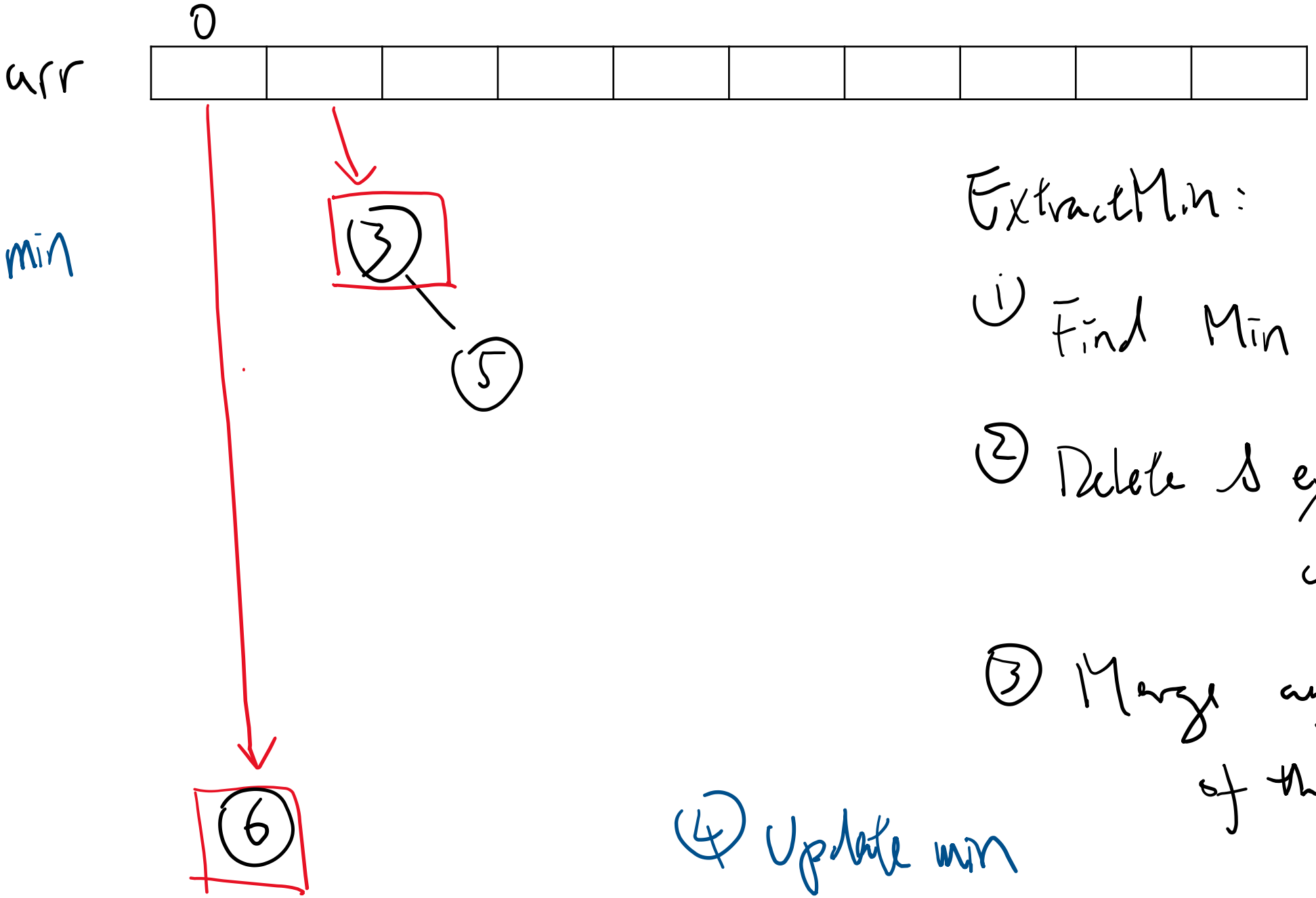


ExtractMin:

① Find Min

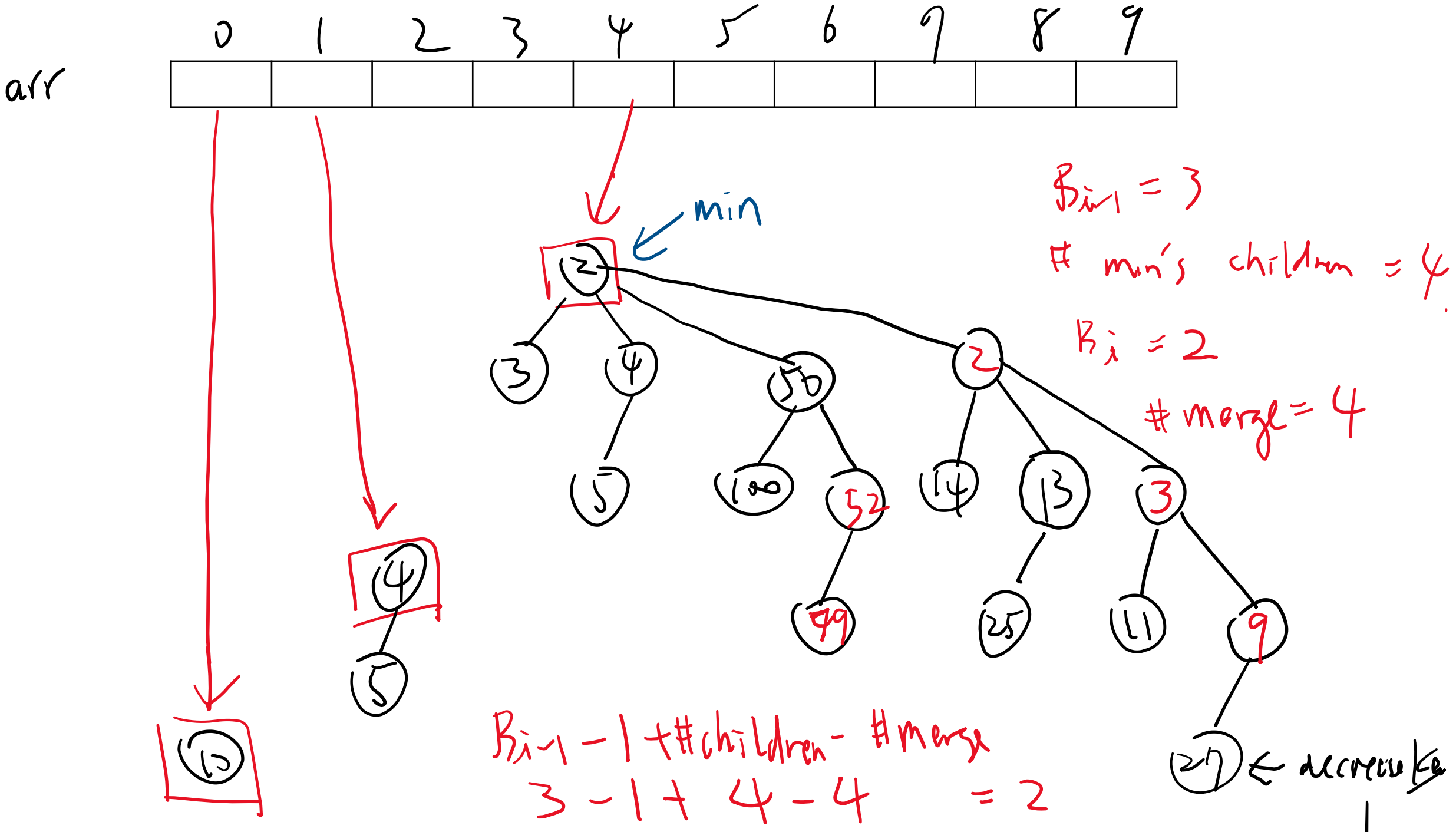
② Delete it & expose Min's children

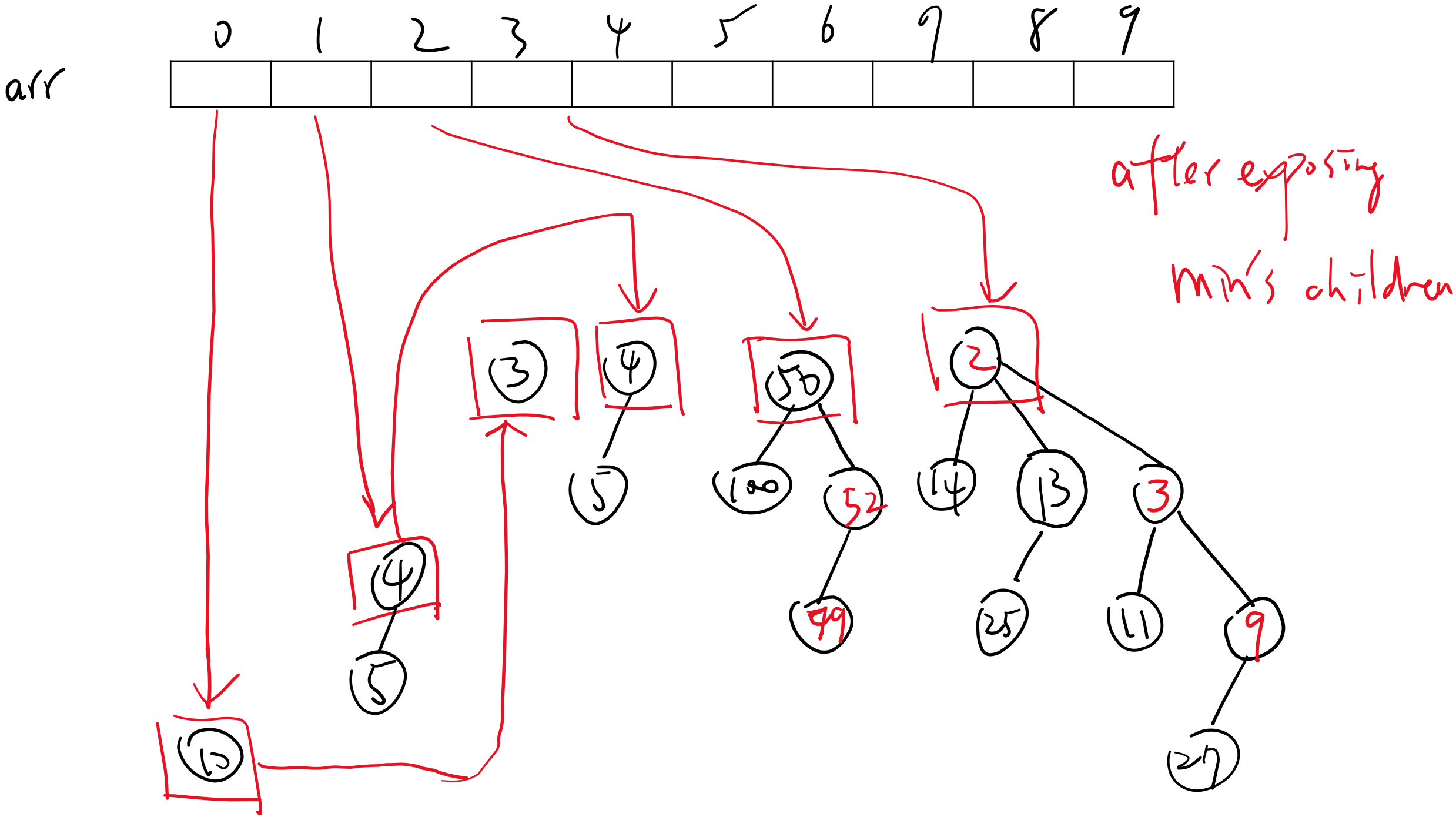
③ Merge any 2 trees of the same size.

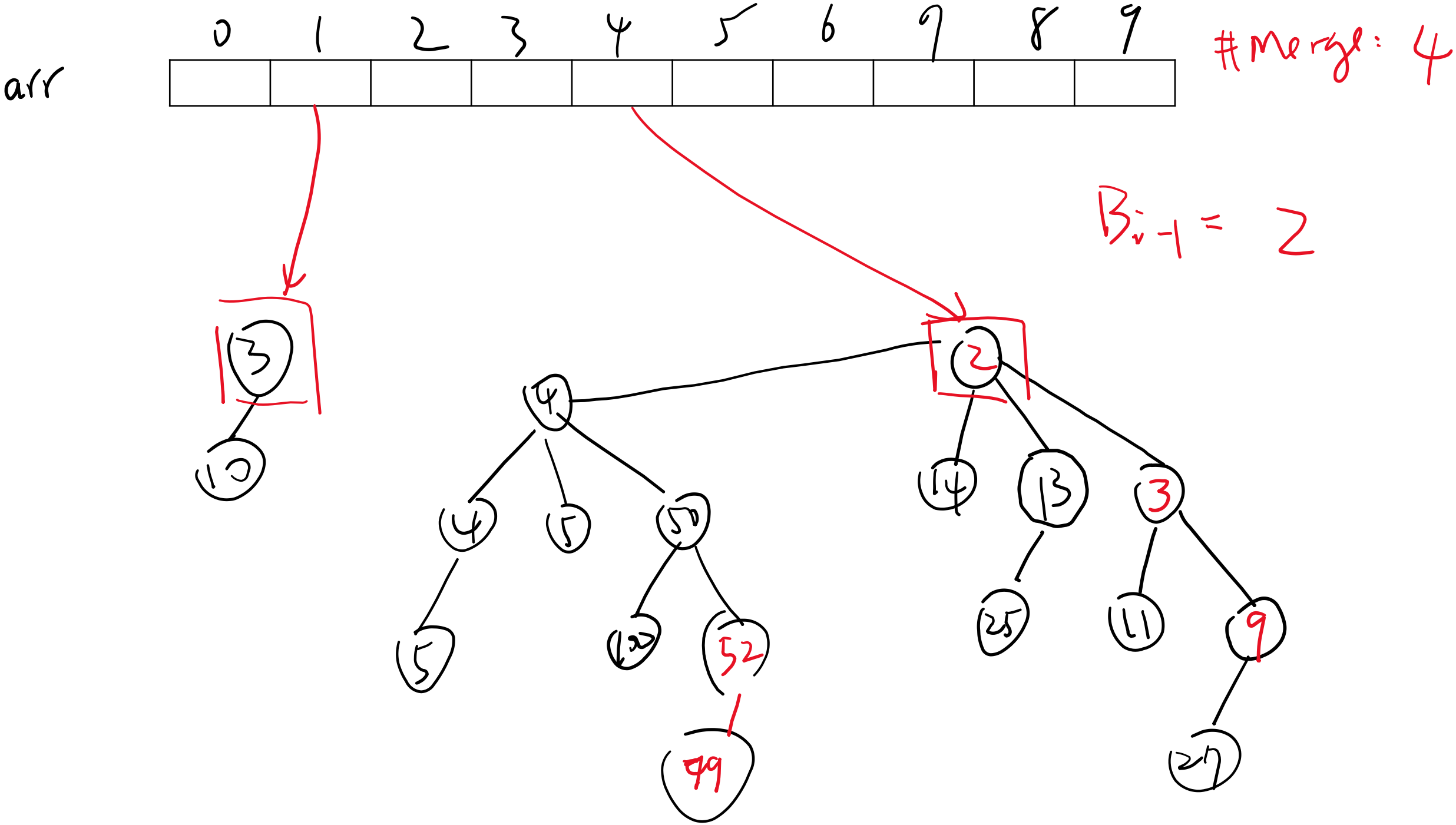


ExtractMin:

1. Find Min
2. Delete & expose Min's children
3. Merge any 2 trees of the same size.
4. Update min


arr
0 1 2 3 4 5 6 7 8 9
min
$B_{i-1} = 3$
min's children = 4
$B_i = 2$
merge = 4
$B_{i-1} - 1 + \#children - \#merge$
$3 - 1 + 4 - 4 = 2$
decrease key


arr
0 1 2 3 4 5 6 7 8 9
after exposing
min's children


arr
0 1 2 3 4 5 6 7 8 9
#Merge: 4
$B_{i-1} = 2$
3
10
4
5
5
50
20
52
99
2
14
13
3
25
11
9
27

Time Complexity

- Insert: 1° create a single-node tree $O(1)$
  2° store the address of the tree in arr[0] O(
  3° Update min $O(1)$

- DecreaseKey: swap upward if needed

  $O(\text{tree height})$

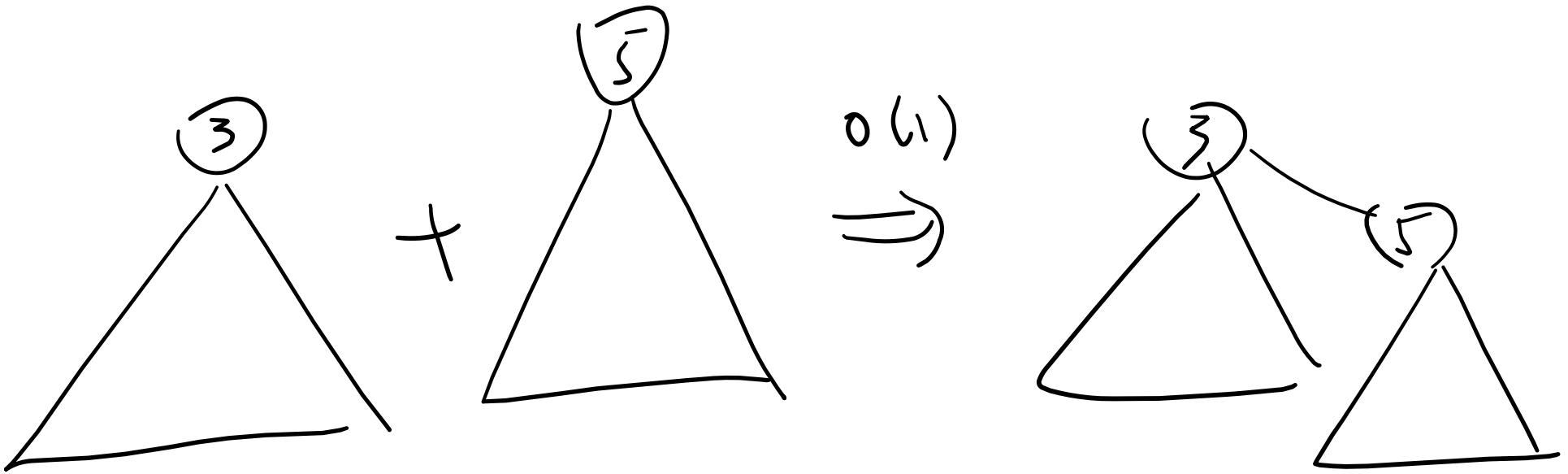
  $= O(\log n)$

ExtractMin: 1° Find Min $O(1)$ $O(\log n)$

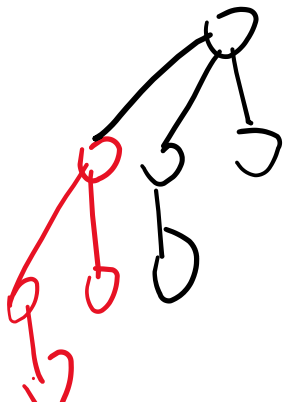
2° Expose Min's subtrees $O(\#\text{Min's children}) = O(\log n)$

$O(\text{size of arr}) + O(\#\text{merges}) = O(n)$ 3° Merge trees of the same size

$O(\log n)$ 4° Update Min



# nodes in a tree	# children of root	tree height
1	0	0
2	1	1
4	2	2
8	3	3
16	4	4
32	5	5
64	6	6
n	$\log n$	$\log n$
2^d	d	



B_i = # trees after the i^{th} OP.

T_i = $O(\log n)$ + # merges → actual time complexity of exactMin

$1+2+4+3$ (for scanning arr) 3

$B_i = B_{i-1} - 1 +$ # Min's children $-$ # merges

$B_i - B_{i-1} = O(\log n) -$ # merges $- 1$

$\overline{T_i} = T_i + (B_i - B_{i-1})$

$= O(\log n) +$ # merges $+ O(\log n) -$ # merges $- 1 = O(\log n)$

B_i = c# trees after the i^{th} OP.

$T_i \leq O(\log n)$ + c #merges (c is some constant)

$B_i = B_{i-1} - 1 +$ #Min's children $-$ #merges

$B_i - B_{i-1} = O(\log n) -$ #merges $- 1$

$\overline{T_i} = T_i + (B_i - B_{i-1})$

$= O(\log n) + c \cdot$ #merges $+ O(\log n) -$ #merges $- 1 = O(\log n)$

Assume $T_i \le O(\log n) + \underline{c \cdot \#\text{merges}}$ for some constant c.

n_i^{tree} : # trees after the i^{th} OP

$\underline{B_i} = \underline{c} \cdot \underline{n_i^{tree}}$

Mi's children

$n_i^{tree} = n_{i-1}^{tree} - 1 + O(\log n) - \#\text{merges}$

$n_i^{tree} - n_{i-1}^{tree} = O(\log n) - \#\text{merges} - 1$

$B_i - B_{i-1} = c \cdot n_i^{tree} - c \cdot n_{i-1}^{tree} = c\left(n_i^{tree} - n_{i-1}^{tree}\right) = O(\log n) - c \cdot \#\text{merges} - c$

$\overline{T_i} = T_i + B_i - B_{i-1} \le O(\log n) + c \cdot \#\text{merges} + O(\log n) - c \cdot \#\text{merges} - c = O(\log n)$

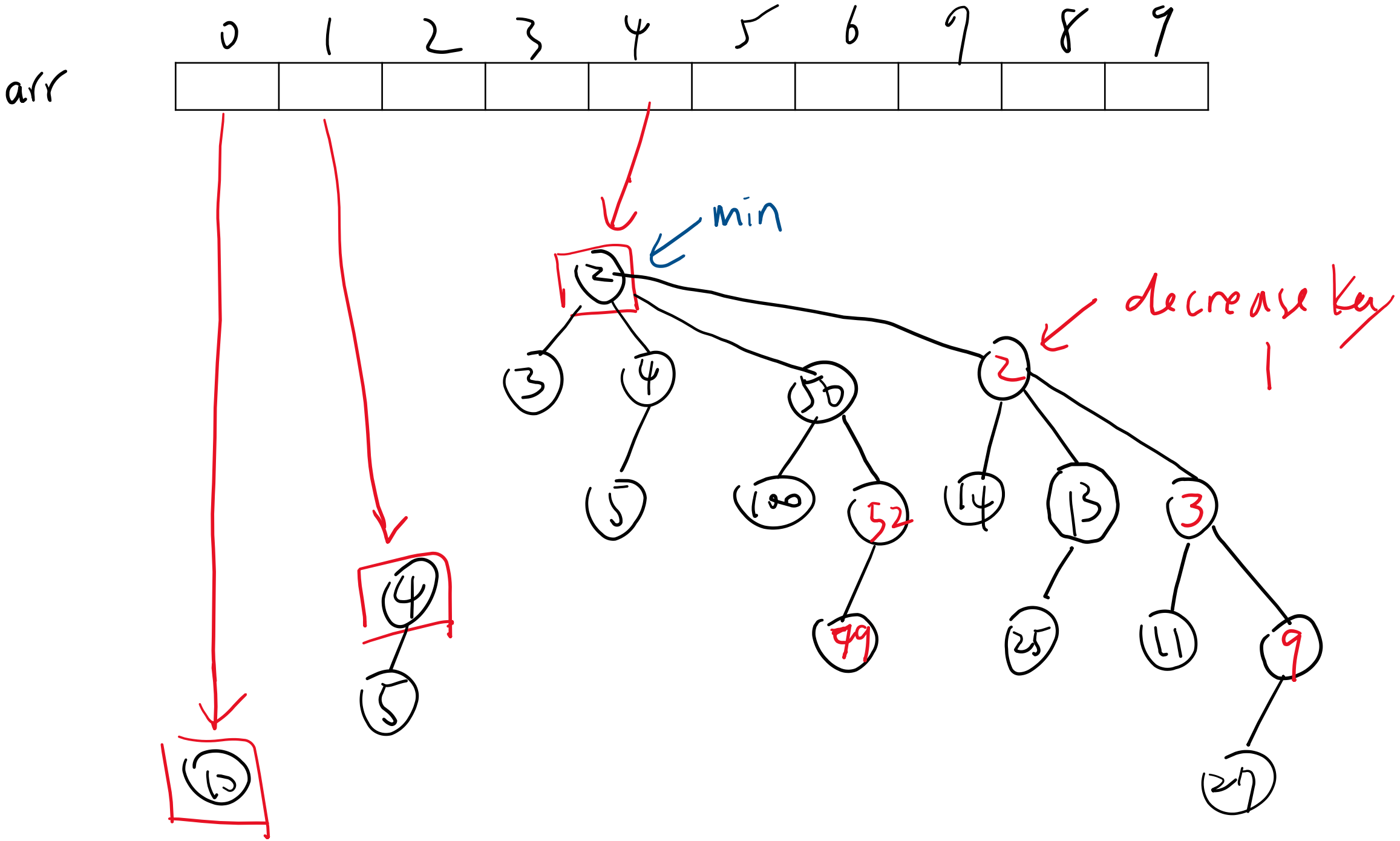
Fibonacci Heap: lazy binomial heap + O(1) decreaseKey
amortized

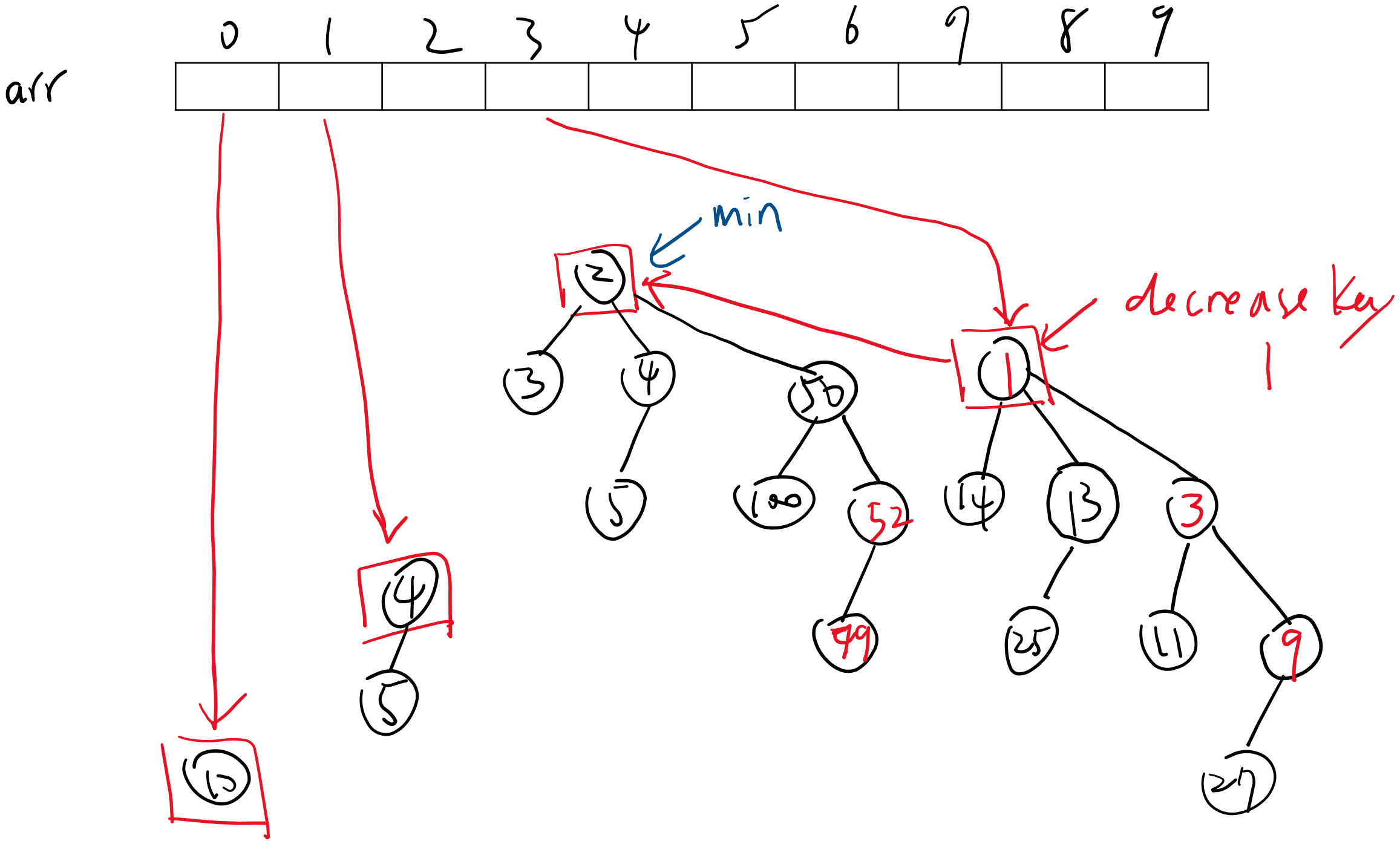
Key Idea 1: Expose / Cut the node when decreaseKey.

drawback: flat tree: $\Rightarrow$ order is large
$\Rightarrow$ expose is slow.

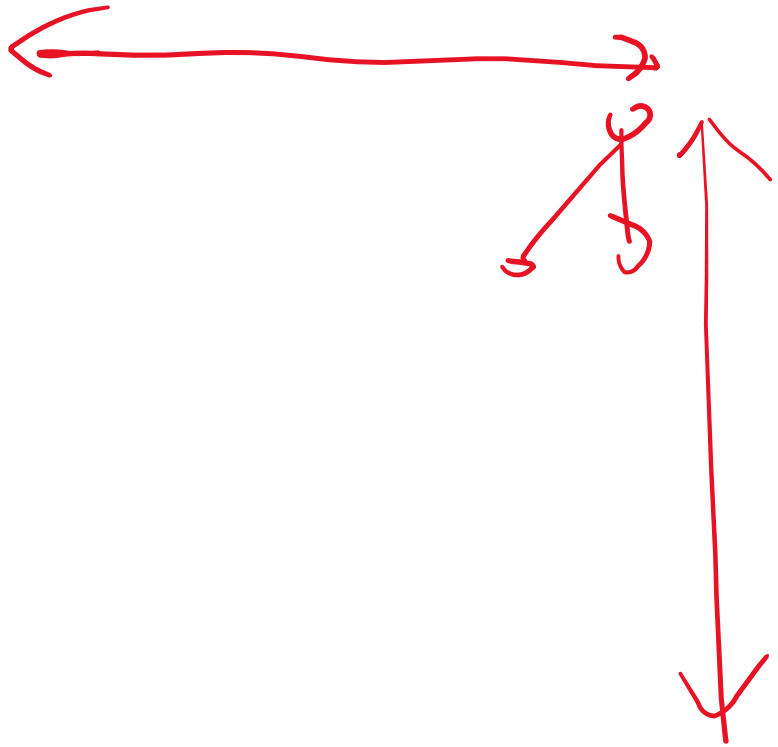
Key Idea 2: Merge trees of the same "order"

def: The order of a tree T is
the # children of T's root.

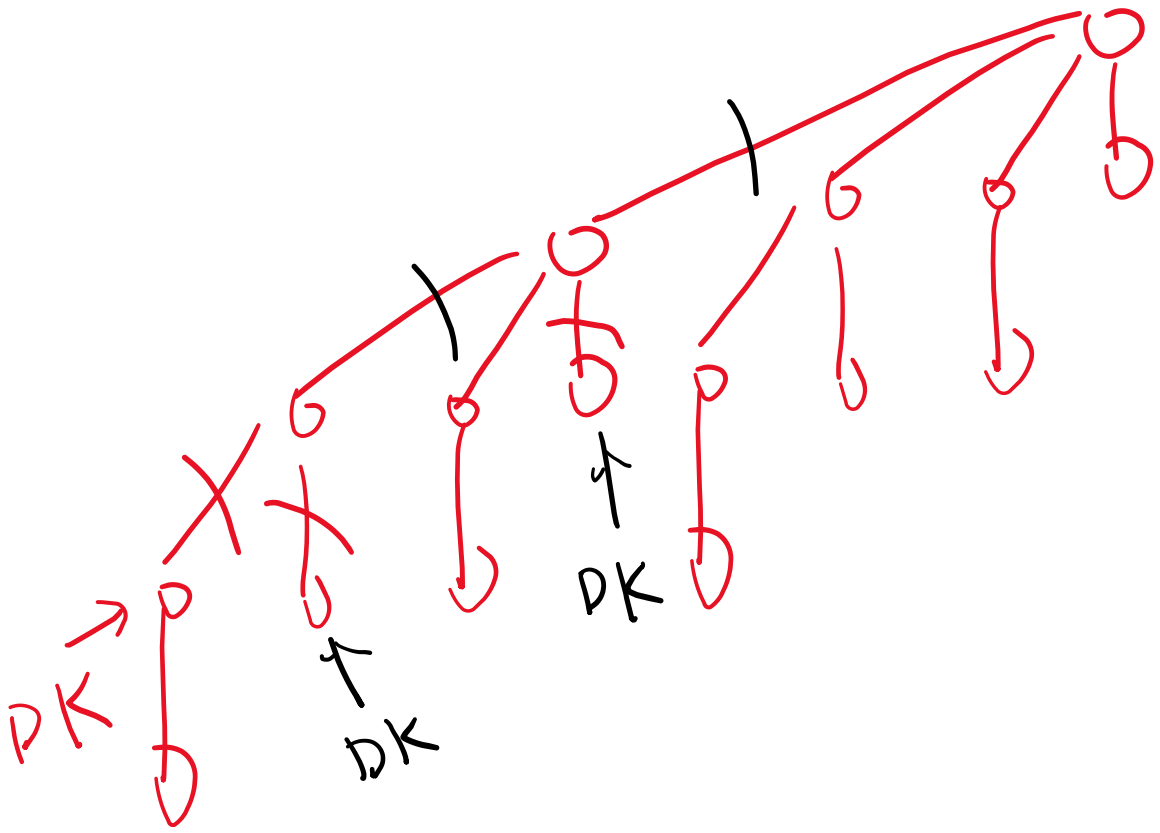

arr
0 1 2 3 4 5 6 7 8 9
min
decrease key
1


arr
0 1 2 3 4 5 6 7 8 9
min
decrease key
1

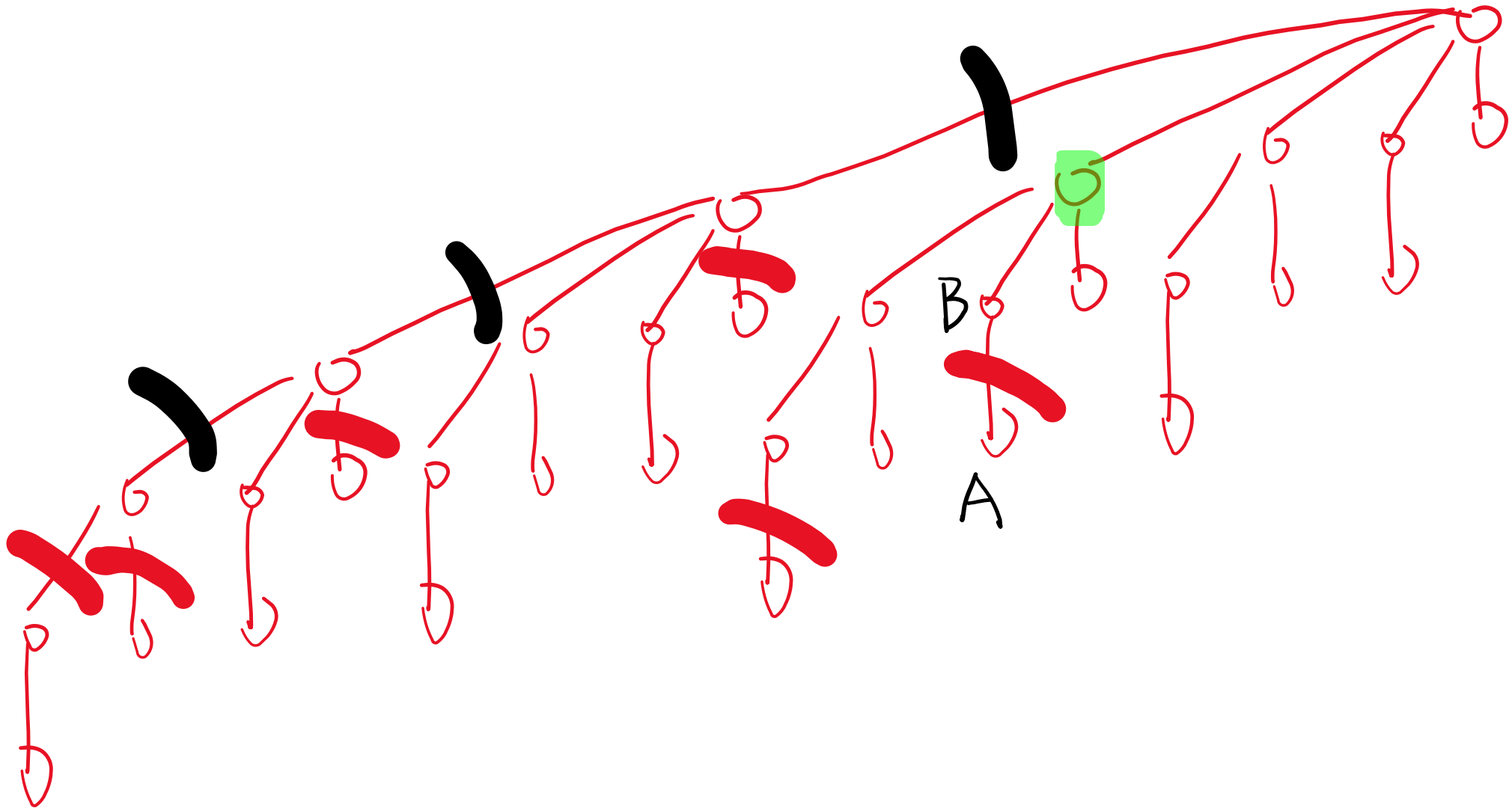
Flat tree



Key Idea 3: If 2 children of a node v is cut, cut v as well.



arr[i]: a list of trees of order i


B
A

Implementation:

1° Initially every node is "unmarked"

2° If v is cut:

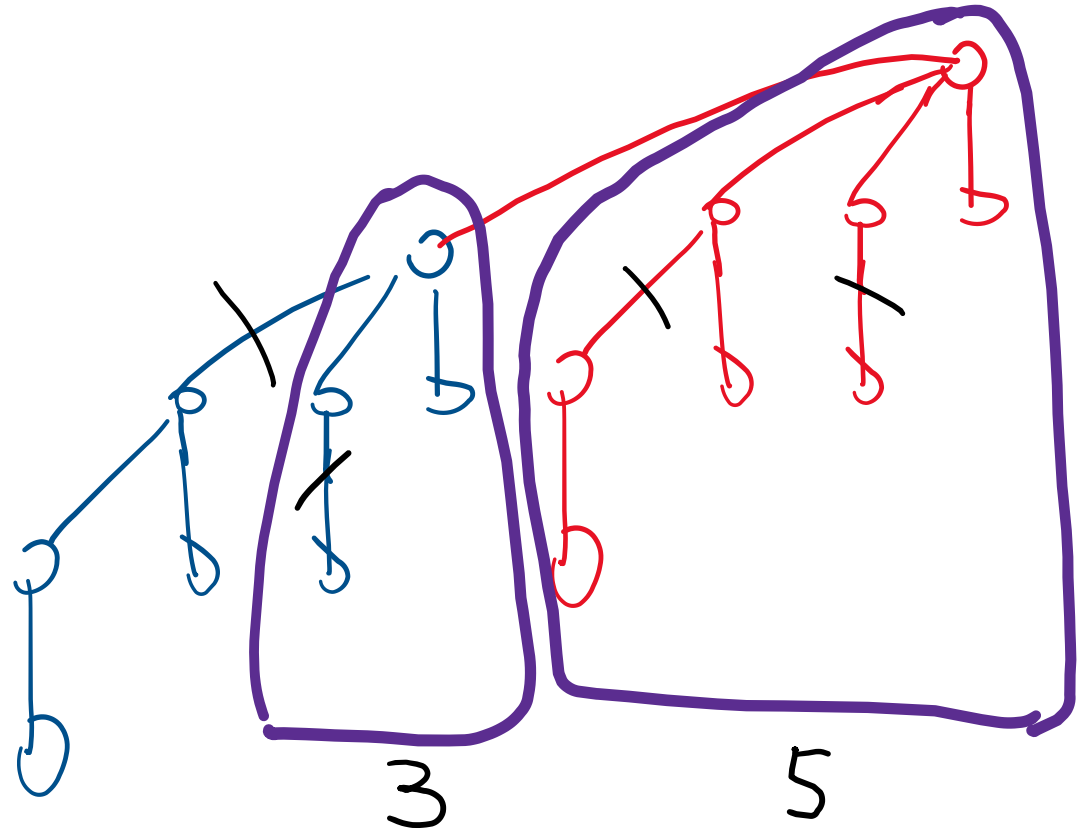
1° unmark v

2° if v's parent is unmarked,
mark v's parent

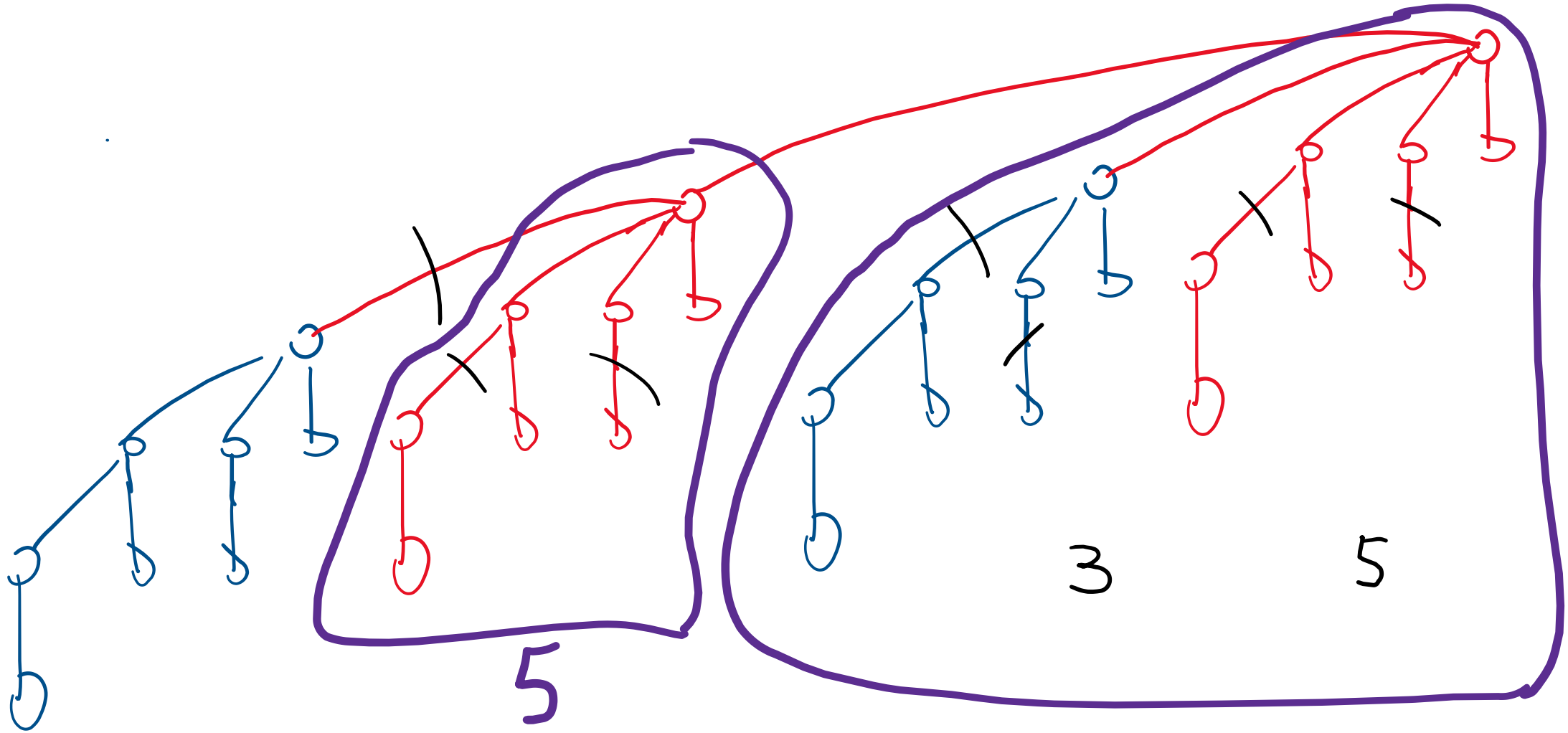
else: cut v's parent, unmark v's parent.

if v's parent is root
store v's parent in a "lower" list in arr

k	min # nodes in an order-k tree
0	1
1	2
2	3
3	5
4	8
5	13
k	$\Omega(1.61^k)$



order $= O(\log n) \Rightarrow$ extractMin in $O(\log n)$ amortized time


5
3
5

Time Complexity of Decrease Key

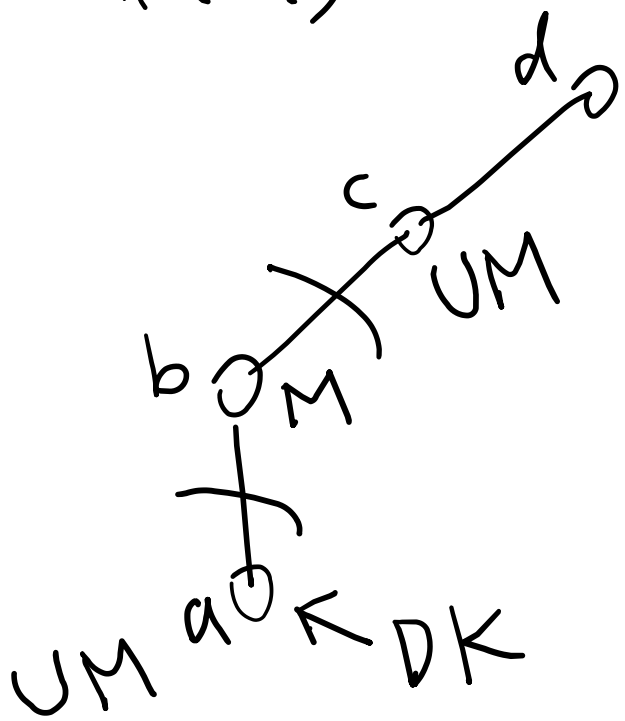
T_i = # cuts

B_i = $n_i^{tree} + n_i^{mark}$

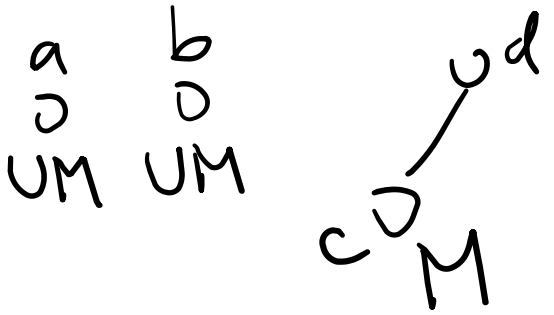
$n_i^{tree} = n_{i-1}^{tree} + \#\,cuts$

$n_i^{mark} = n_{i-1}^{mark}$

# cuts	$n_i^{mark} - n_{i-1}^{mark}$
1	0 or 1
2	-1 or 0



⇒

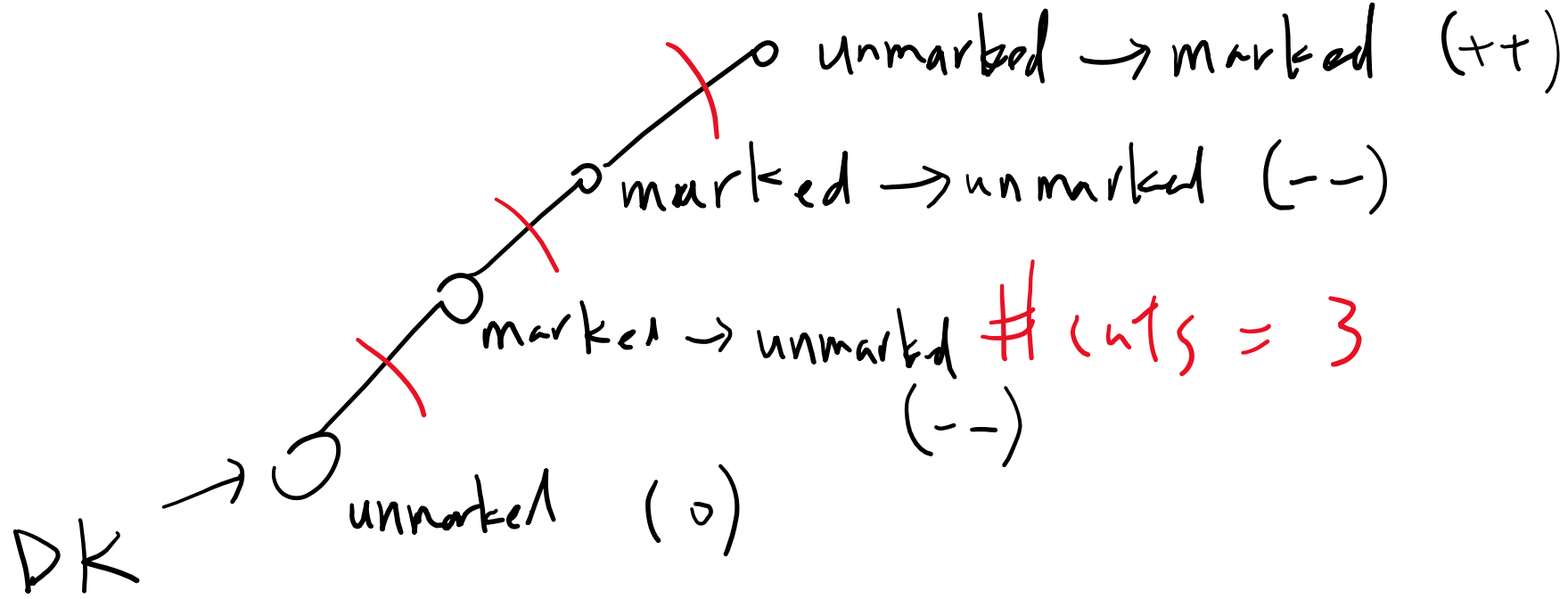


$n_i^{mark} - n_{i-1}^{mark} \leq 1 - (\#cuts - 1) = 2 - \#cuts$

↑ this is larger when the 1st cut vertex is unmarked.

the last marked node

↳ 1st cut



T_i = #cuts

$n_i^{tree} - n_{i-1}^{tree}$ = #cuts

$n_i^{mark} - n_{i-1}^{mark} \leq 2 - \#cuts$

× $B_i = n_i^{tree} + n_i^{mark}$

× $\overline{T_i} = T_i + (B_i - B_{i-1}) \leq \#cuts + \#cuts + (2 - \#cuts)$

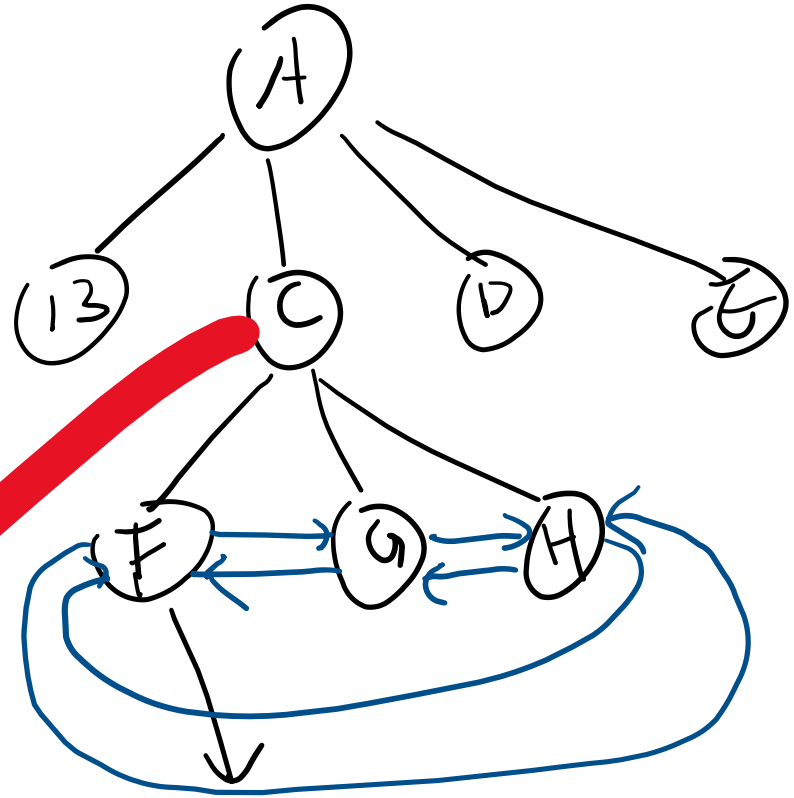
$= 2 + \#cuts$

$\beta_i = n_i^{tree} + 2 n_i^{mark}$

$\overline{T_i} = T_i + (\beta_i - \beta_{i-1}) \leq \#cuts + \#cuts + 2(2 - \#cuts)$

$= O(1)$

```
Struct node {
    int data // priority, distance
    int order
    bool isMarked
    Struct node* parent
    list<struct node*> children → removing a
                                  child takes
                                  O(log n) time
}
```



child : F

left_sibling : H

right_sibling : G

```
Struct node {
    int data // priority, distance
    int order
    bool isMarked
    Struct node* parent, child, left_child, right_child
}
```

↓

removing a child takes O(1) time

```
RemoveFromTree(v,x){
    if (v->parent->child == v) {
        v->parent->child = v->right_sibling
    }
    v->right_sibling->left_sibling = v->left_sibling
    v->left_sibling->rightsibling = v->right_sibling
    // cut, mark in
}
```

① Set

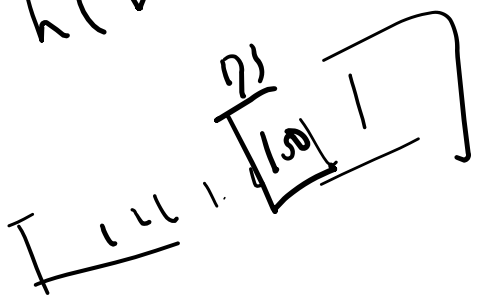
a collection of data

distinct

{3, 6, 10, 12}

h(data) → hashed value

e.g., h(10) = 93



② Map

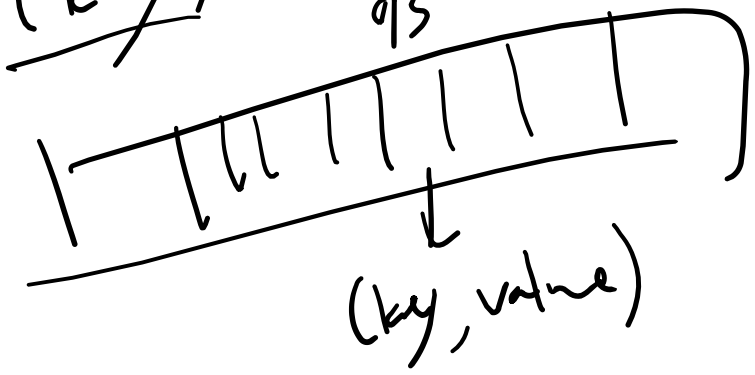
a collection of key-value pairs

{(3, 10), (5, 7), ...}

key value

key ← value

h(key) = 93



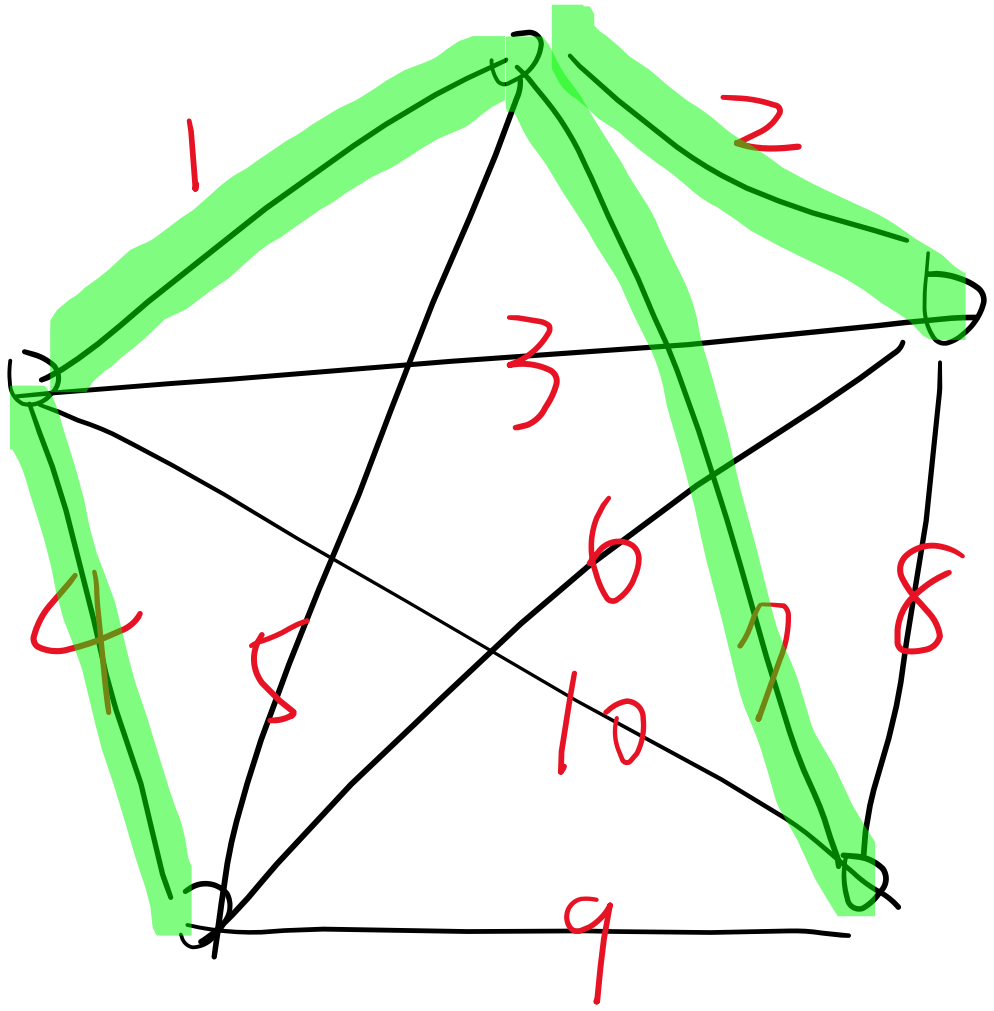
Minimum Spanning Tree

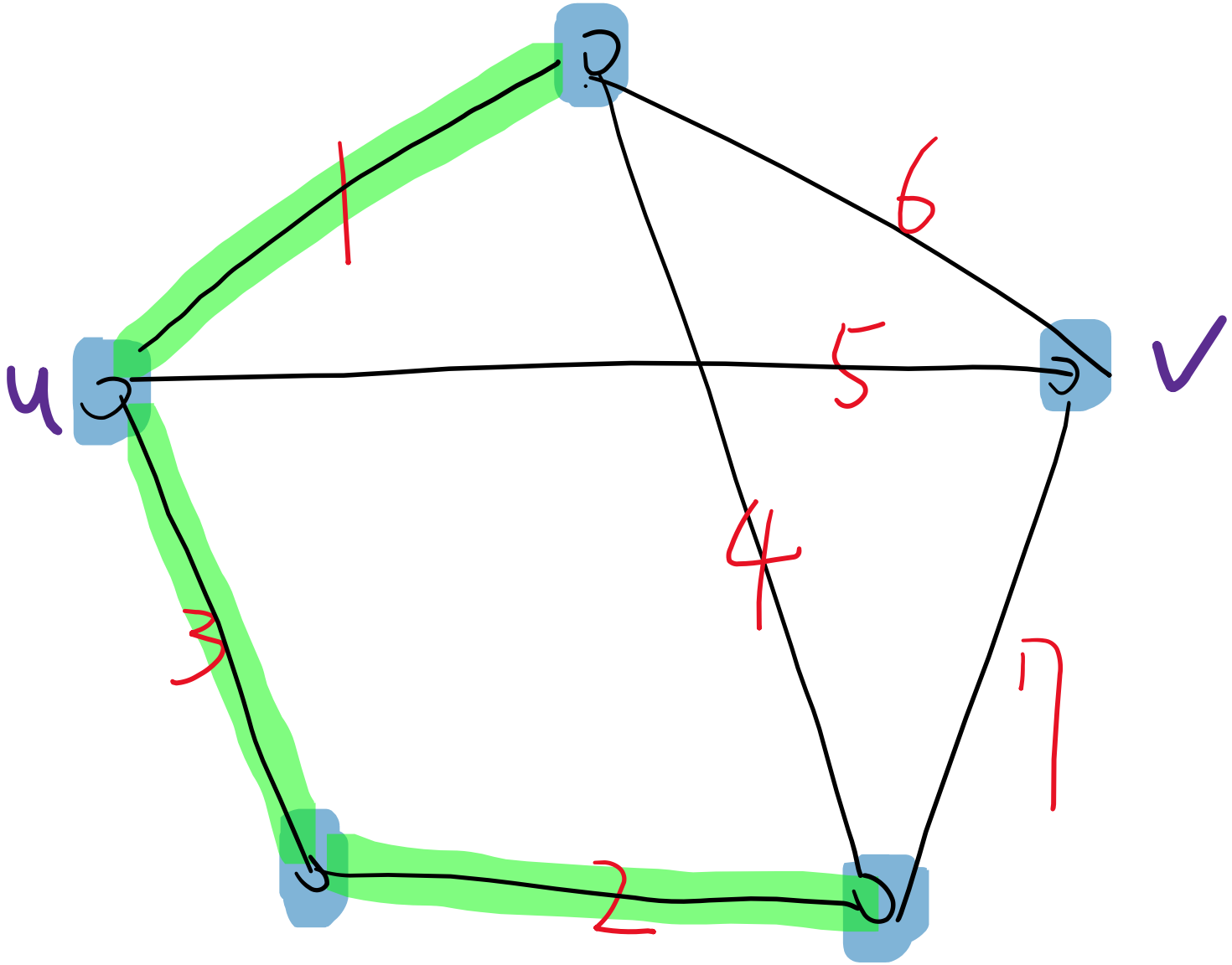
Input: An edge-weighted graph $G=(V,E)$

output: a tree T of G such that:

1. T contains V
2. total edge weight is minimized.

Kruskal's Algorithm




u
v
1
2
3
4
5
6
7

1° Sort edges in non-decreasing order of their weight.

$m \log m$

$= O(m \log n^2)$

$= O(m \log n)$

Let $w(e_1) \le w(e_2) \le \cdots \le w(e_m)$

2° For ($i = 1$ to m) {

Let $(u, v) = e_i$

If (u, v) does not create a cycle,

add (u, v) to the minimum spanning tree T

}, return T if T contains $n-1$ edges.

1° Sort edges

2° Initially, every node is a tree MakeSet(v) ∀ v in V

3° For (i = 1 to m) {

(u,v) = e_i

if u and v are in different trees, Find(u) ≠ Find(v)

add (u,v) to T & Union(u,v)

return T if T has n-1 edges.

}

Method 1: Store a collection of Trees
as a set of graphs.

To check if u & v are in the same tree:

$O(n)$ 1° Find the graph that contains u
∵ there are $O(n)$ trees

$O(n)^2$ ---------- v

$O(1)$ 3° See if these 2 graph are the same.

Method 2: Store the vertex set of a tree T_i in S_i

And then store $S_1, S_2, S_3, \cdots$ in a

disjoint sets data structure.

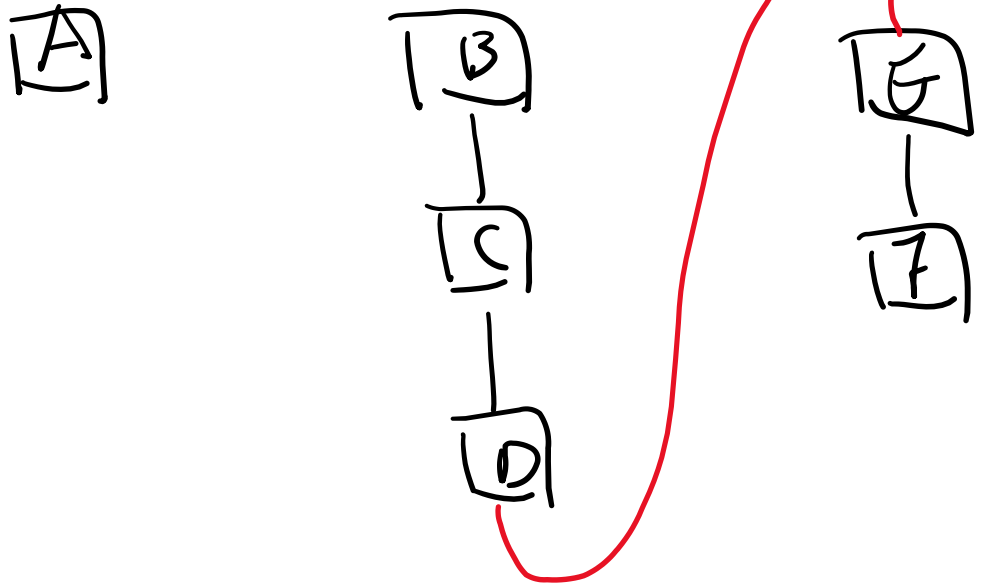
$O(n)$ 1° MakeSet(v): create a set that consists of v

$O(n)$ 2° Union(u,v): union the sets that contains u & v.

$O(m)$ 3° Find(u): return the "id" of the set that contains u.

Method A: Store each set in a linked list.

{A}, {B, C, D}, {E, F}



Union(u,v): O(n)

Union(F, D)

concatenate two lists

O(1) MakeSet(v): create a list that contains v

O(n) Find(v): return the head of the list containing v

Find(D) = B

Time Complexity of Kruskal's Algorithm.

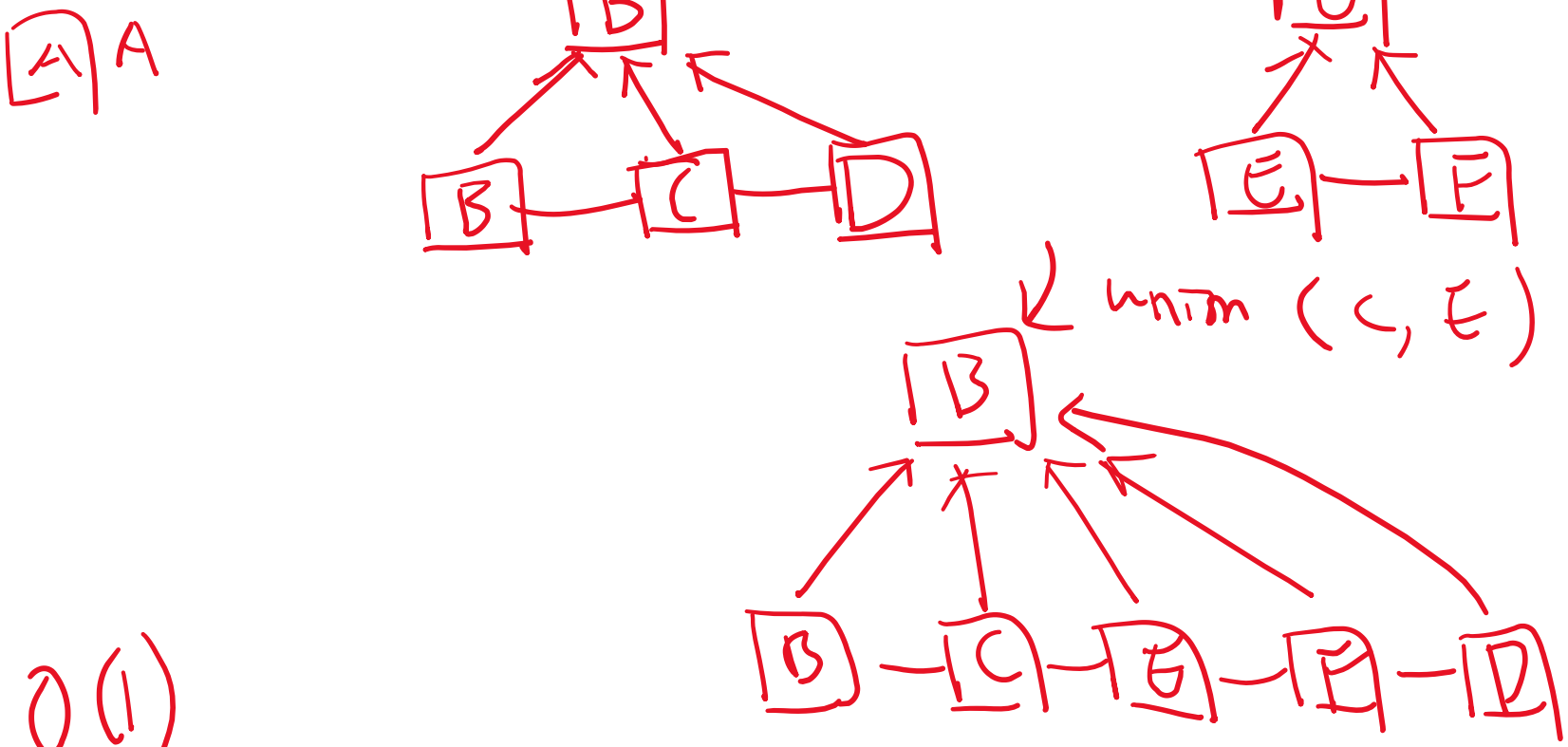
$$O(m \log n) + O(1) \times n + O(n) \times O(n) + O(n) \times O(m)$$

- $O(m \log n)$: Sort edges
- $O(1) \times n$: MakeSet
- $O(n) \times O(n)$: union
- $O(n) \times O(m)$: find

$$= O(nm)$$

Method B: Method A + every node stores the head.

{A} {B, C, D} {E, F}



Find(u): O(1)

Union: O(size of the smaller set)

1° Let S_u be the list that contain u

S_v contain v

if $|S_u| \geq |S_v|$, then {

S_v's head = S_u's head

Update the head pointers in S_v

}

Key Observation: Once v's head pointer is updated, the size of v's set is increased by a factor of ≥ 2.

$\Rightarrow$ Every node updates its head pointer at most $O(\log n)$

$\Rightarrow$ total head updates $= O(n \log n)$

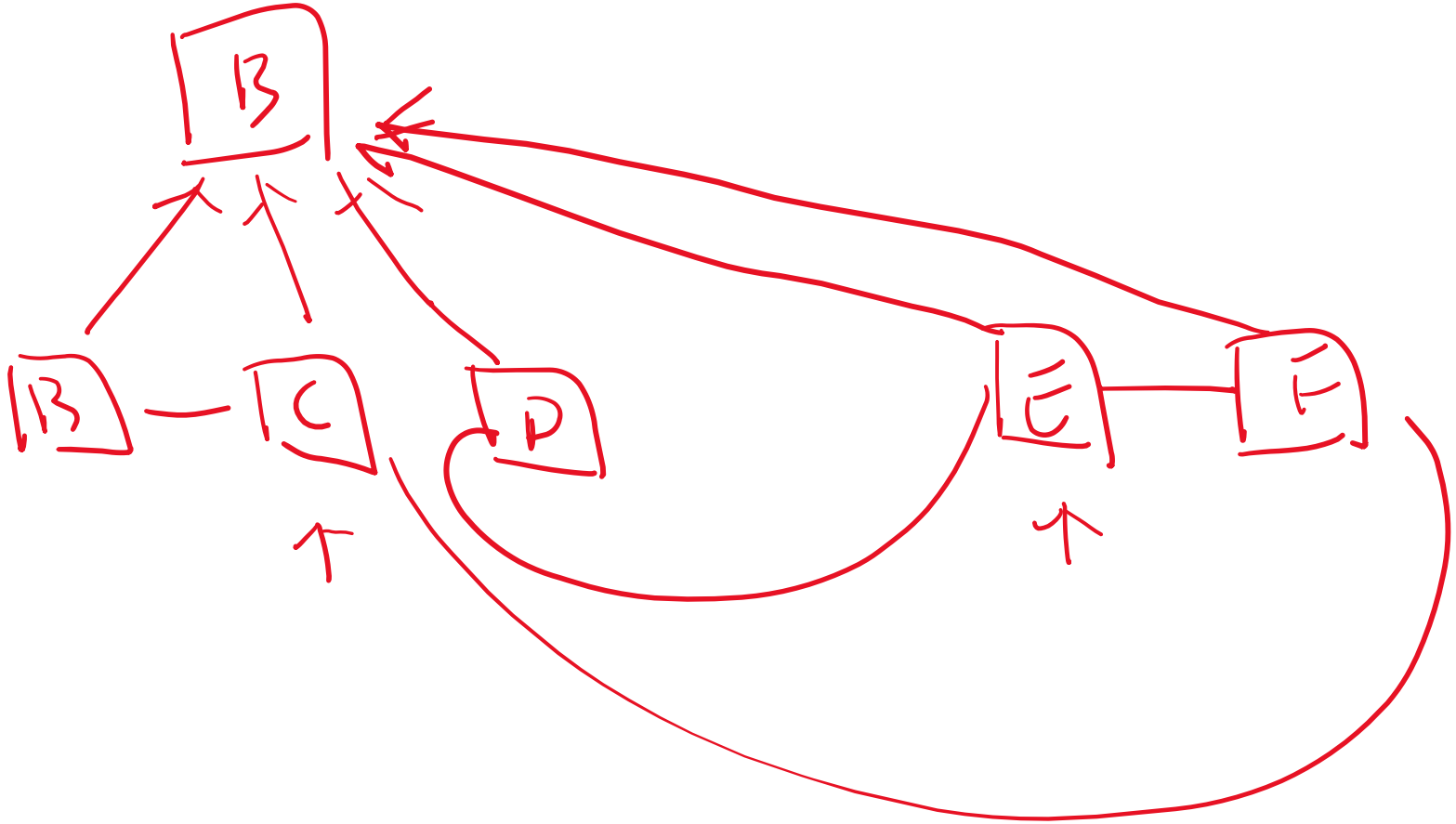
$\Rightarrow$ n union takes $O(n \log n)$ time

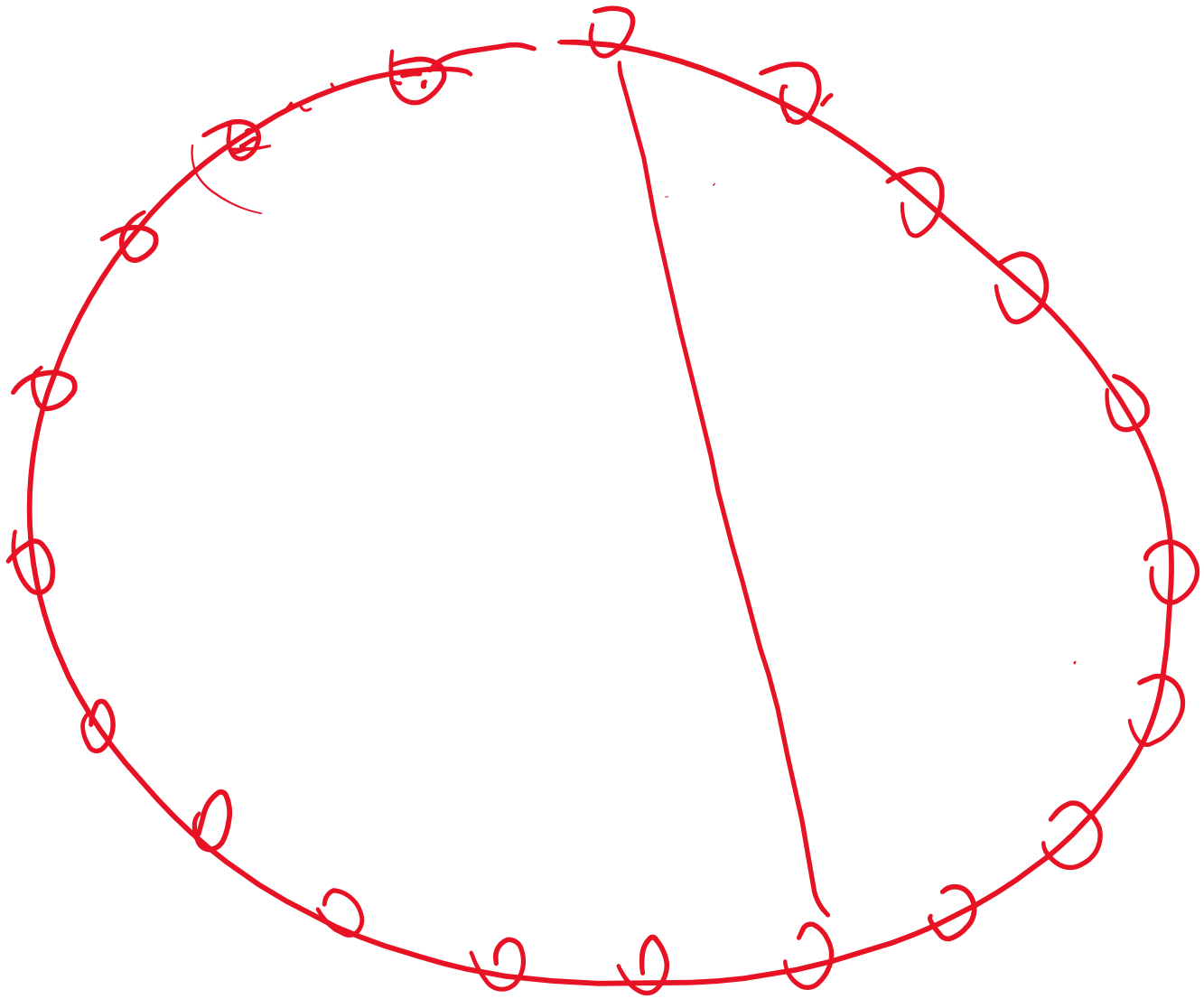
amortized time complexity of union $= \frac{O(n \log n)}{n} = O(\log n)$

Kruskal's algorithm:

$$O(m \log n) + O(1) \times O(n) + O(\log n)\, O(n) + O(1)\, O(m)$$

$$= O(m \log n)$$


B
B
C
D
E
F



λ_1

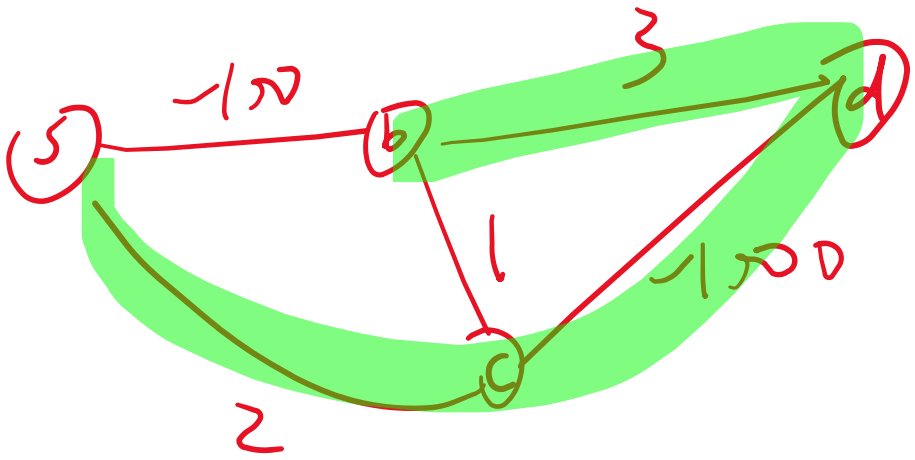
$\lambda_{1\infty}$

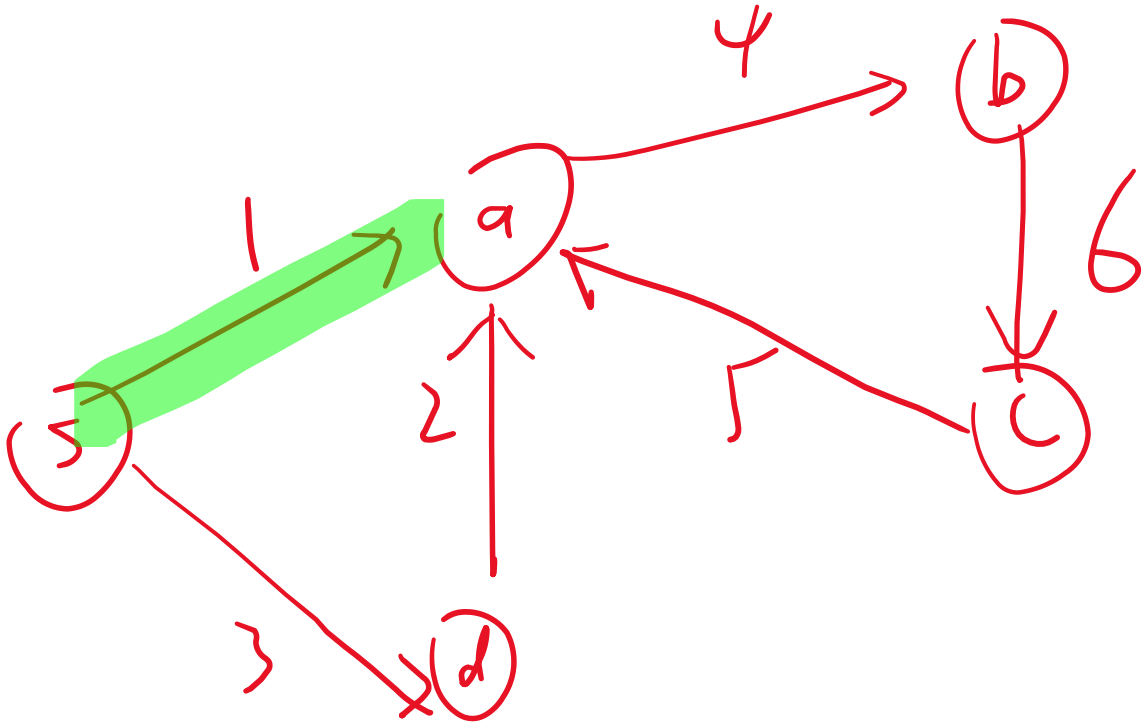
data $\{ d_1, d_2, \cdots, d_n \}$

$\Downarrow$ Store in Hash Table

key = d_i

value = d_i


a
b
c
d
10
3
1
100
2


4
b
a
1
6
s
2
5
c
3
d