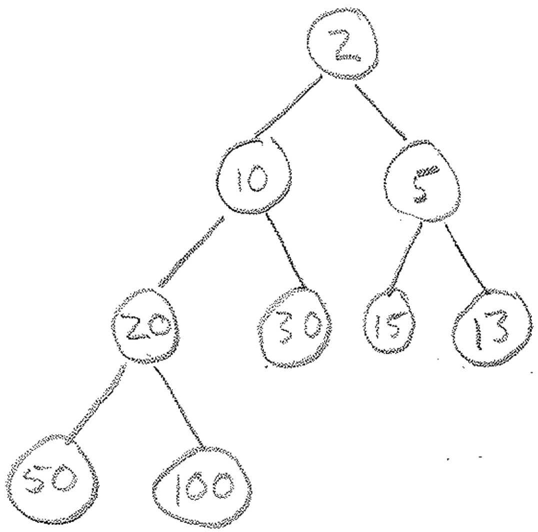
Tree Data Structures: Heaps (Review)

A heap is a Binary Tree with the following properties:

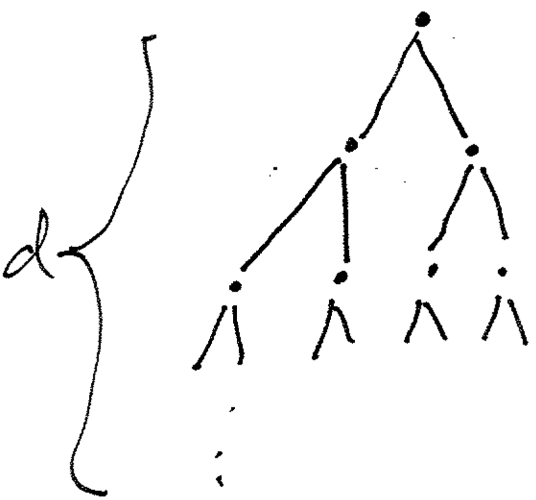
- Every node has a Key that is smaller than both its children (min-heap)
- It is full: all nodes are present at every level, except possibly the last (deepest)
- ~~At the~~ last level it is full-to-the-left



properties

① the minimum element is always at the root

② what is the depth of a heap with n nodes?



depth	number of nodes
0	1
1	2
2	4
3	8
⋮	
d	2^d

$$\sum_{i=0}^{d} 2^i = 2^{d+1} - 1$$

$n = 9$ depth: 3

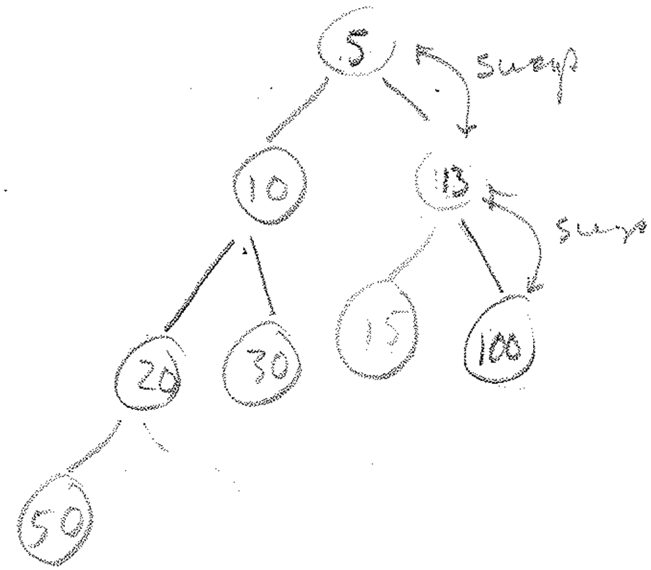
$$n = 2^{d+1} - 1$$

$$\log_2(n+1) - 1 = d \in \Theta(\log(n))$$

Basic Operations:

1) get-and-remove minimum

2) insert an element



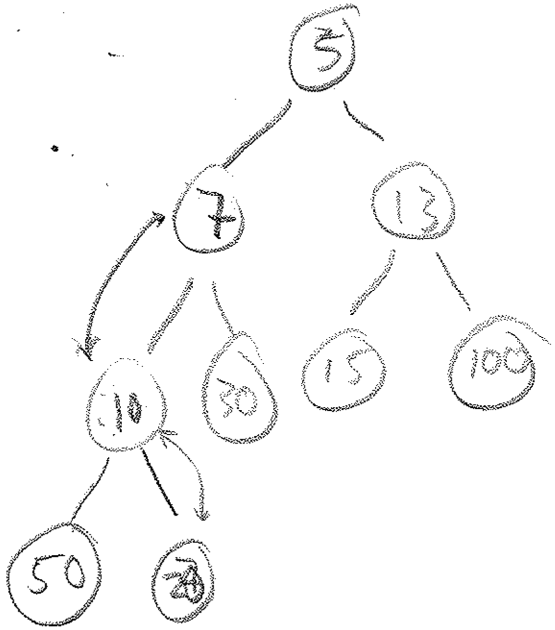
1) save off the root O(1)

2) replace the root with the "last" key O(1)

3) fix the heap:

exchange the swapped element with the min of its children until: O(d) = O(log(n))

- heap prop is satisfied or
- it becomes a leaf



insert 7:

1. insert at the "end" of the heap $O(1)$
2. fix the heap: "heapify"

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

implementation: Array

0	1	2	3	4	5	6	7	8	9	
–	5	7	13	10	30	15	100	50	20	...

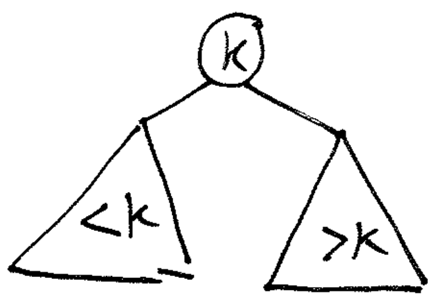
i: left Child: $2i$

right Child $2i+1$

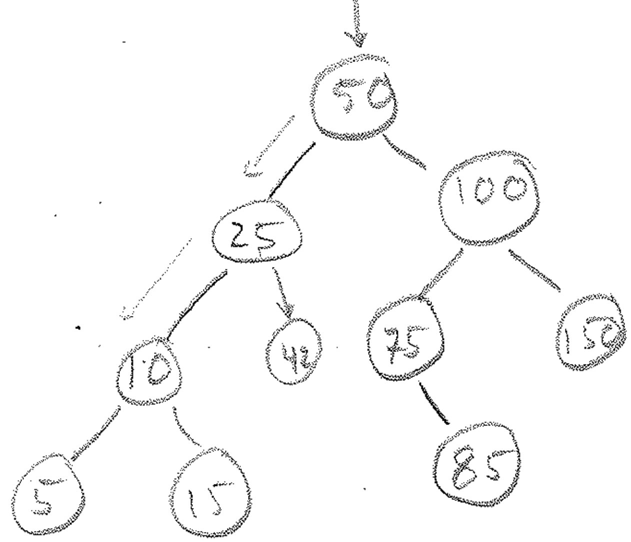
parent $\lfloor \frac{i}{2} \rfloor$

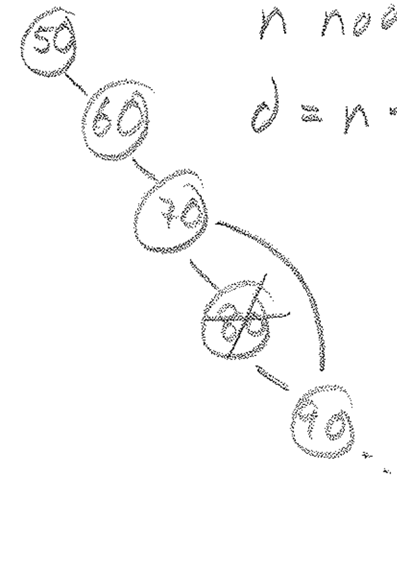
Binary Search Trees:

- Binary Trees
- Every node's key K is such that:
  - every node in K's left-sub-tree $< K$
  - every node in K's right sub tree $> K$



- insertion
- retrieval
- deletion.





n nodes

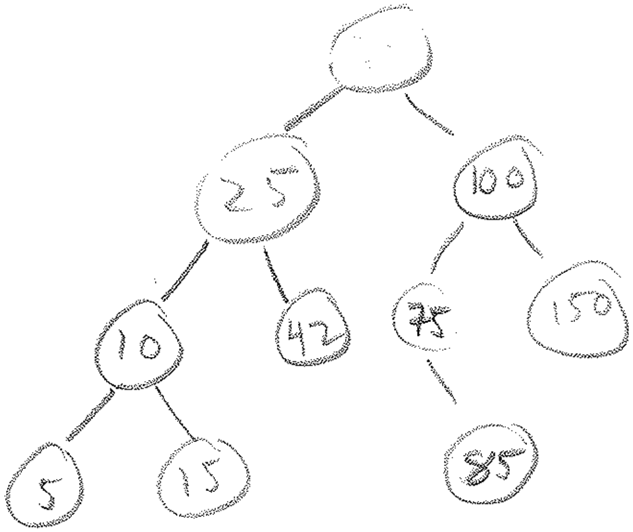
$d = n - 1$

retrieval: search for 10, 42

insert 42: search, insert as a new leaf.

- start at the root
- traverse left/right until:
  - you find a match or
  - you reach the end of the tree

$\Big\}\ O(d) \neq O(\log(n))$



delete: 85, 25, 50

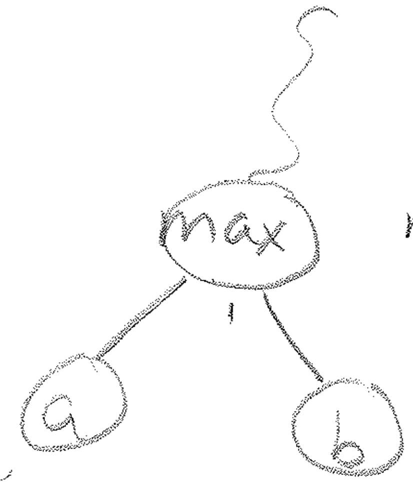
① find it (search) O(d)

if it is a leaf: delete it

if it has 1 child:
"promote" the single child

if it has 2 children

replace the node with the max. value in the left subtree or the min. value in the right subtree



not possible

Binary Search trees may not be balanced.

- $d = O(n)$
- Goal: add more structure to BSTs to guarantee $d = O(\log(n))$
- Balanced BSTs

  AVL trees

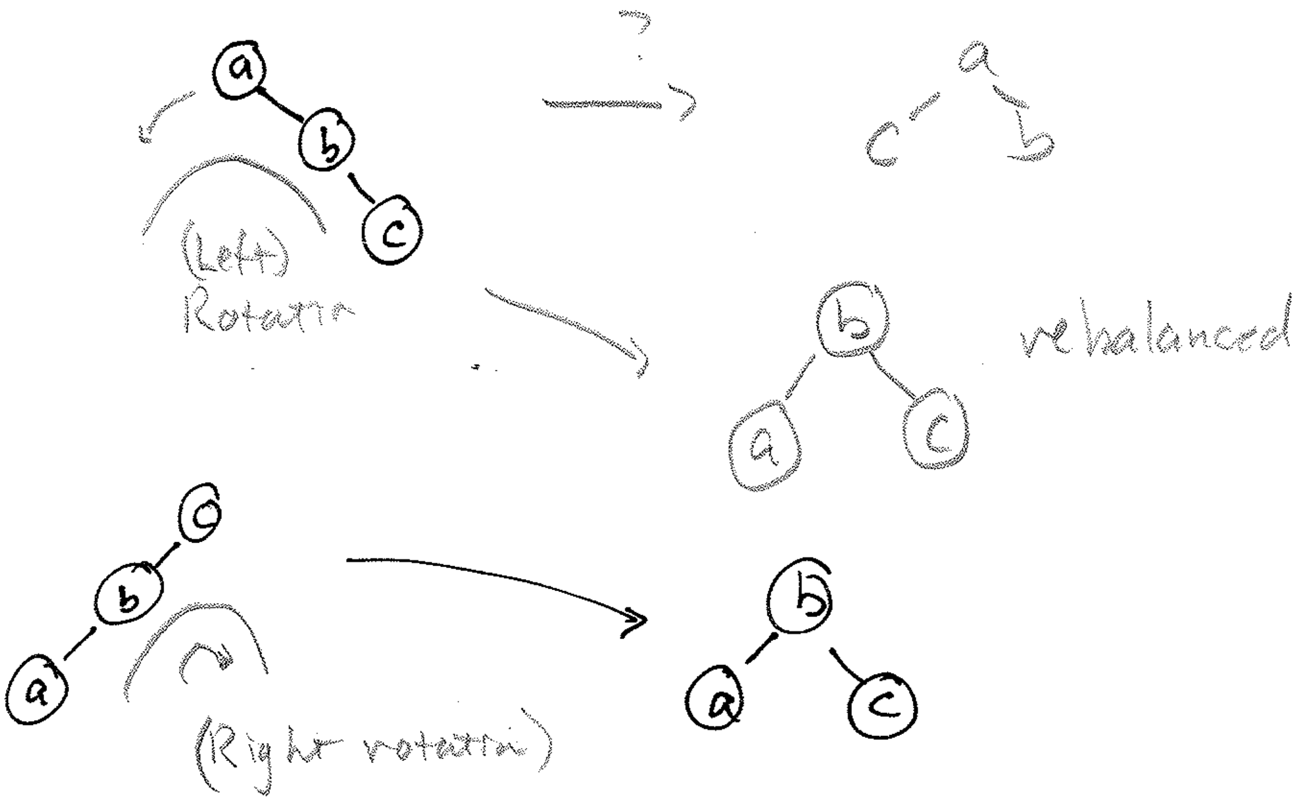
  B-trees (2, 3)

  Red-Black trees

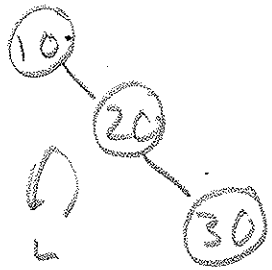
  treaps ?

  etc.

$a < b < c$

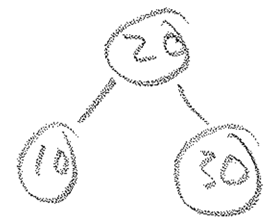


10, 20, 30

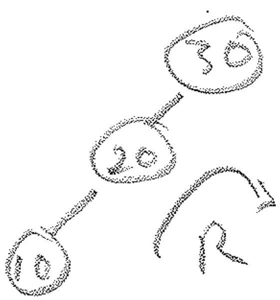


10 30 20

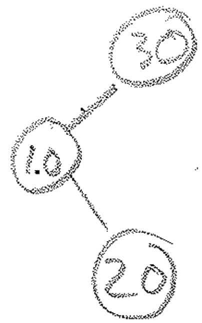
20 10 30



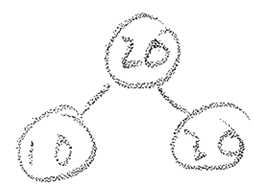
30 20 10

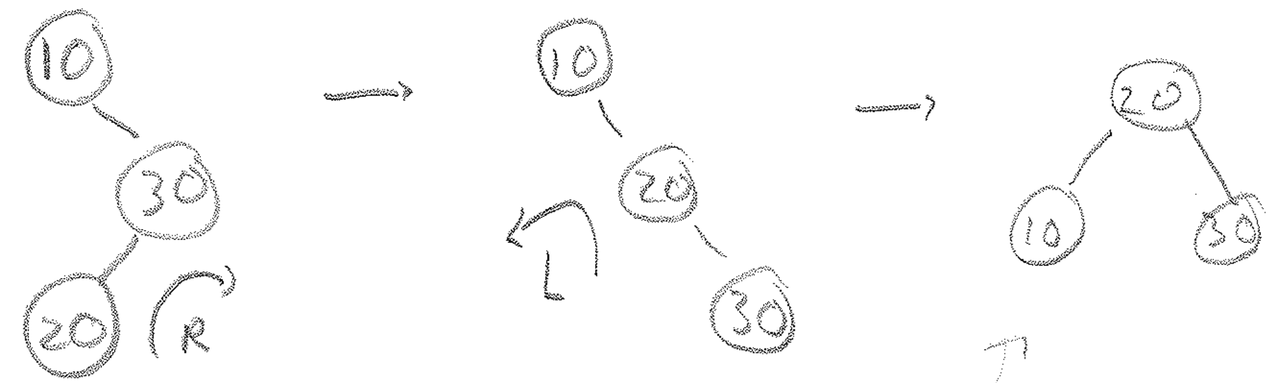


30 10 20

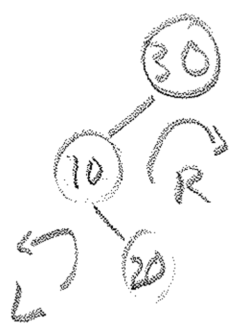


20 30 10





Right-left rotation (RL)

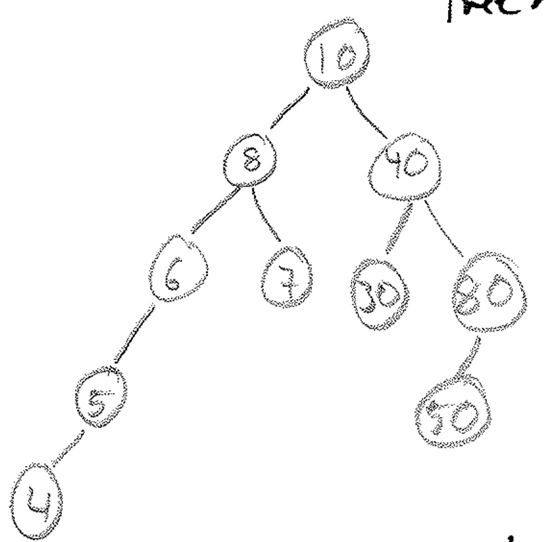


Left-right rotation (LR)

Defn

- The height of a node u in a Binary Tree is the length of the longest path from u to any descendant leaf

Tree height: 4



node	height
10	4
30	0
40	2
80	1

(leaves: height = zero)

- The height of a (sub) tree is the max. height of any of its nodes ie. the height of the root

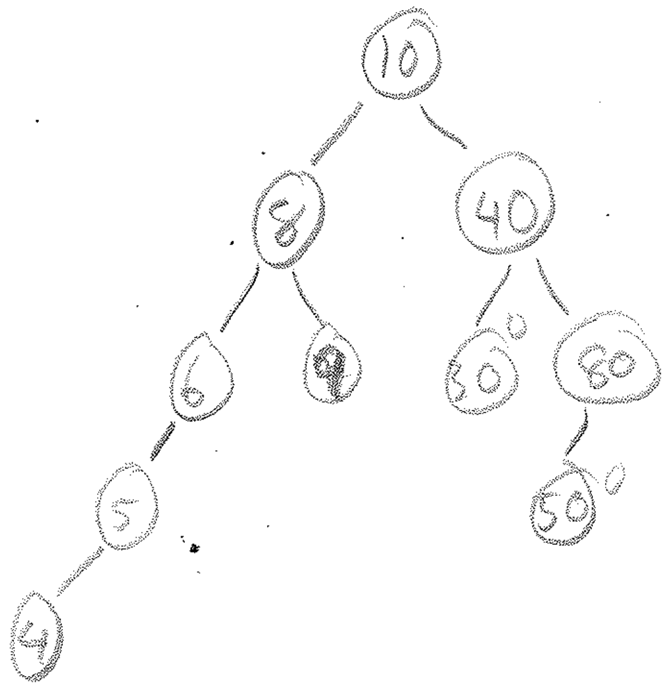
Defn

The balance factor (bf) of a ~~tree~~ node u is

$$bf(u) = height(T_\ell) - height(T_r)$$

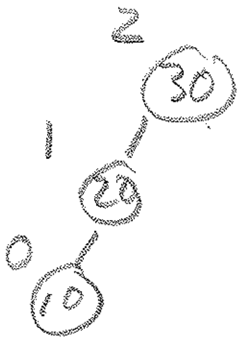
T_ℓ, T_r are the left/right subtrees of u

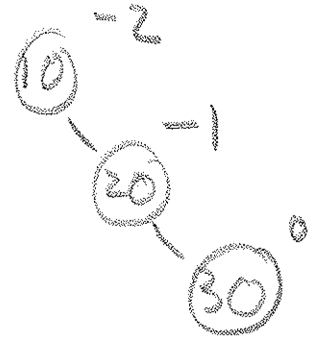
- single node tree has height 0
- empty tree has height -1

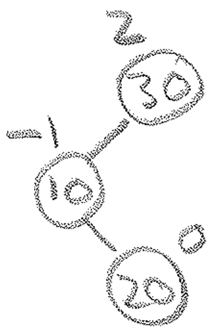


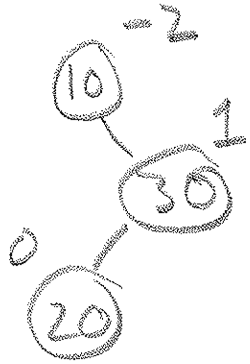
node	balance factor
10	3 − 2 = 1
8	2 − 0 = 2
9	−1 − (−1) = 0
6	1 − (−1) = 2
30	0
50	0
4	0
5	0 − (−1) = 1
80	0 − (−1) = 1
40	0 − 1 = −1

leaves : bf = 0







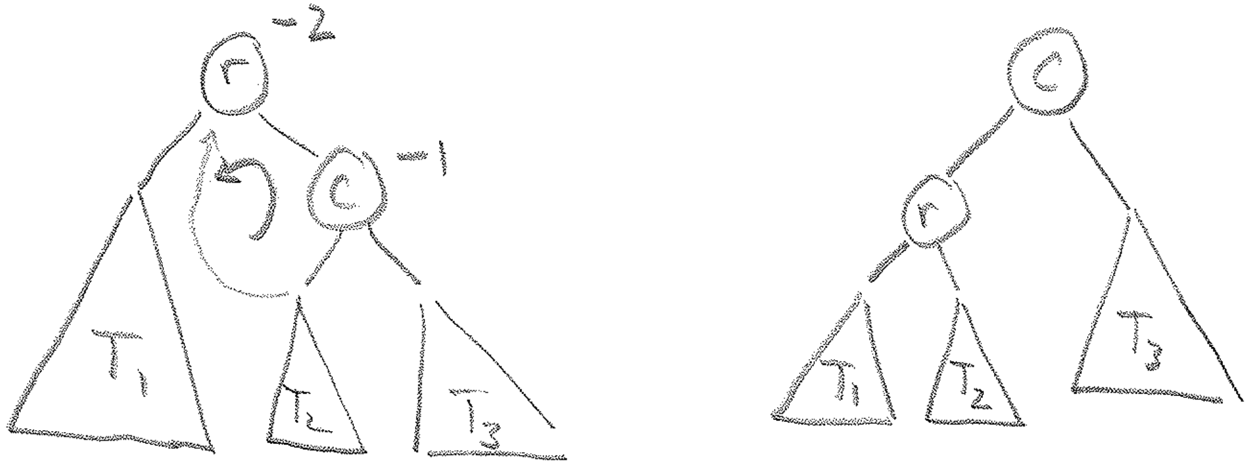


balance factor:

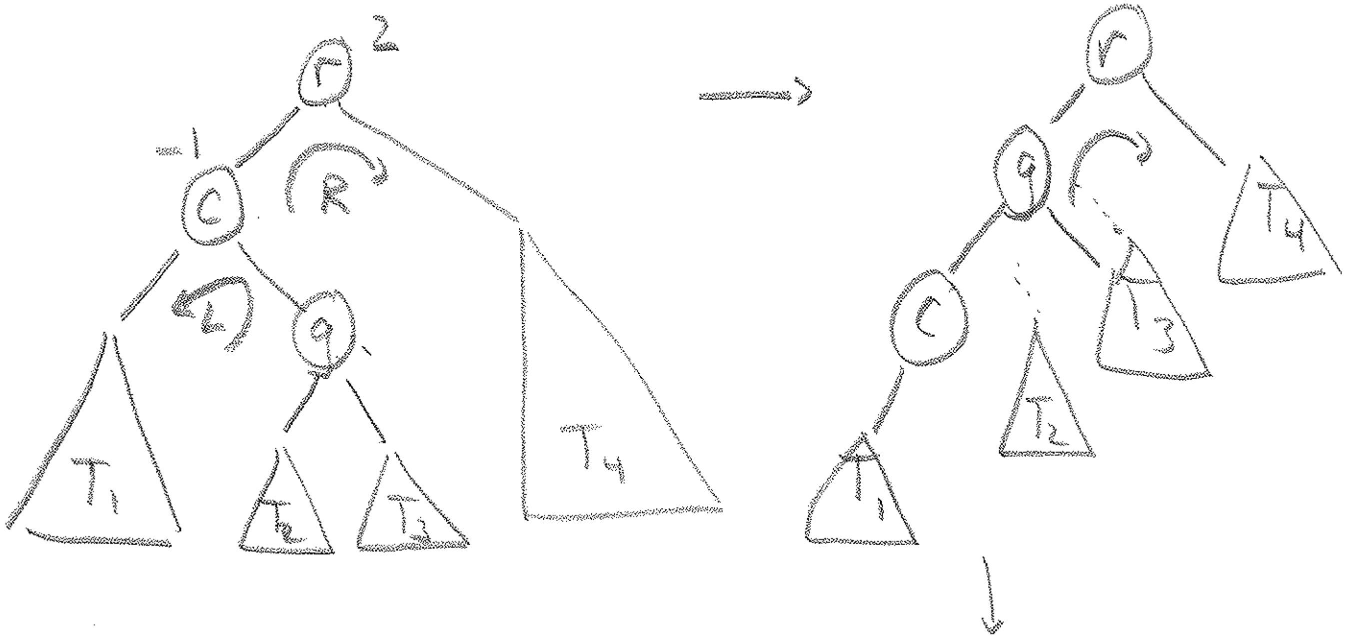
positive → skewed to the left → Right rotation

negative → skewed to the right ⇒ left rotation

General Left rotation



Generalized LR

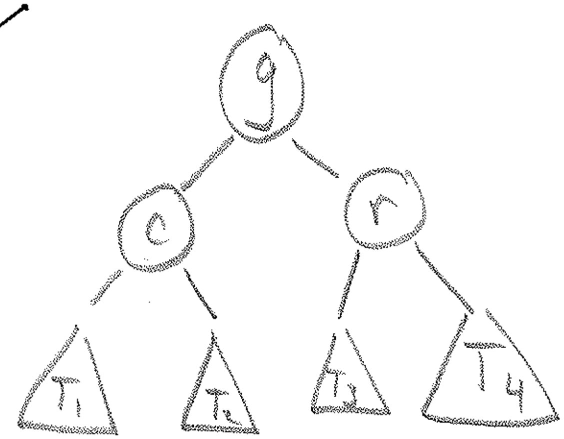


$r < T_4$ ✓ $c < g < r$ ✓

$g < T_3$ ✓

$T_2 < g$ ✓

$T_1 < c$ ✓



search: exactly the same

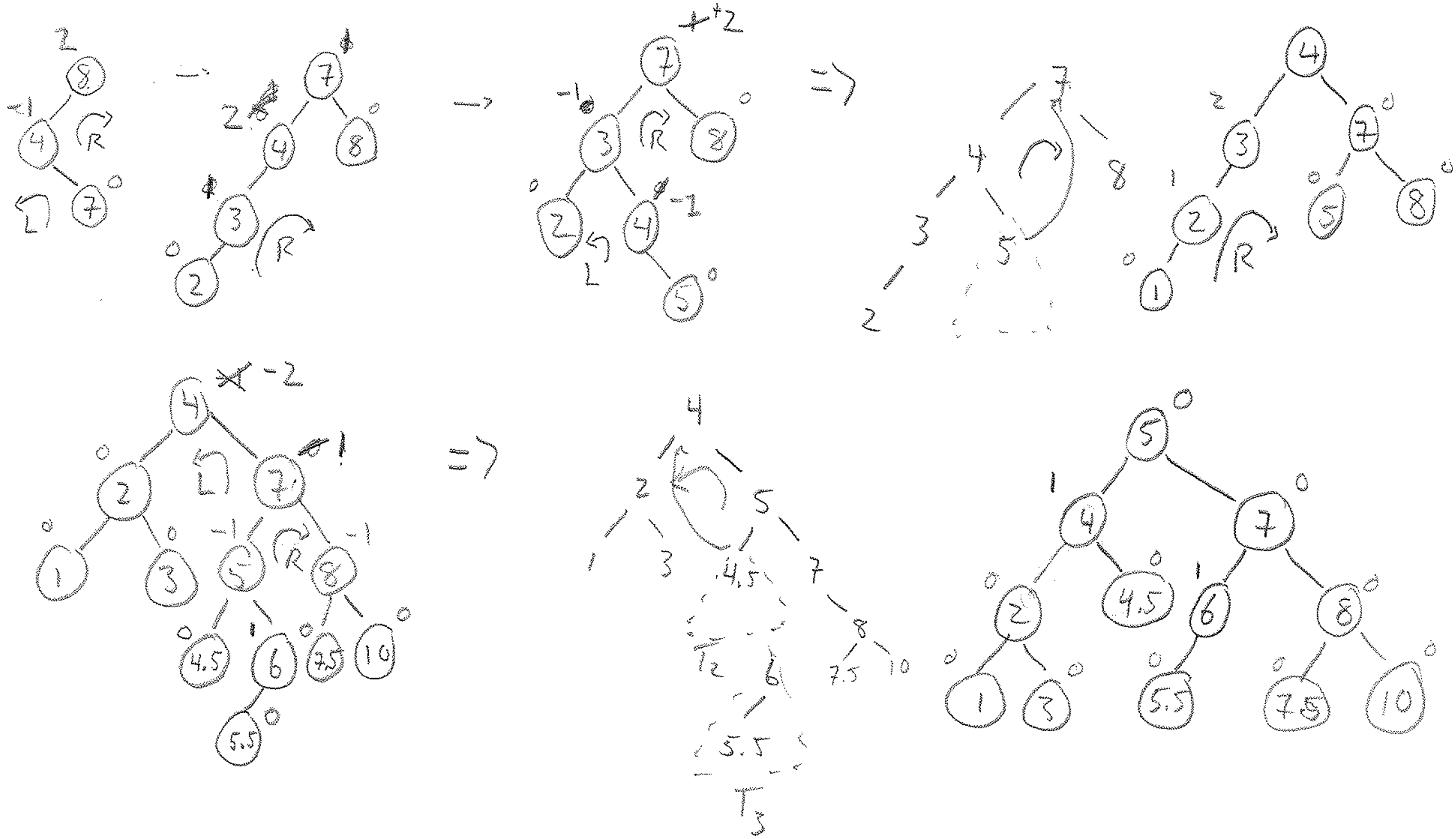
insertion: same: insert as a leaf

additionally: rebalance if necessary

deletion:

AVL Tree Example

Insert: 8, 4, 7, 3, 2, 5, 1, 10, 6



~~AE~~ AVL:

- height is bounded:

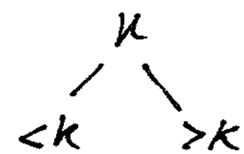
  $$\lfloor \log(n) \rfloor \leq h \leq 1.4405 \cdot \log_2(n+2) - .3277$$

- $h \in \Theta(\log(n))$

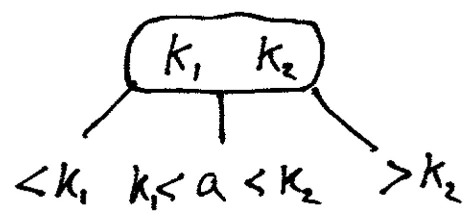
- All operations are $O(\log(n))$

2-3 Trees

- Every node has at most 3 children (2 or 3 children)
- Zero children: a leaf
- 1 child: DNE = Does not exist
- 2-nodes: 1 key 2 children



- 3 node: 2 keys K_1 K_2, $K_1 < K_2$
  3 children



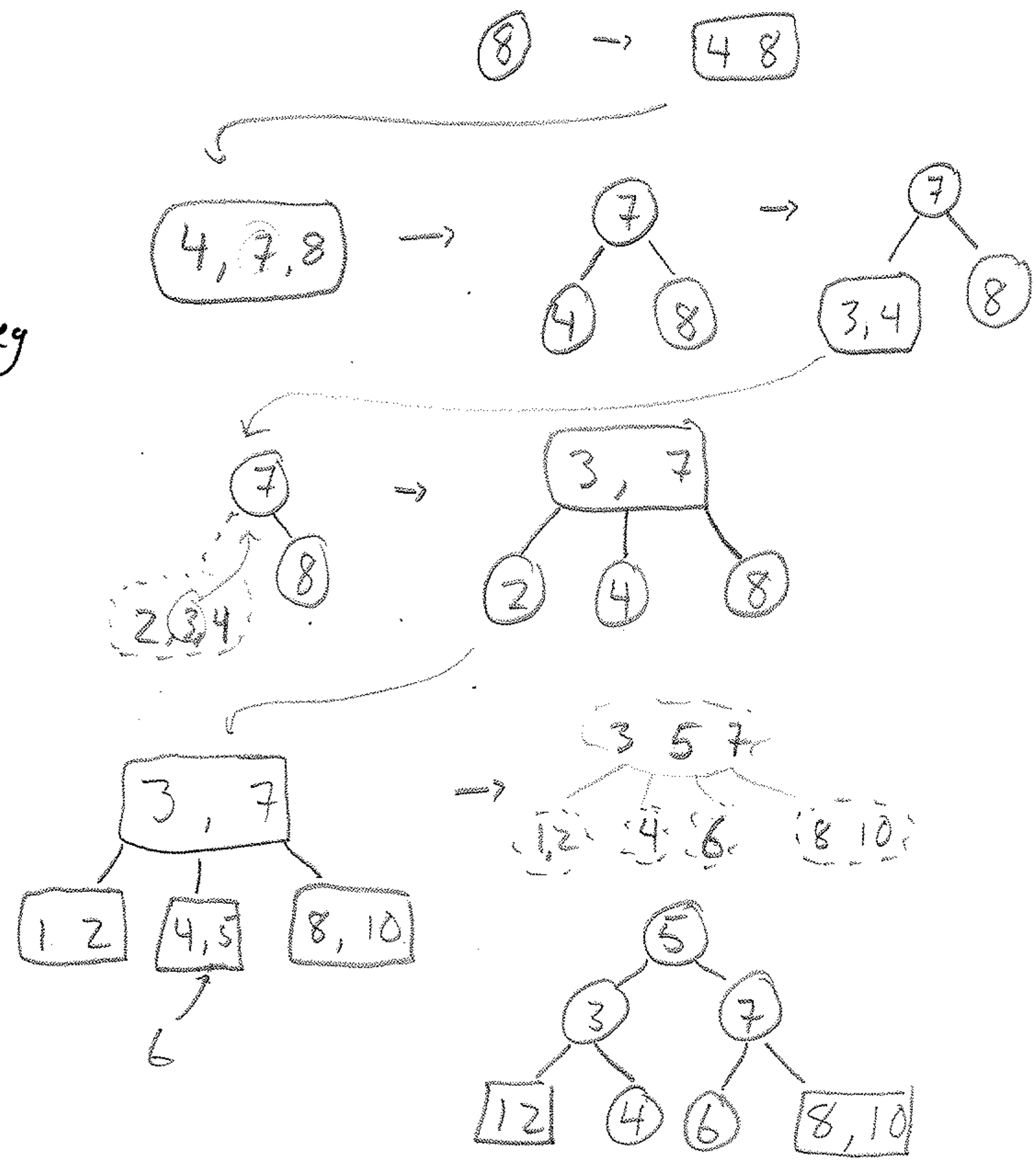
- ~~All nodes~~ The tree is "full":

  all leaves are at the same level.

8 4 7 3 2 5 1 10 6

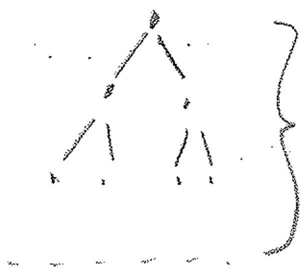
Insertion:

- always done in a leaf
- if inserted into a 2 node: it becomes a 3 node
- if inserted into a 3 node: middle element is promoted and the other 2 ~~nodes,~~ Key split into 2-nodes



min size$^{(n)}$ of a 2-3 of depth $\underline{d}$: $2^{d+1}-1$

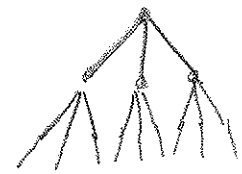
max sized d



d

1
2
4
8
⋮
2^d

$$\sum_{i=0}^{d} 2^i = 2^{d+1}-1$$



# nodes	# keys
1	2
3	6
9	18
⋮	
3^d	$2\cdot 3^d$

$$2\sum_{i=1}^{d} 3^i = 2\left[\frac{3^{d+1}-1}{2}\right]$$

$$= 2\left[3^{d+1}-1\right]$$

$$\sum_{i=0}^{n} x^i \overset{?}{=} \frac{x^{n+1}-1}{x-1}$$

# of keys depth

$$2^{d+1}-1 \le n \le 2[3^{d+1}-1]$$

$$2^{d+1}-1 \le n \rightarrow d \le \log_2(n+1)-1$$

$$n \le 2[3^{d+1}-1] \rightarrow \log_3\left(\frac{n}{2}+1\right)-1 \le d$$

$$\log_3\left(\frac{n}{2}+1\right)-1 \le d \le \log_2(n+1)-1$$

$$\therefore \quad d \in \Theta(\log(n))$$

Java: TreeSet<T> (Interface: SortedSet<T>)

↓

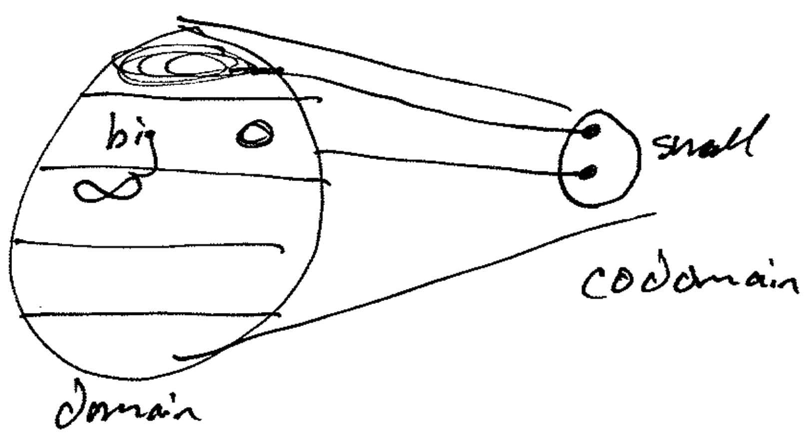
Red-Black Tree

Python: ?

hash-based data structures

Hash Tables

- amortized constant-time $O(1)$ operations
- Elements are stored in a regular old array
- Element's location ~~is~~ (index) is determined by a hash function
- A hash function maps a large domain to a small codomain



$h(x) = x \bmod 7$ % (remainder)

$h(17) = 3$

$h(23) = 2$

$h(28) = 0$

$h(29) = 1$

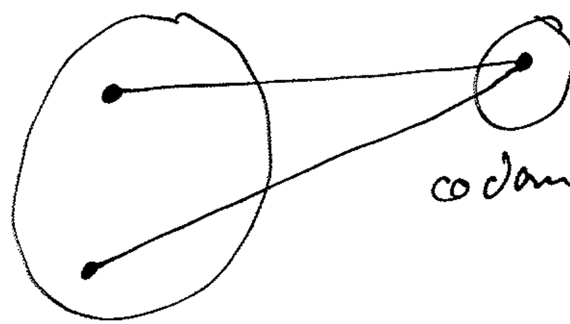
$h(27) = 6$

$\vdots$

$h(24) = 3$

$h(17) = h(24) = 3$

but $17 \neq 24$



codomain

domain

- hash functions are <u>not</u> 1-1
- different values may map to the same value
- collision

$h: \mathbb{Z} \to \{0, 1, \ldots 6\} = \mathbb{Z}_7$

zee mod 7

$\mathbb{Z}_m = \{0, 1, 2, \ldots m-1\}$

Hash Table: hold objects

- WLOG: map objects $\longrightarrow \mathbb{Z}$

(int) .hashCode()

- python: (?) __hash__(self) magic methods

  __str__(self) ..

  __construct__

$h: \mathbb{Z} \to \mathbb{Z}_m = \{0, 1, 2, \dots m-1\}$

Object $\to$ int $\to$ int $(0 \dots m-1)$

indices of a regular old array.

hashing: finding the location of an object is potentially $O(1)$

hashed object: used as a key to map to a value.

HashMap<K, V>

$h(k) = 7k + 13 \mod 8$

116

100, 108

0	1	2	3	4	5	6	7
13	4	~~100~~	10	~~108~~	16	116	86

Dirty bits

10, 16, 4, 13, 86

$h(10) = 7 \cdot 10 + 13 \mod 8$

$= 3$

$h(16) = 7 \cdot 16 + 13 \mod 8$

$h(4) = 1$

$h(13) = 7 \cdot 13 + 13 \mod 8$

$= 0$

$h(86) = 7 \cdot 86 + 13 \mod 8$

$h(100) = 713 \mod 8$

$= 1$ collision

$h(108) = 7 \cdot 108 + 13 \mod 8$

$= 1$

$h(\cancel{10}116)$

Collision Resolution:

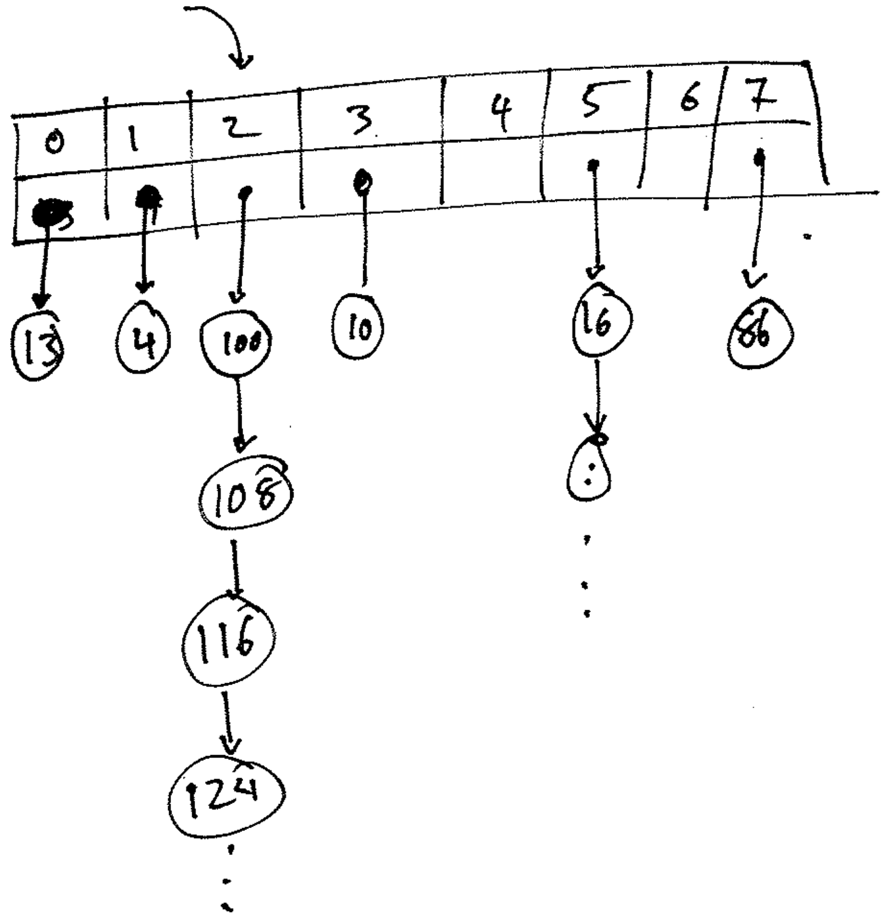
- Linear probing: if a cell is occupied, go to the next cell until you find an unoccupied cell

- linear function probe: $h'(k) = h(k) + \ell \bmod m$

- quadratic probing:

$$h(k,i) = h(k) + c_1 i + c_2 i^2 \bmod m$$

- Chaining: cells can be <u>other</u> data structures instead of 1 object



Linked List O(n)

BST

~~Hash Table~~ (?)

- A load factor:

$$\alpha = \frac{n}{m} = \frac{\text{number of elements in the hashtable}}{\text{size of the hash table}}$$

$0 \le \alpha \le 1$

α : % of "fullness"

1 = 100% full

0 = empty

.5 = 50% full

.75 = 75% full

- when the hash table hits a predetermined load factor

↓

rehashed. increase m

- created a new larger array, rehash every element.

- $O(n)$ operation

<u>but</u> infrequent operation

Average or

Amortized cost

$\sim O(1)$

8. high balance factor = poor performance, low memory

low balance factor = less chance of collisions

= better performance

= more memory

Time / space trade off data structure.