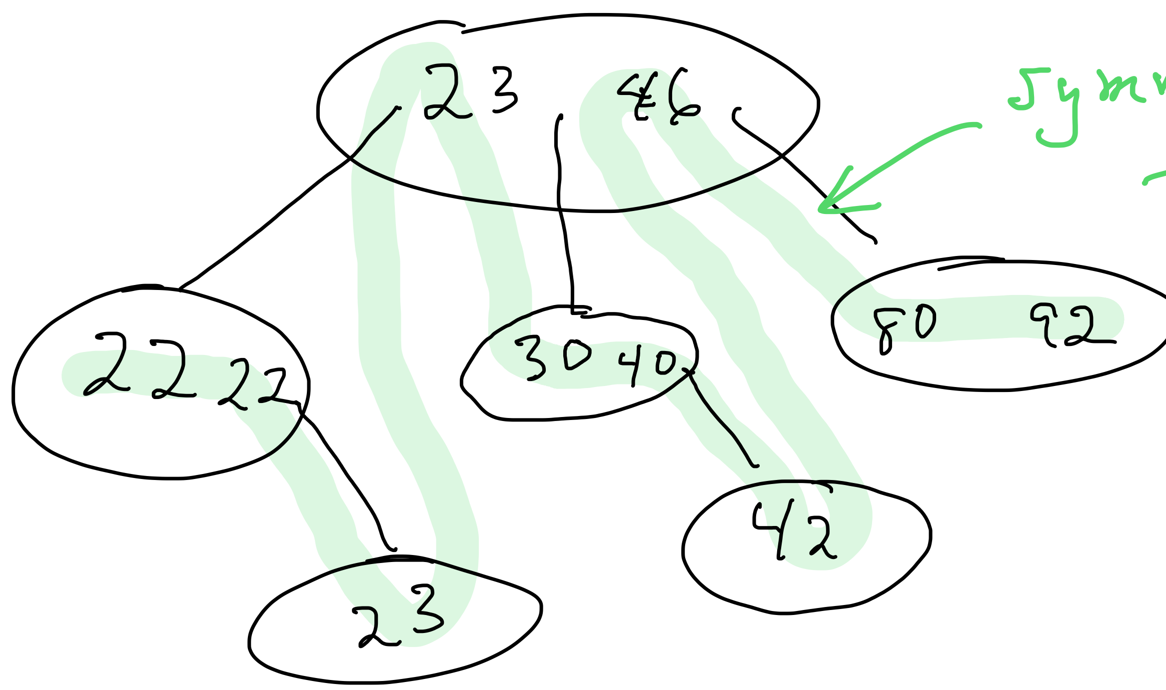
Data structures: package 2

Ternary trees.

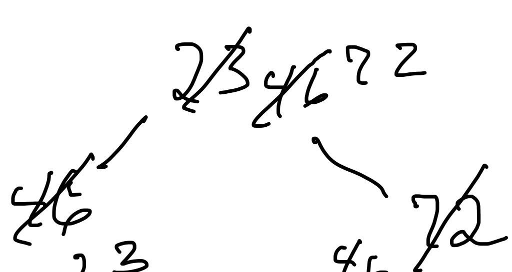


symmetric order traversal

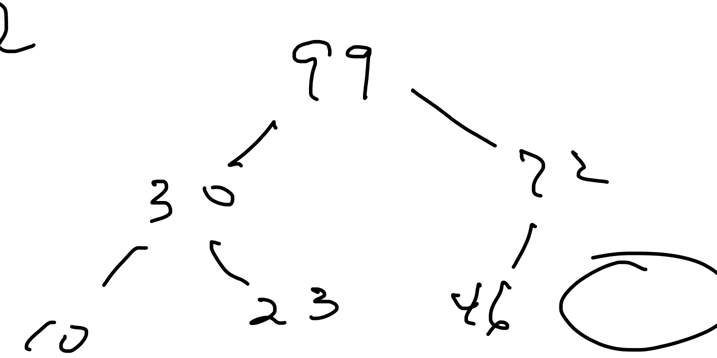
((22 22(23)) 23 (30 40(42)) 46 (80 92))

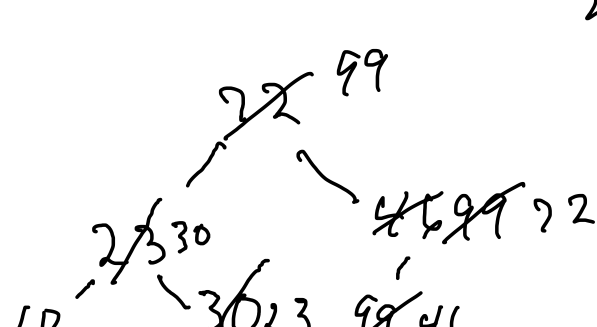
Heaps (data structure, as opposed to "the heap" = free storage allocation)

top-heavy: value at a parent ≥ values at children.
(top-light: ≤)



sift up



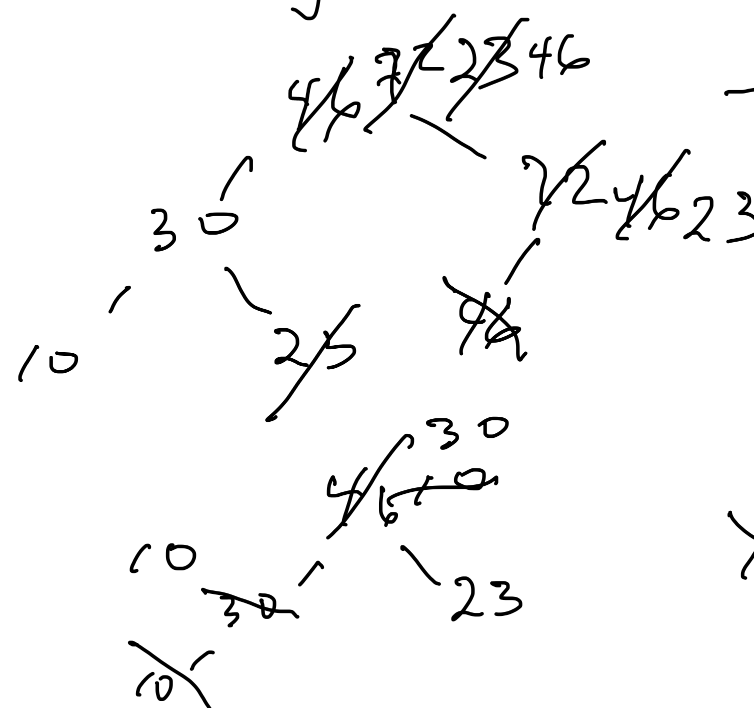


insertion: placing $\Theta(1)$ sifting up: $\Theta(\log n)$

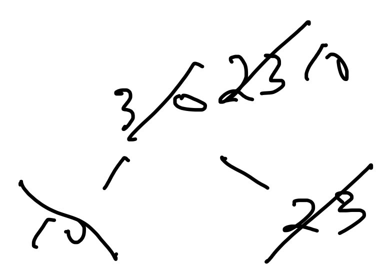
$\Longrightarrow \Theta(\log n)$

deletion: always returns top value.

sift down



→ 99, 72, 46, 30; 23, 10



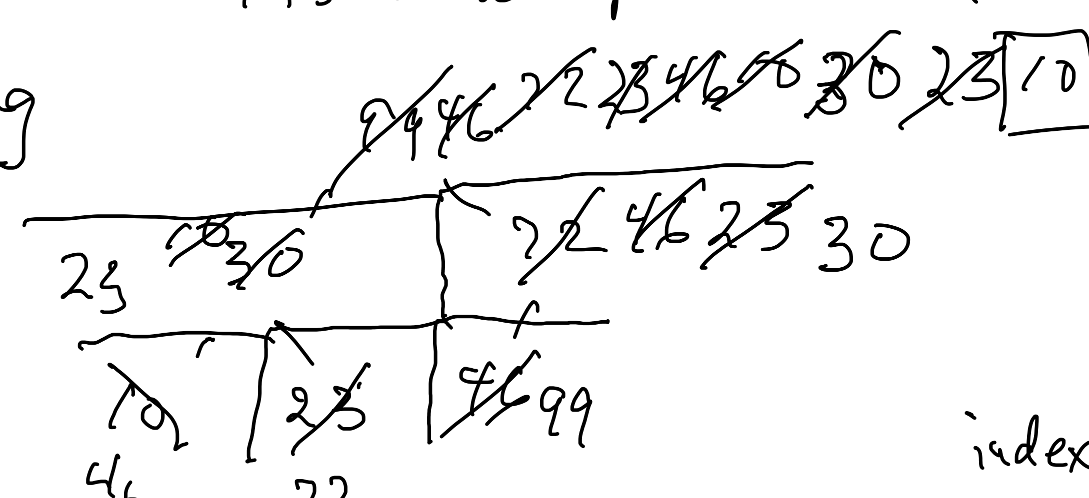
removal: $\Theta(1)$, replacing with last element: $\Theta(1)$, sifting down: $\Theta(\log n)$

$\Longrightarrow \Theta(\log n)$

purposes:

priority queue: each element represents work to do
its value represents its priority

Sorting



| index | children |
|---|---|
| 1 | 2, 3 |
| 2 | 4, 5 |
| 3 | 6 |
| $n$ | $2n, 2n+1$ |

How to store a heap on the computer?

array

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | | |
|---|---|---|---|---|---|---|---|---|
| $\infty$ | 99 | 30 | 72 | 10 | 23 | 46 | | |

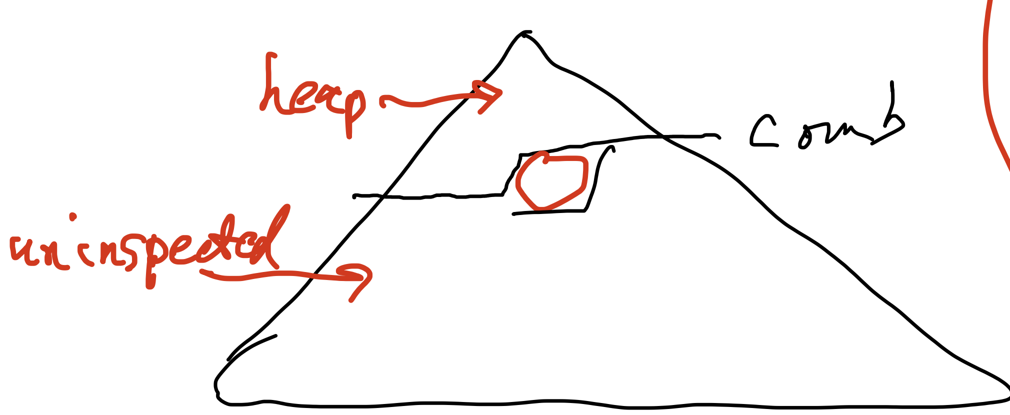
| index | parent |
|---|---|
| 1 | |
| 2 | 1 |
| 3 | 1 |
| 4 | 2 |
| 5 | 2 |
| 6 | 3 |
| $n$ | $\lfloor n/2 \rfloor$ |

to sort $n$ numbers.

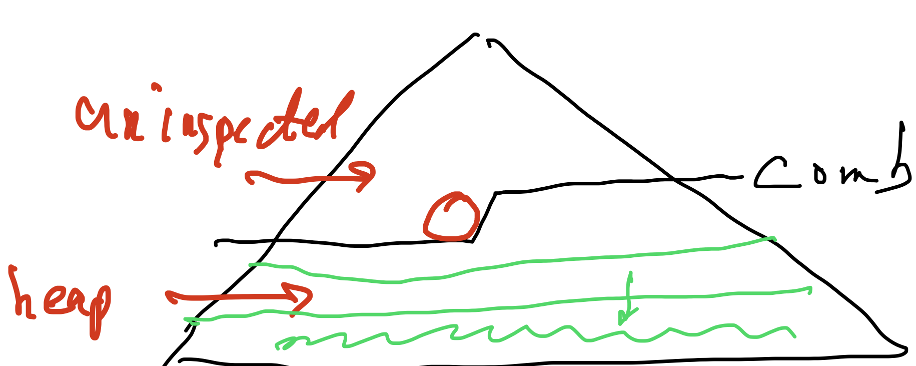
I: insert them 1 by 1 into an initially empty heap.
delete them 1 by 1 into available space at end.

$$\Theta(n \log n) + \Theta(n \log n) = \Theta(n \log n)$$

II: insert them into an unsorted array $\Theta(n)$
heapify the array: $\Theta(n \log n)$ or $\Theta(n)$
delete as before: $\Theta(n \log n)$
$= \Theta(n \log n)$




for each new element, sift up.
$\Theta(n \log n)$




for each new element, sift down.
trick: start with comb at half-way point

$\frac{1}{2}n$ : no sifting needed

$\frac{1}{4}n$ : sift 1 step

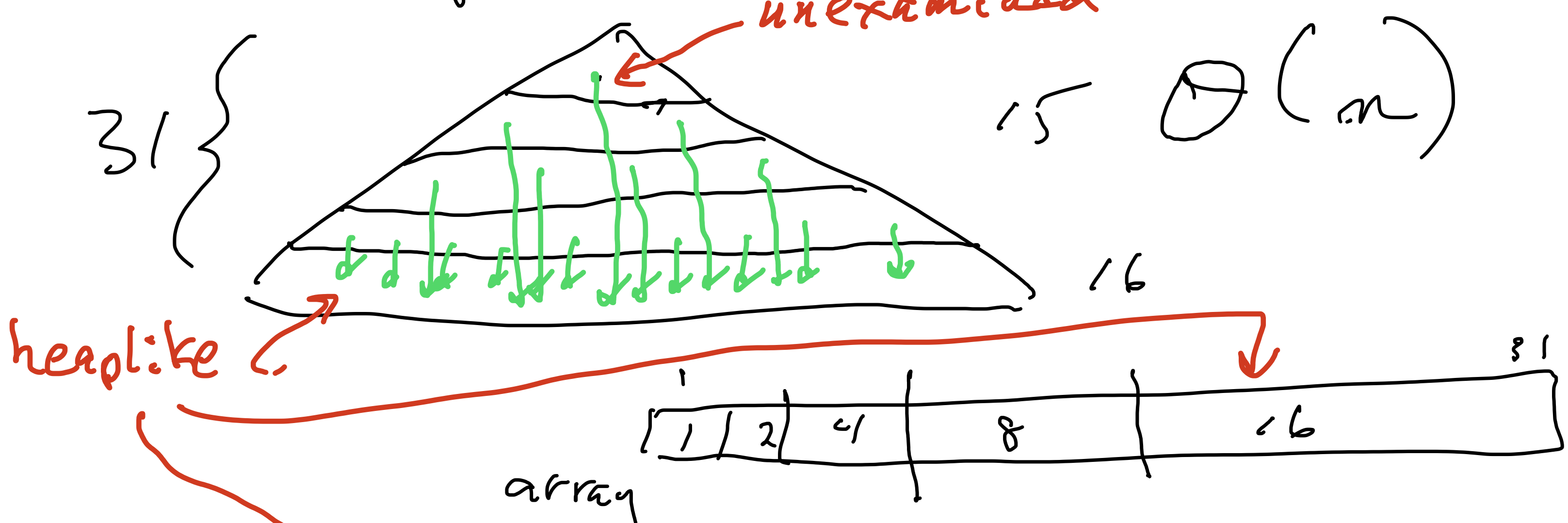
$\frac{1}{8}n$ : sift 2 steps

$\frac{1}{16}n$ : sift 3 steps

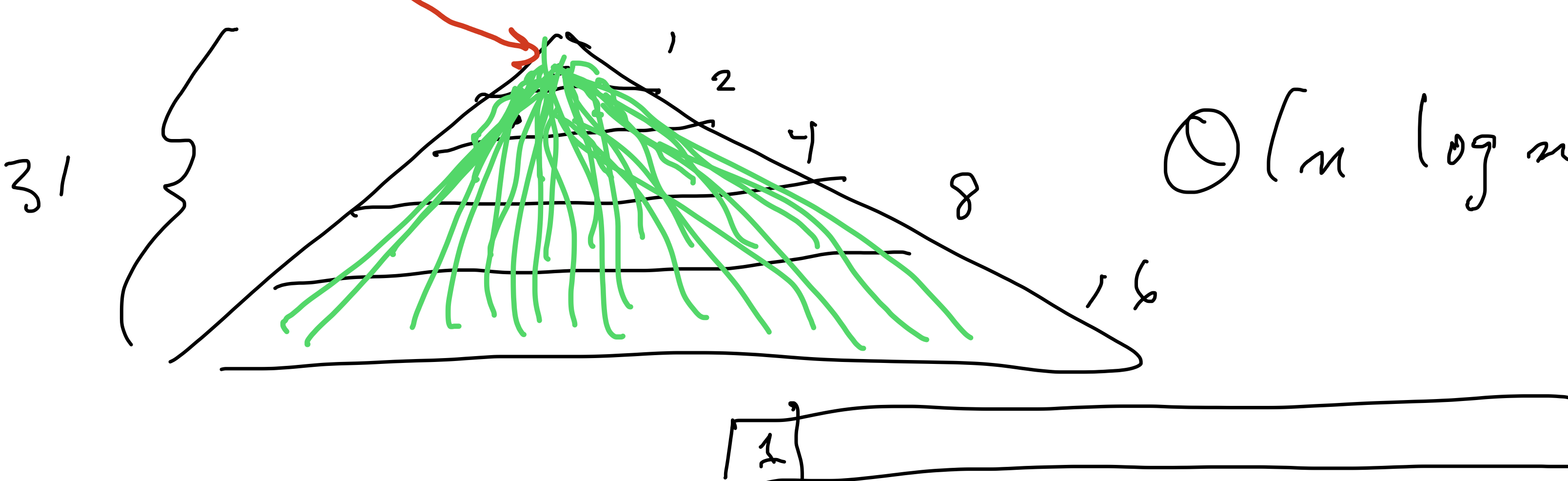
⋮

$$\frac{1}{2}n(0)+\frac{1}{4}n(1)+\frac{1}{8}n(2)+\cdots = n\left(\frac{1}{4}+\frac{2}{8}+\frac{3}{16}+\frac{4}{32}+\cdots\right) \le n$$

$$\lim_{j\to\infty} \frac{n}{2}\sum_{1\le j} j/2^j \to n$$




$\Theta(n)$



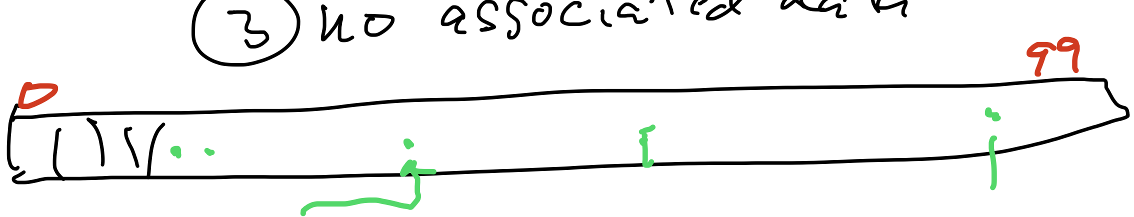

$\Theta(n \log n)$

Bin sort

Assume: ① know small range of keys eg. 0–99

② no duplicates

③ no associated data



97, 43, 64, 10, 12, 14

method: set the bit in array for each key
then read of which bits are set.

Time complexity: $n$ = # of elements to sort
$r$ = range of possible values

$$\Theta(n) + \Theta(r) = \Theta(n+r) = \Theta(r)$$

because $n \le r$

Relax assumptions?

① ✗

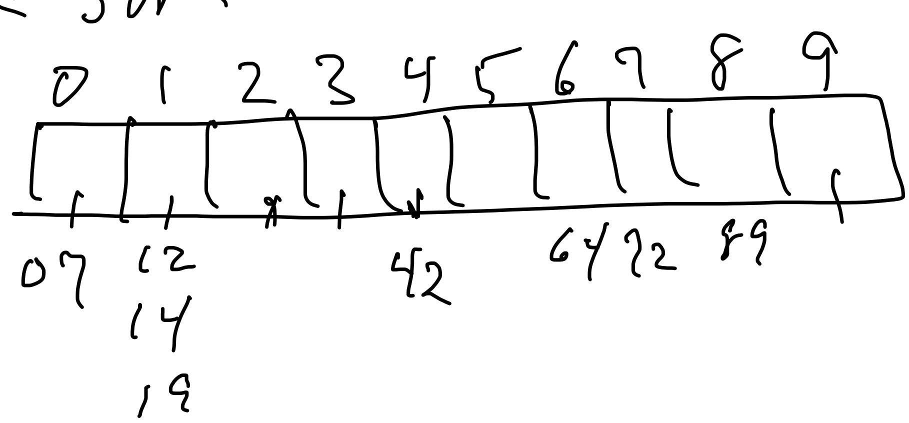
② instead of a bit, every cell of array is an integer count. Space is now $\Theta(r \cdot \log n)$

cells width of a cell.

③ storing data in array instead of a bit
Space is now $\Theta(r \cdot \text{sizeof(data)}) = \Theta(r)$
more precisely: $r \cdot$ sizeofdata bytes.

---

Radix sort




| | | |
|---|---|---|
| 89 | 42 | 07 |
| 42 | 72 | 12 |
| 64 | 12 | 14 |
| 72 | 64 | 19 |
| 12 | 14 | 42 |
| 14 | 07 | 64 |
| 19 | 89 | 72 |
| 07 | 19 | 89 |

stable sort: duplicates in order


| | | |
|---|---|---|
| 11 | 11 | 11 |
| 12 | 12 | 12 |

each pass uses a different place, adds at list ends.

Complexity: each pass: $n$ steps

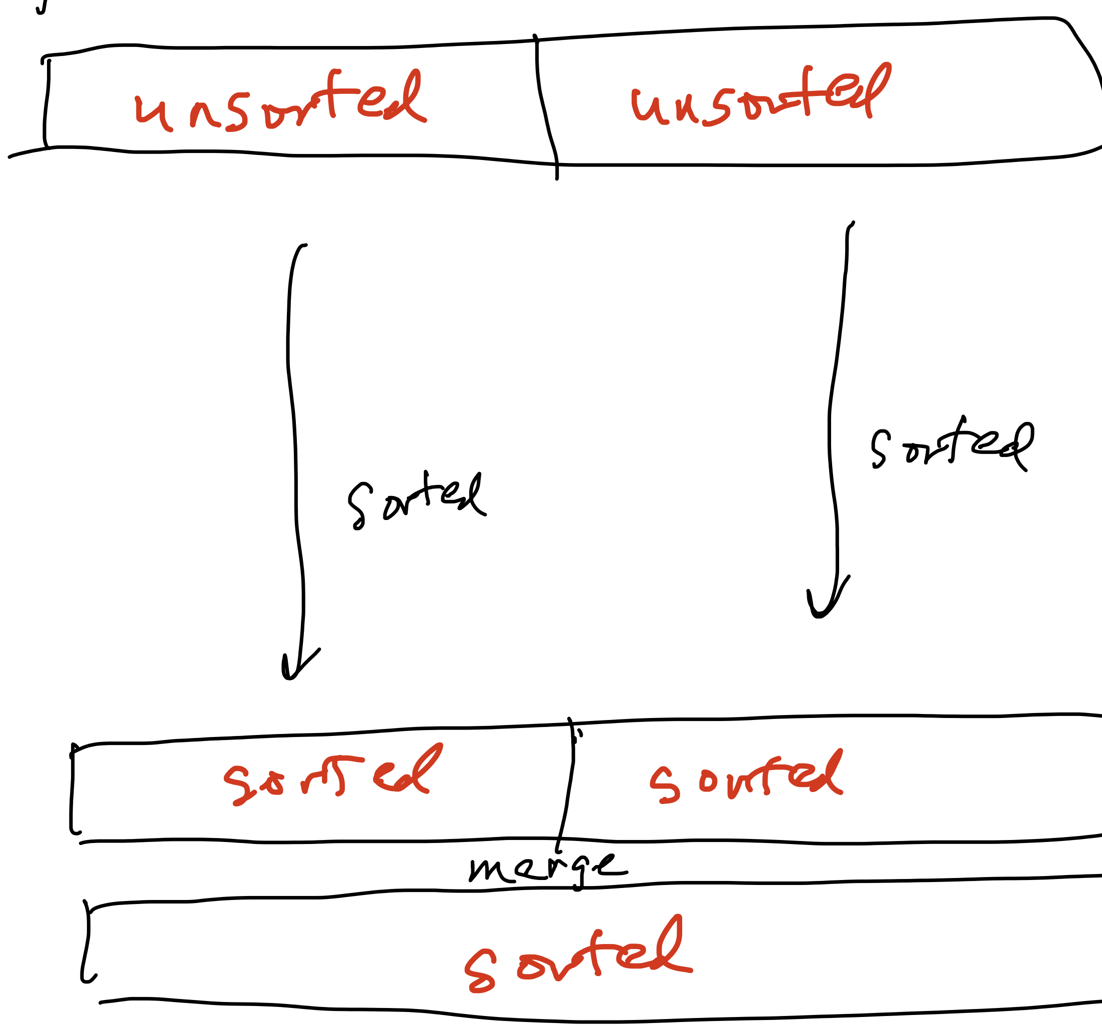
# of passes = # of digits = $d$

$$O(n\, d)$$

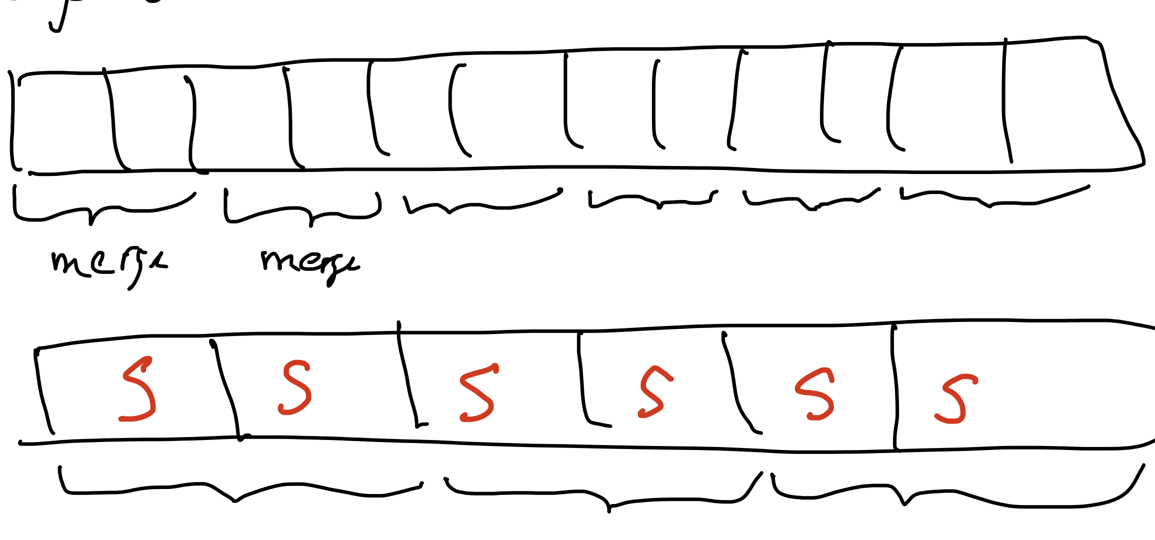
$\nwarrow$ $\log(n)$?

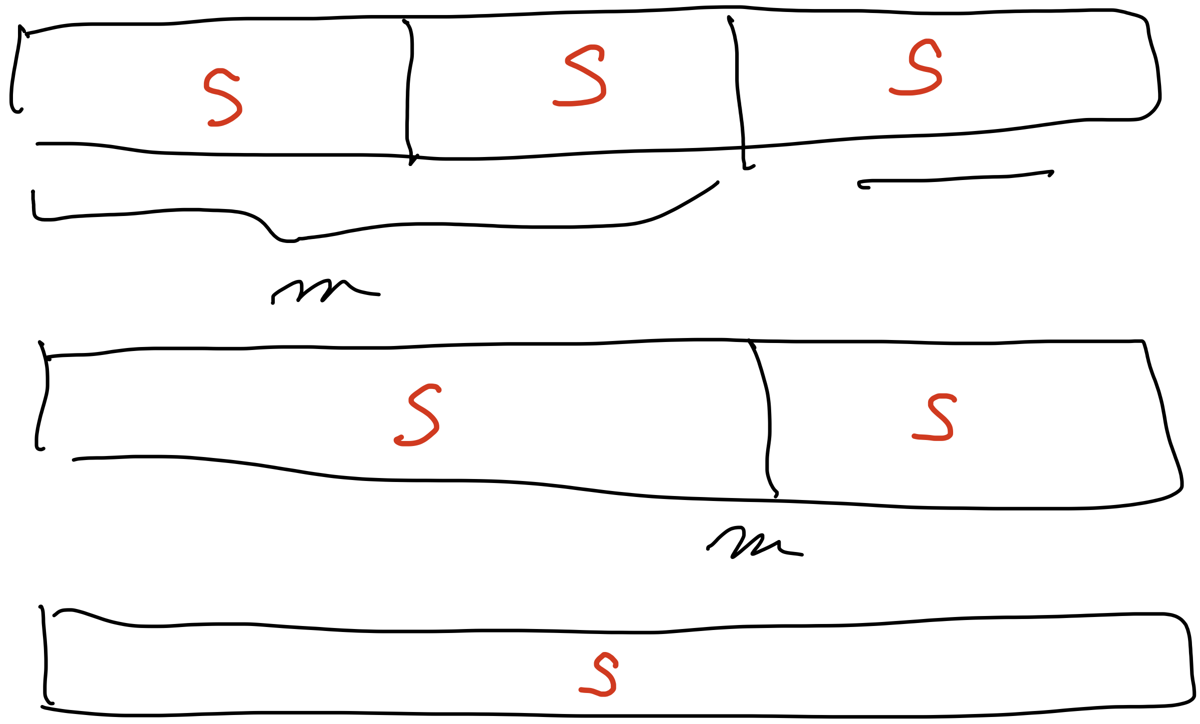
---

Merge Sort

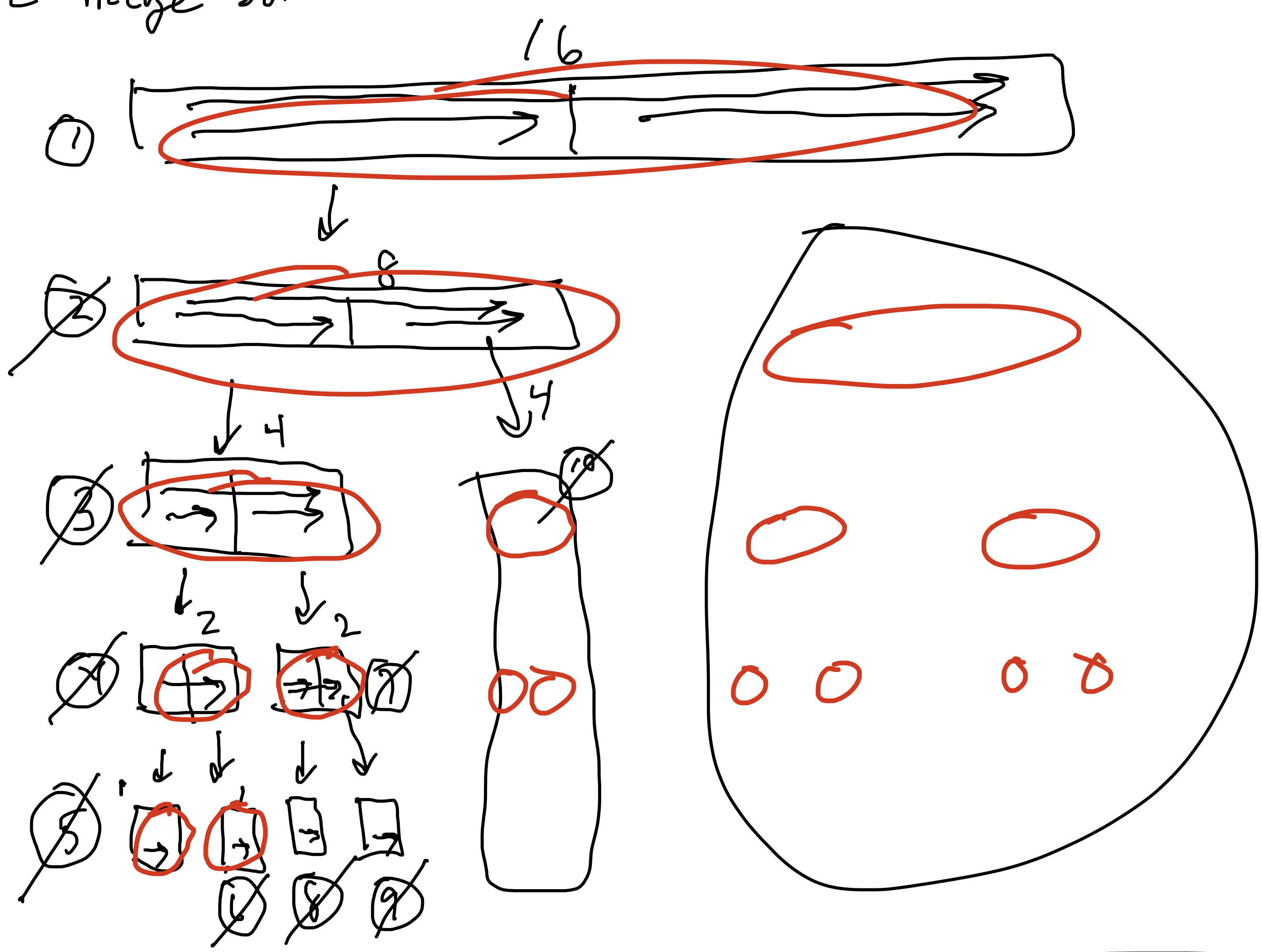



Iterative merge sort






Recursive merge sort



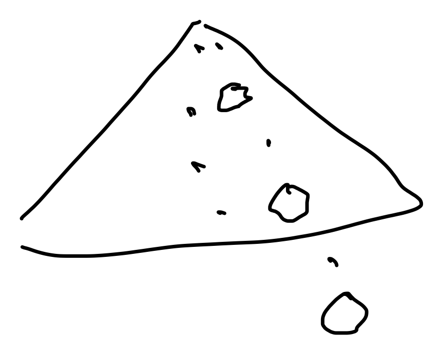

Red-black trees (Guibas + Sedgewick)(1978)

goal: binary trees that are mostly balanced.

"self-balancing" trees

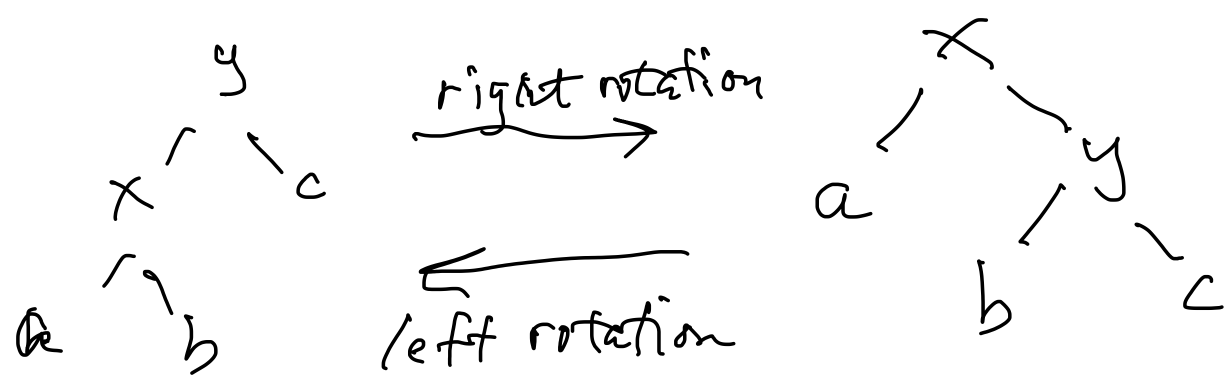
insertion may cause a balancing $\Theta(\log n)$

Rules:
1) each node is red or black.
2) each node is linked to left, right, parent.
3) null nodes are black.
4) root is black
5) red nodes have only black children.
6) all paths from root to a leaf have the same number of black nodes.



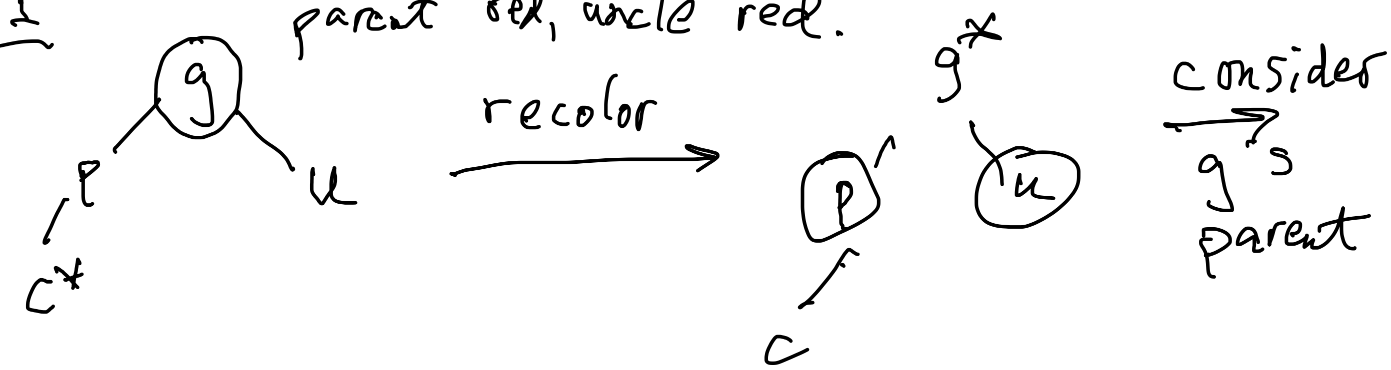
To insert:

Sometimes we rotate at a node



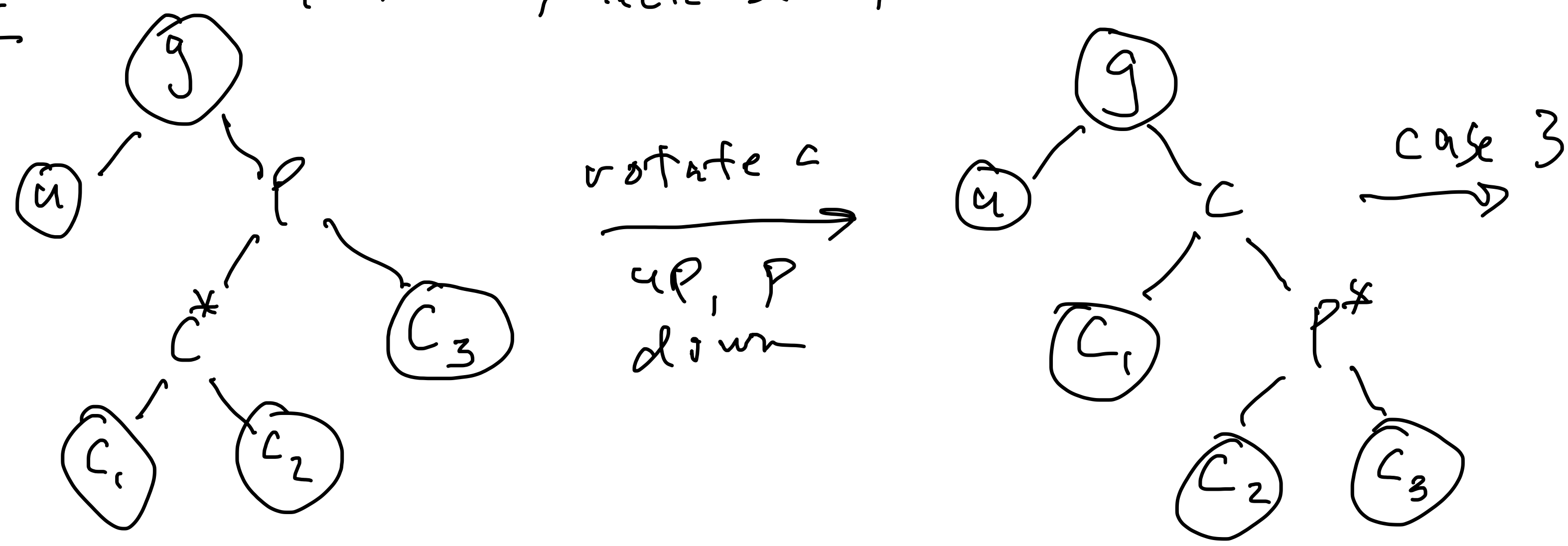

put new node in the tree (standard insertion)
color it red.
→ rotate as necessary
make root black (circled)

Case 1 parent red, uncle red.



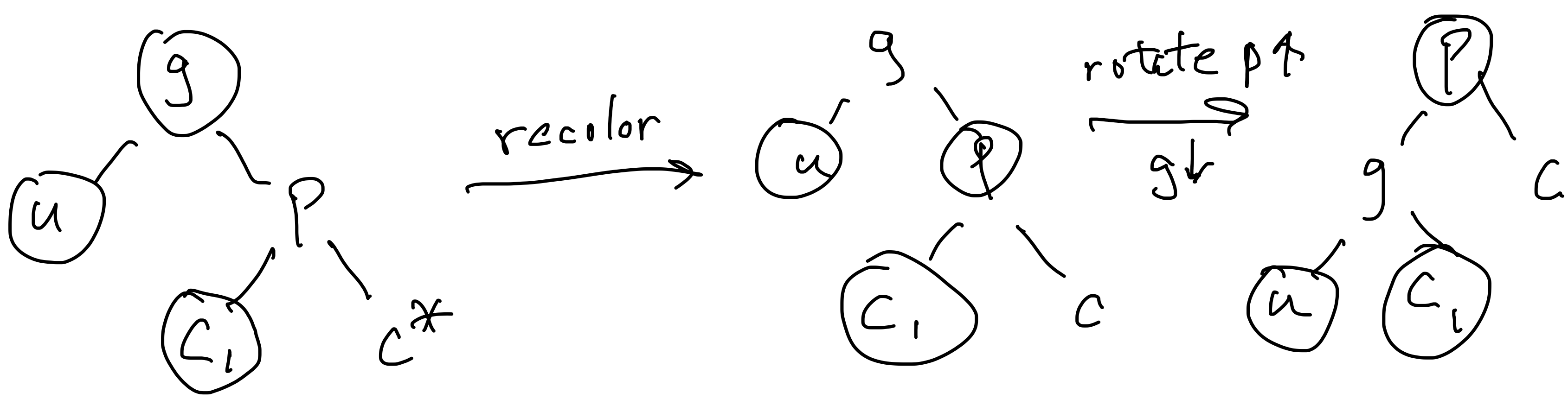
Case 2

parent red, uncle black, c inside

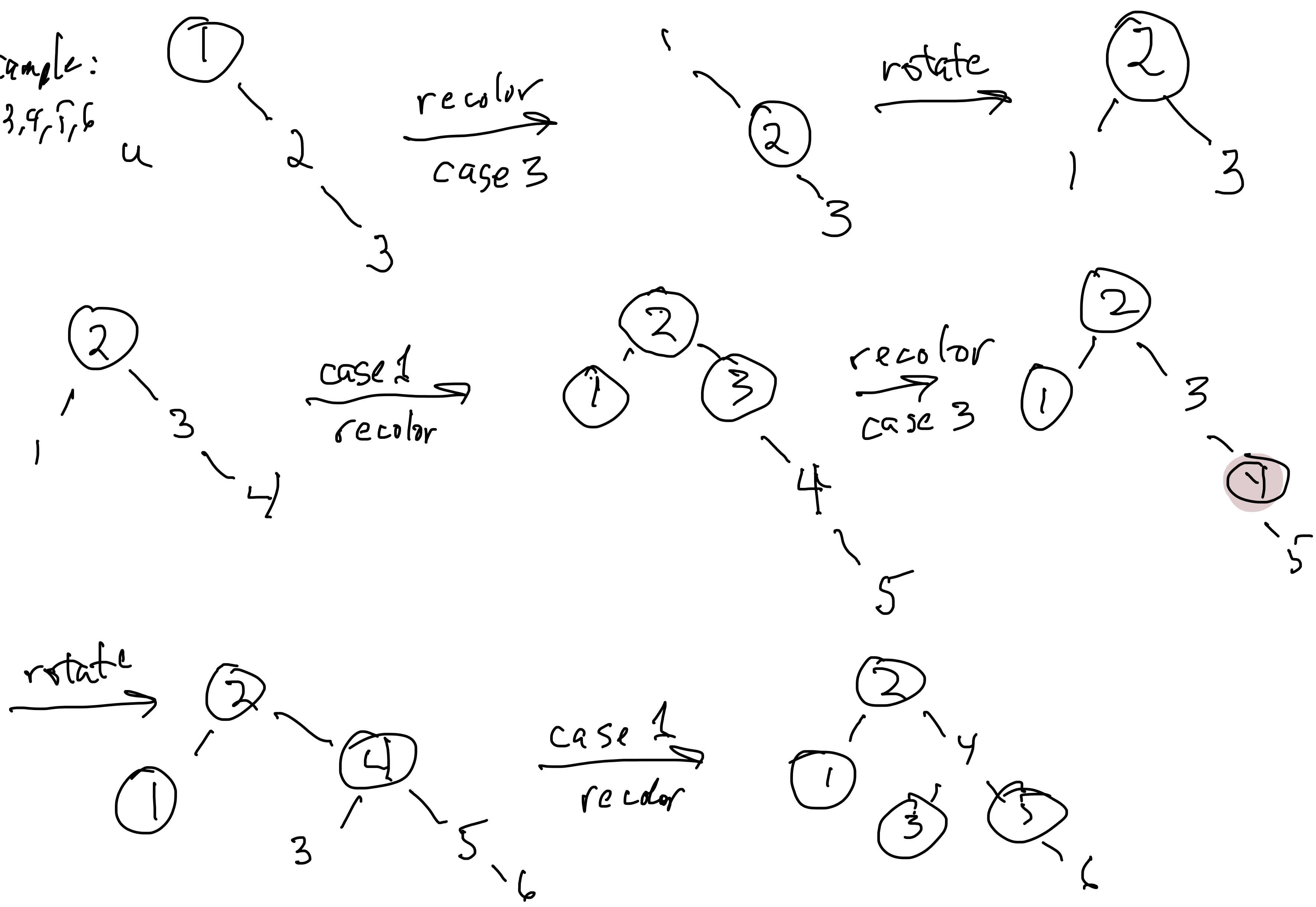



Case 3

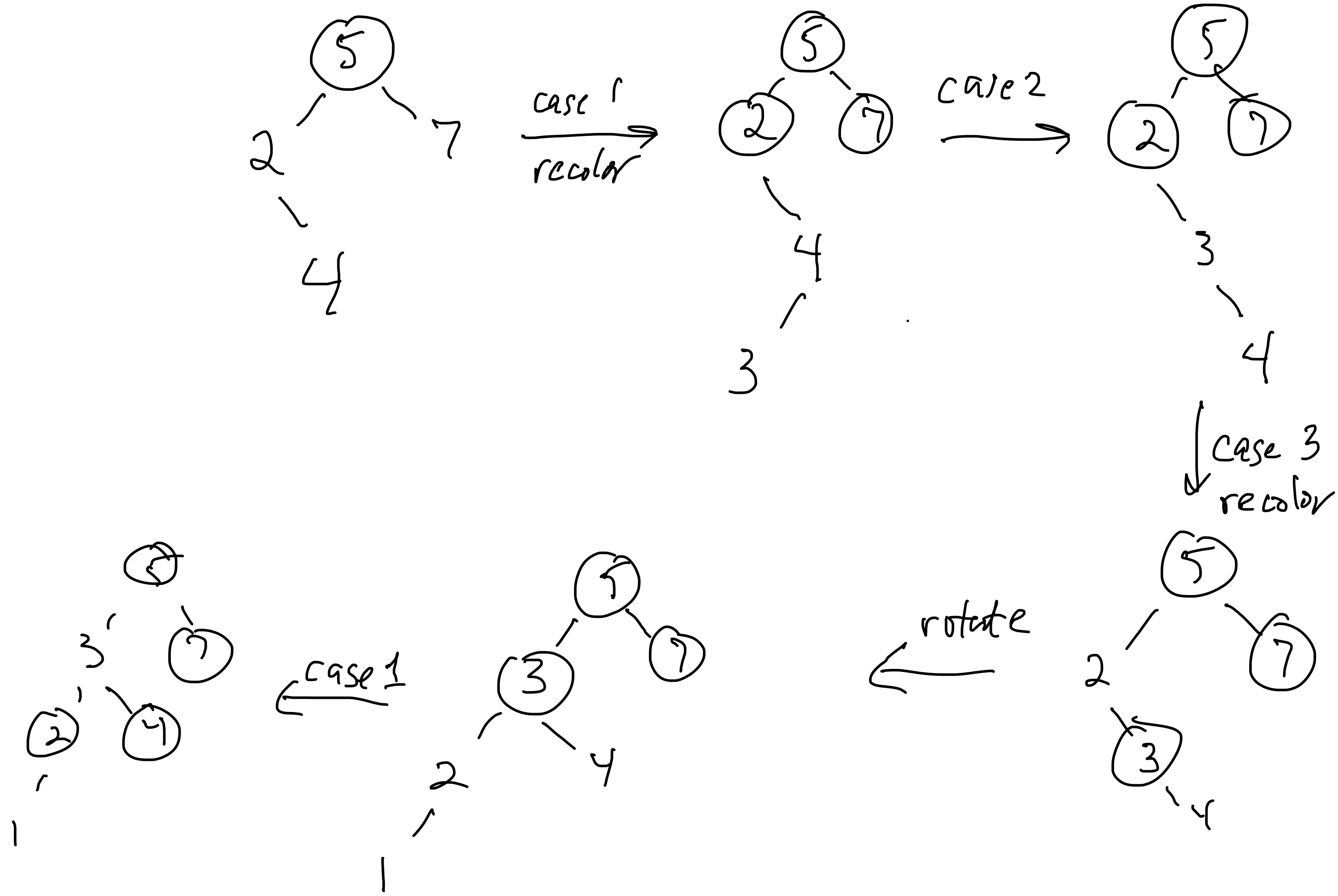
parent red, uncle black, c outside




Example: 1,2,3,4,5,6



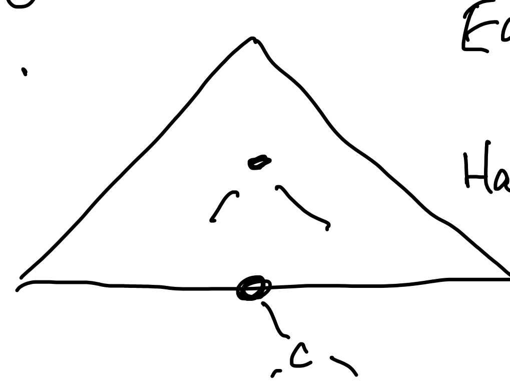
Example: 5 2 7 4 3 1




Properties of binary trees

1) Expected worst path is $\Theta(\log n)$ deep.
   If random data.
2) Can have $\Theta(n)$ worst path length.
3) Can use a self-balancing tree (Red black, AVL)
4) Insertion: $\Theta(\log n)$
5) Deletion: ugly.

delete node d

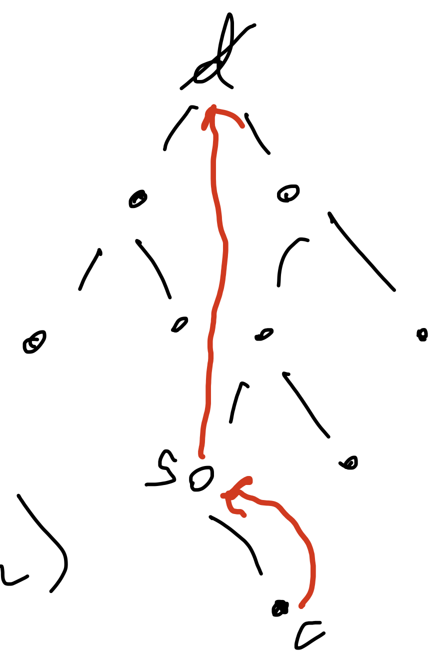



Easy: d is a leaf
remove it.

Harder: d has 1 child c:
move c into d's
place

Hardest: $d$ has 2 children.

find $d$'s successor $s$ : $s = RL^*$



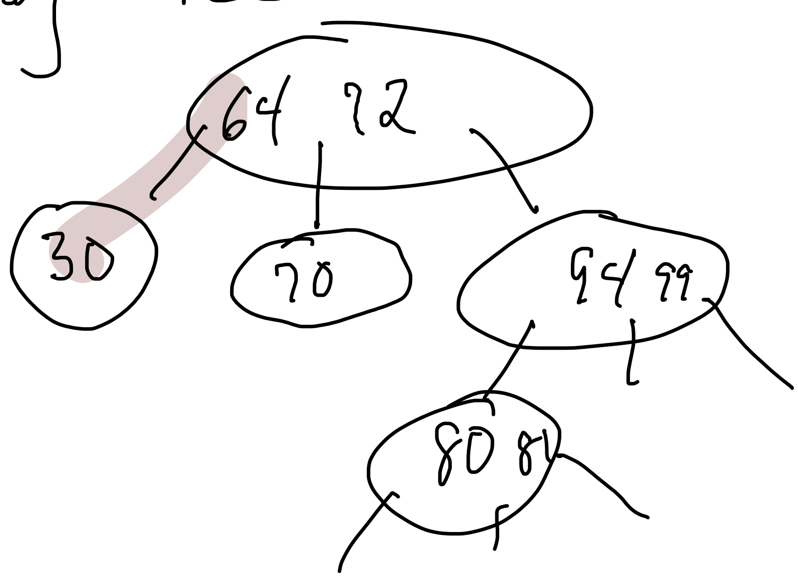

replace $d$ with $s$.

reparent $d$'s right child ($c$) in place of $s$.

$\Theta(\log n)$

---

Ternary trees

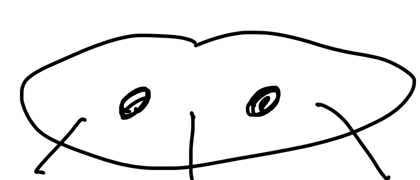



Complexity for searching

lucky: balanced. Depth $= \Theta(\log n)$

$\log_3 n$

work at each level



# comparisons 1 2 2

avg # comparisons = $5/3$

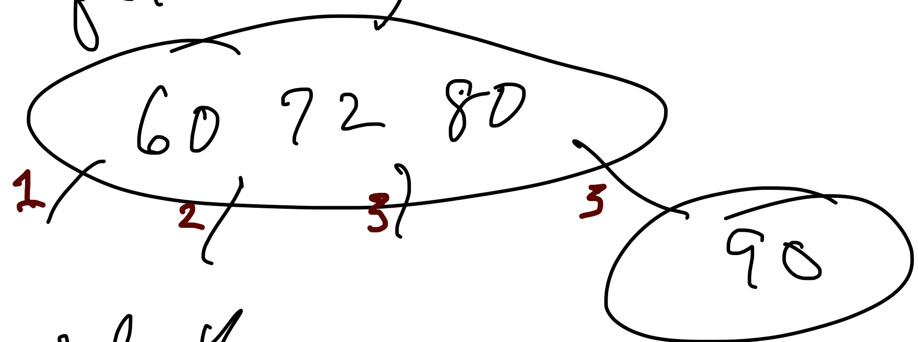
total work for searching $\frac{5}{3} \log_3(n)$

compare to binary tree: $1 \log_2(n)$

ternary tree is 1.05 cost of binary tree

5% degradation.

what about quaternary tree?




$\log_4 n$ : depth

$\frac{9}{4}$ work at each level : $\frac{9}{4} \log_4 n$

12.5% degradation in comparison with binary.

---

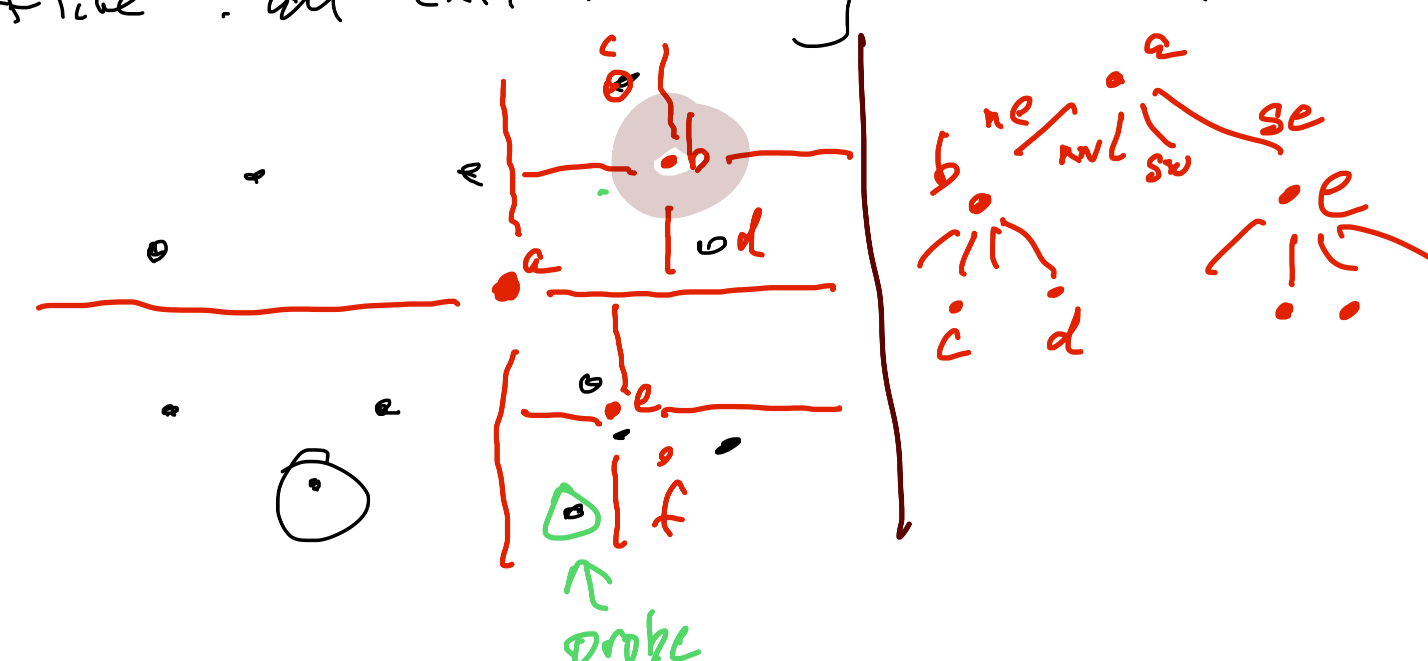
Other generalization?

Quad trees (Finkel, 1973)

2-dimensional data.

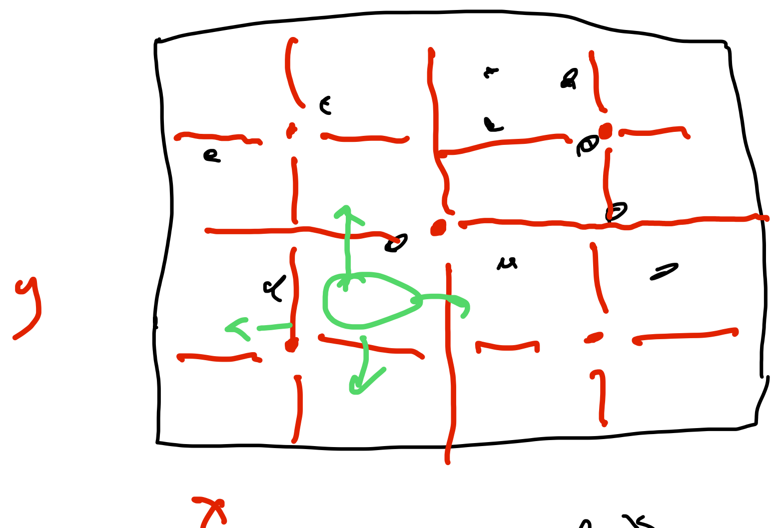
online : data appear 1 by 1.

vs offline : all data are already available.



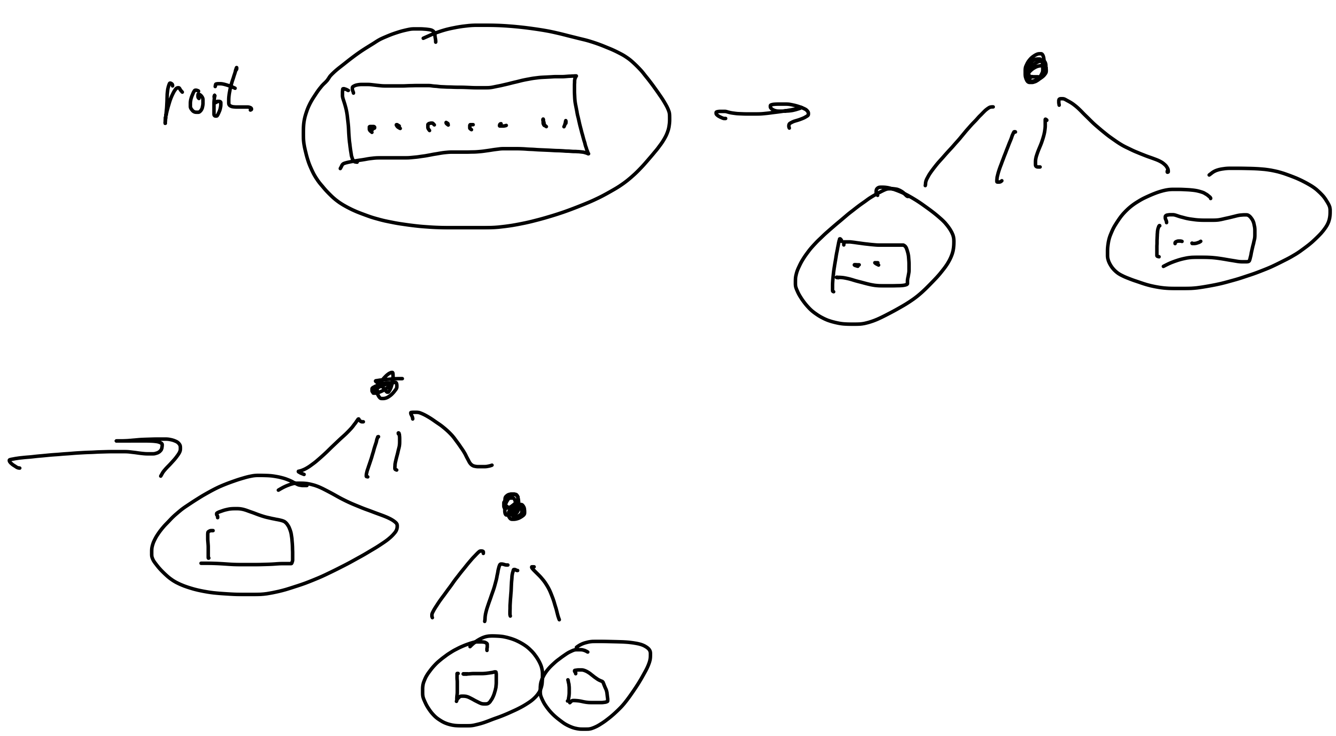
options:

1. only use points as tree nodes
2. stop subdividing when region holds only a few points
   about 10 is a good "bucket" size.
3. place tree nodes in center of their region.




4. place tree nodes at "median" location
   trying to balance the # of points in
   each quadrant.

Online: put all first points in to a bucket.
when it fills, split it and build a root of tree.

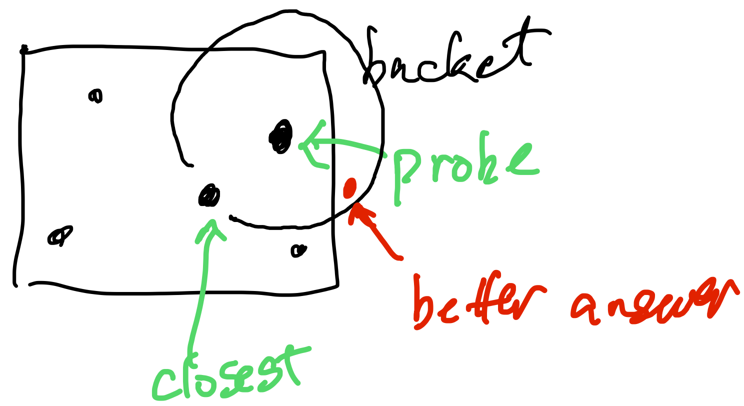



good for
1. searching
2. (Jon Bentley) nearest neighbor search.

probe: 1) dive down tree to reach bucket that the probe would be in.

Those bucket points are close to probe.

2) Consider them all, find closest among them.

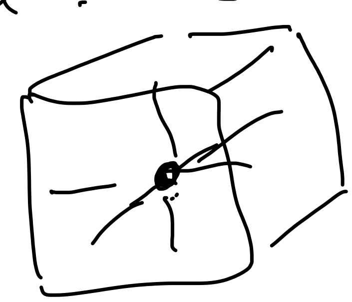



3) go to neighboring regions.

---

Extend good trees to higher dimensions?

yes: 3d. Oct Trees

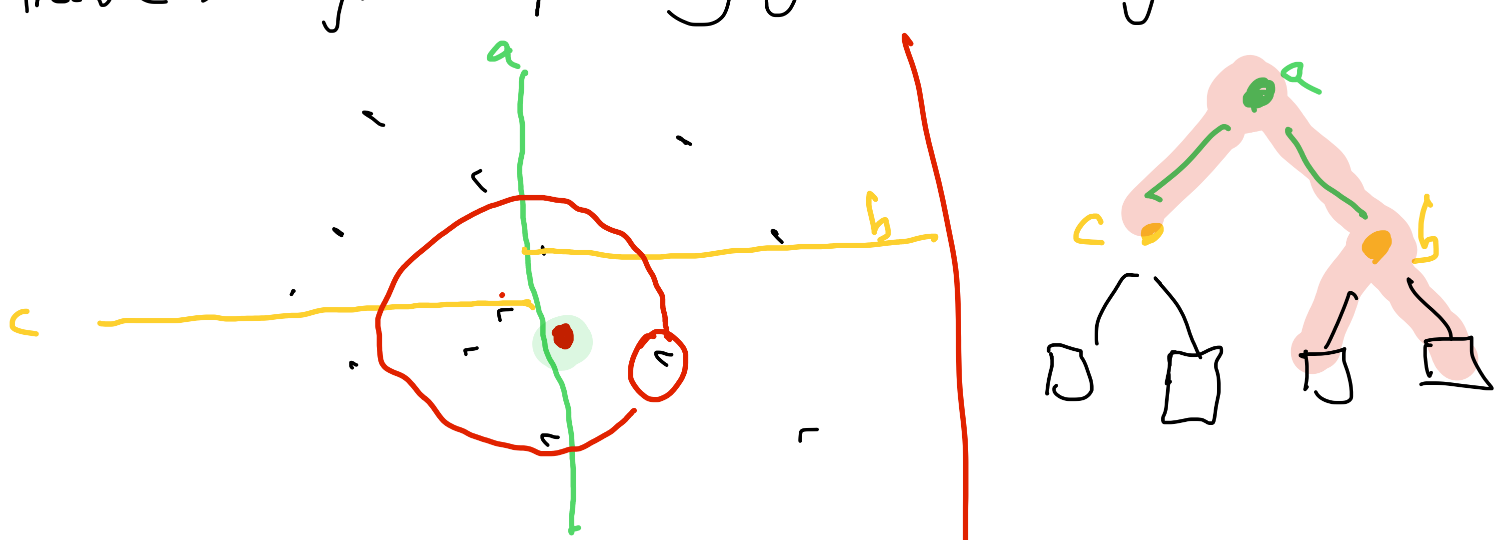


each internal node has 8 children.

people use oct trees to represent 3-d images and objects.

---

K-d trees (Bentley): especially good for high dimensions

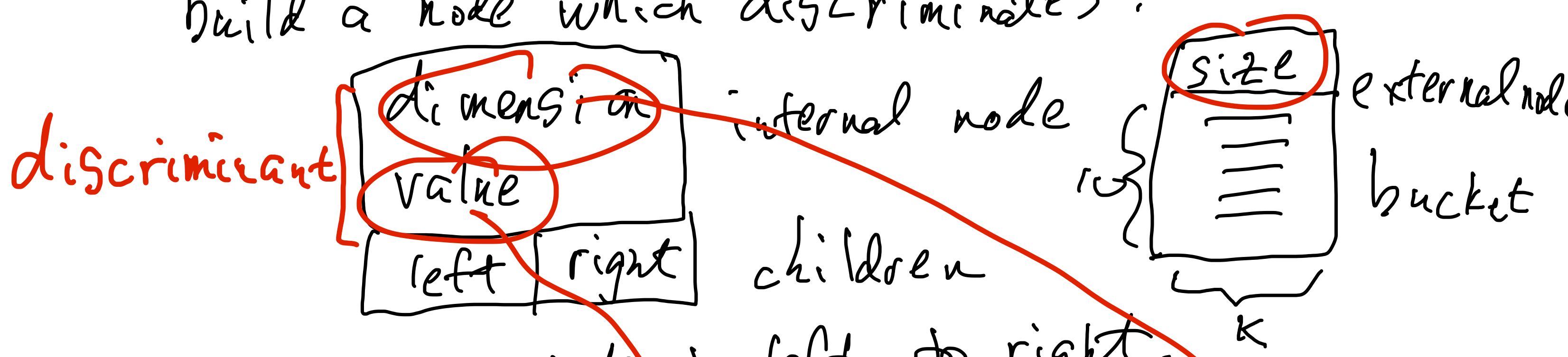


Rule: given a cloud of points:

find the dimension with greatest range.

in that dimension, find median value.

build a node which discriminates:




recurse on points to left, to right.

those whose value in the dimension are less than or equal to the discriminating value.

Result: balanced tree,

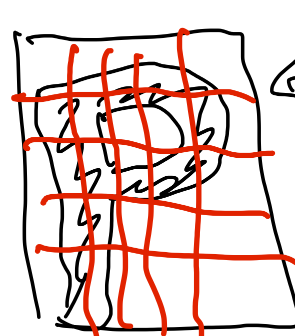
leaf nodes are buckets (with $b \simeq 10$ at most points)

Good for:

representing clouds of high-dimensional data

finding nearest neighbors.

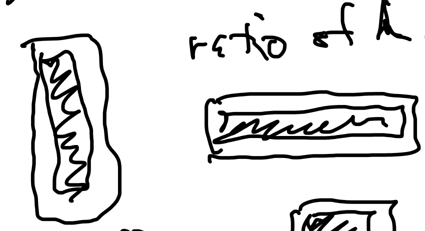
Optical character recognition:



← rectangle surrounding a letter

distinguish I from —

ratio of h/w

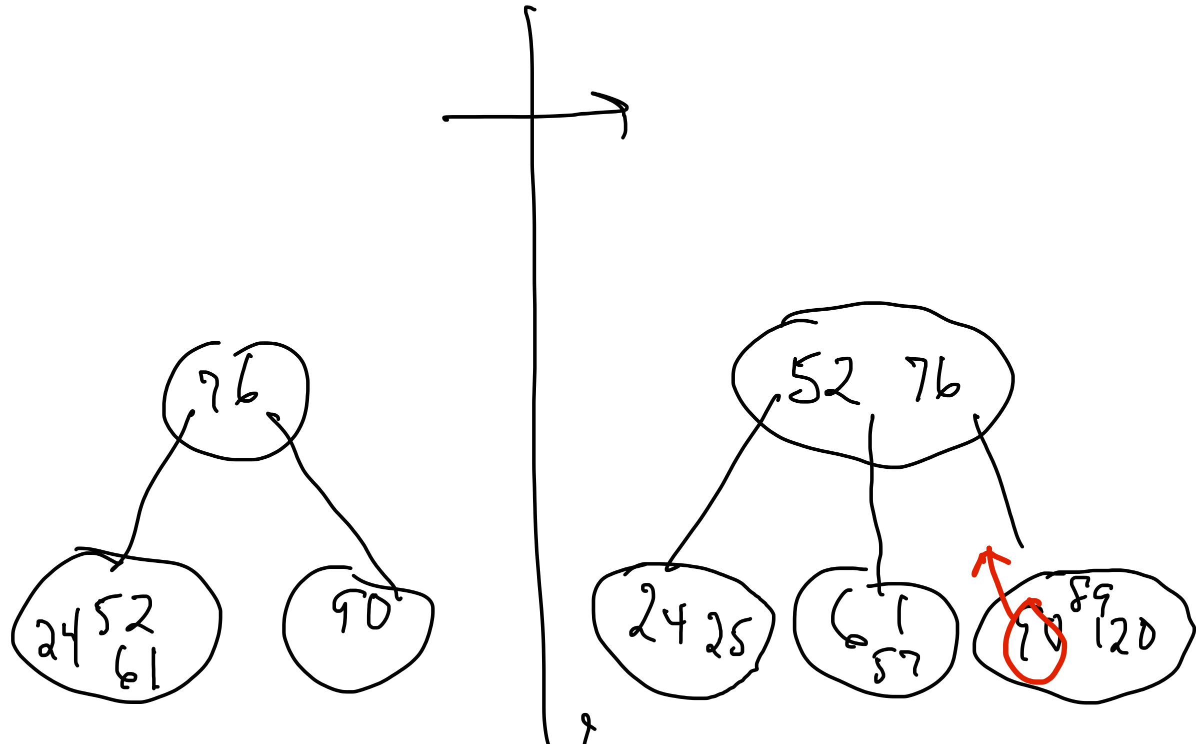


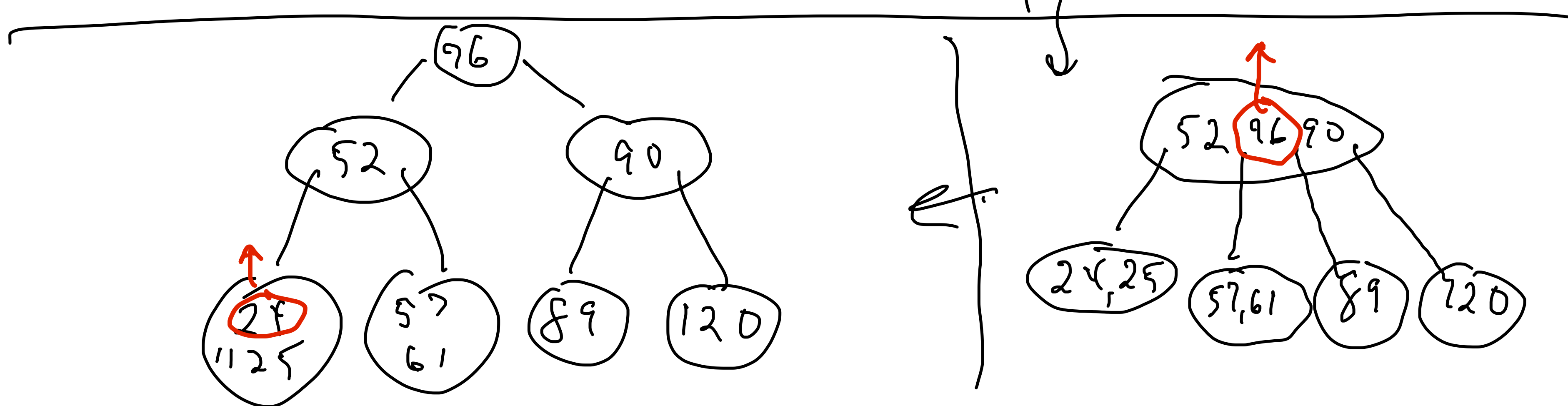
vector of 25 fractions.

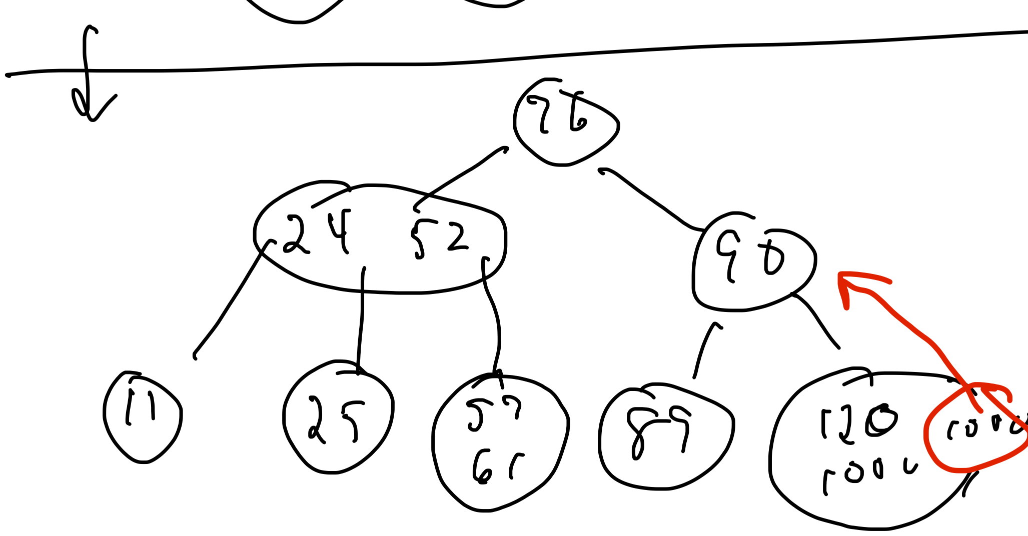
+ 1 element: aspect ratio

⇒ point in 26-dimensional space

# 2-3 trees :




duplicates are tricky:
can we always break ties to the left?

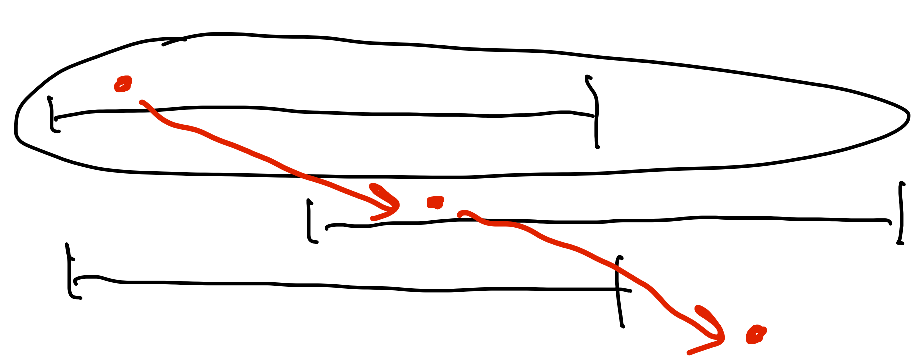
$\Rightarrow$ no duplicates allowed.

$\Rightarrow$ or recursive search must look 2 ways ~~after tie~~

Insertion, search: $\Theta(\log n)$

---

Stooge sort



Analysis:

$$C_n = 1 + 3C_{\frac{2}{3}n}$$

a $3$

b $3/2$

k $0$ (because $1 = n^0$)

$a \quad b^k$

$3 > 1$

$\Theta(n^{\log_b a})$

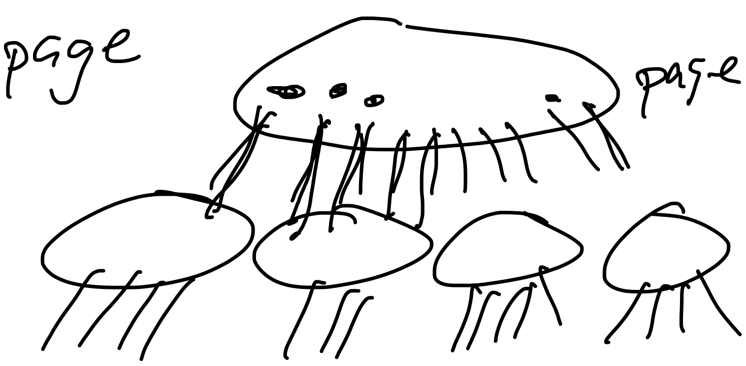
$\Theta(n^{\log_{3/2} 3})$

$\approx \Theta(n^{2.71})$

---

B-trees: extension of 2-3 trees.

Bucket: fits in a disk page




block: 4KB

element { value: 4B
index to a disk block: 4B }

4KB/8B = 512 elements/block

$m = 512$ ($m$ "capacity" of a node)

$m = 3 \Rightarrow$ 2-3 tree

Use $g = \lceil m/2 \rceil$ as the "fill" amount for new nodes.

capacity of a node:

$m-1$ values, $m$ pointers

$g = \lceil \frac{m}{2} \rceil$ (half size)

Internal nodes: $g \ldots m$ children.

exception: root: $1 \ldots m$ children.

insertion: 1) find the right leaf node

2) insert value in node

3) if node is overfull ($m$ values)

split it: Hoist middle value to parent, remaining values: $g$ of them as a new node (left) others as a new node (right). Adjust pointers in parent to point to the resulting nodes.

4) If parent is overfull, split it, and if necessary, continue up the tree to root (which itself can split, generating a new root).

Height of tree (full): $\log_m(n)$

$\Theta(\log n)$.

Deletion of a value $v$. Difficult.

rare → 1) Internal node: Replace $v$ in that node with successor($v$). Then continue with deletion from leaf.

2) Leaf:

common → good case: leaf is still $g$ or more values.

rare → bad case: leaf node is under-full ($< g$ values)

steal a value from a neighbor.

(B* tree: nodes also point to neighbors)

if all neighbors are almost under-full:
merge with one neighbor, taking a value
from parent, ...

---

Summary of trees

in use → Binary (sorted, insertion O(log n), deletion hard)
                                              ↑ expected

traversals (inorder, postorder ...)

Higher arity (ternary, ...)

not practical: cost of dealing with nodes overwhelms benefit of shallower tree

in use → Self-balancing (Red-black, AVL)

guarantee of O(log n)

insertion is more complicated.

in use → Higher dimension

quad, oct

K-d

Other organizations to guarantee balance

2-3 tree

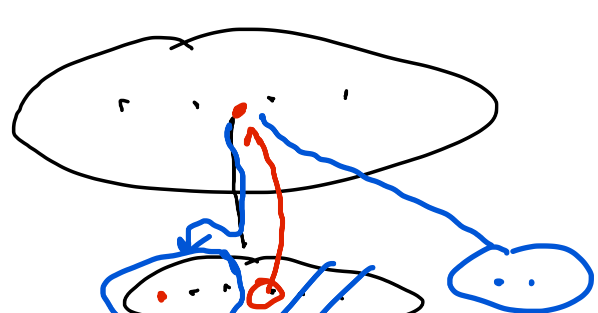
B-tree

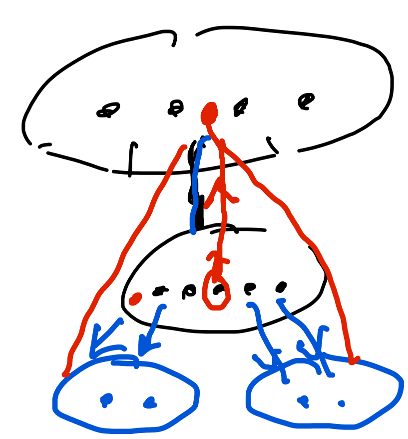
---

Moral:

trees are essential

recursion is essential.





Searching: $\Theta(n)$ array

$\Theta(\log n)$ sorted array

tree

$\Theta(1)$

Hashing

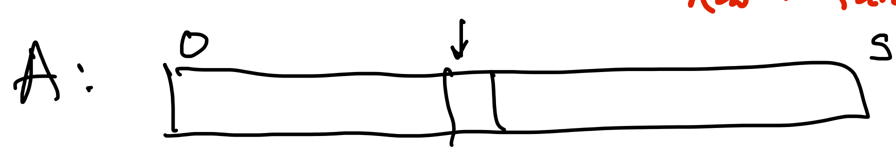
expected, not guaranteed

$n$ elements $\Rightarrow$ key is at least $(\log_2 n)$ bits.

Basic idea: use an array A[ ]

place key k in array at index $h(k)$

hash function

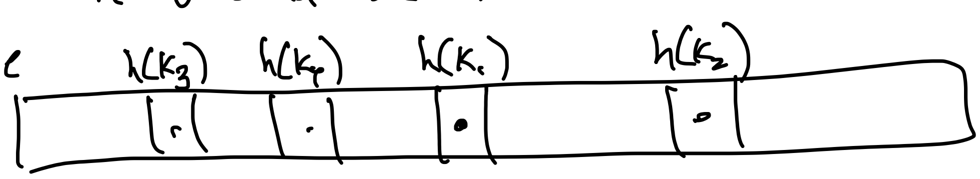



Keys are usually strings

$h(\text{string}) =$ small integer $(0 .. s-1)$

h should be fast.

example




$K_1 =$ once $\quad K_3 =$ a

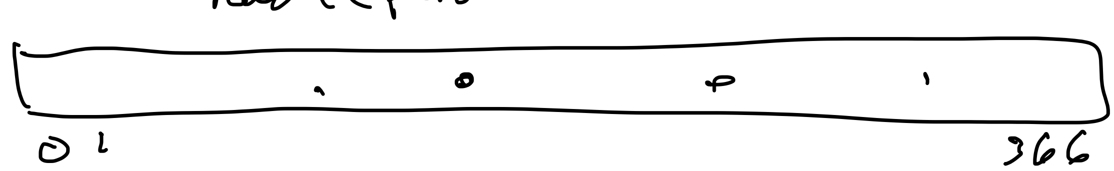
$K_2 =$ upon $\quad K_4 =$ midnight

problem: collisions. Two keys that hash to the same index.

Collisions are unavoidable.

Birthday paradox

hash(person) = birthdate



Prob (no collisions with $j$ students)

$$= \frac{365 \cdot 364 \cdot 363 \cdots (365-j+1)}{365^j}$$

$$= \frac{365!}{(365-j)!} \frac{1}{365^j}$$

$j = 23 \Rightarrow \text{Prob} < 1/2$

$j = 50 \Rightarrow \text{Prob} = 0.029$

Dealing with collisions.

open addressing: if there is a collision, use another location in the array.

disadvantage: clustering

disadvantage: deletion

(perfect hashing: if know all keys in advance, design $h()$ to avoid collisions.)

1) Linear probing. Multiple tries for key $K$.

$p_0 = h(K)$

$p_1 = (h(K)+1) \bmod S.$

$\vdots$

$p_j = (h(K)+j) \bmod S.$

insert: at first empty cell starting at $h(k)$.

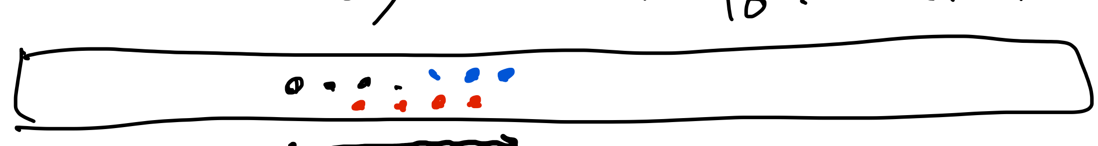
search: $p_0$ (check!), $p_1$ (check!) ...

stop when

1) find key! success

2) find empty slot. failure

3) back to $p_0$. failure (full)



2) Family of hash functions.

$p_0 = h_0(k) \quad \ldots \quad p_j = h_j(k)$

$p_1 = h_1(k)$

hope: avoid clusters merging.

worse and worse as table fills.