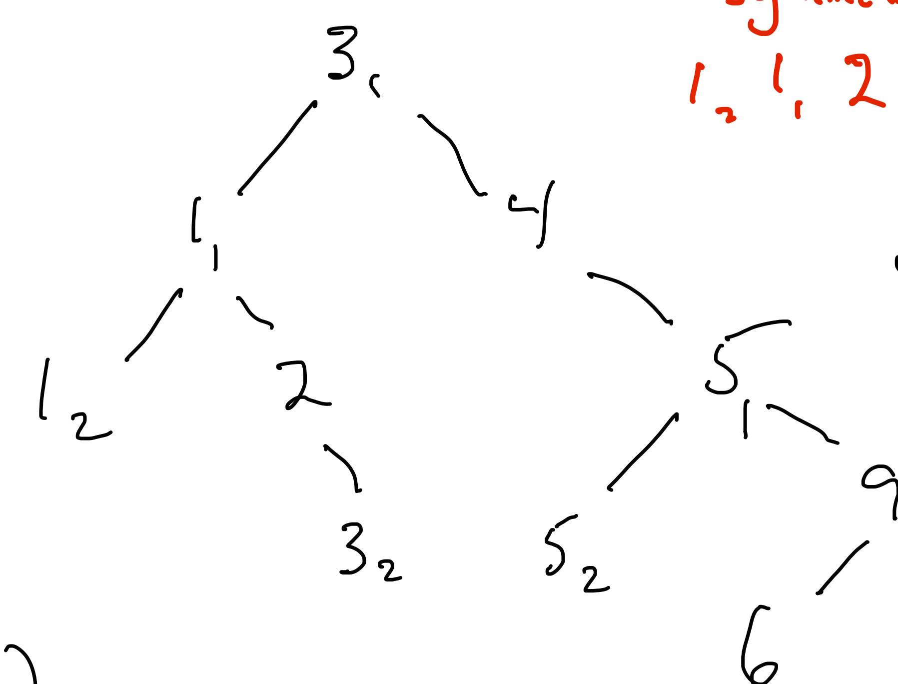
Data structures Packet 3

3_1, 1_1, 4, 1_2, 5_1, 9, 2, 6, 5_2, 3_2

binary tree

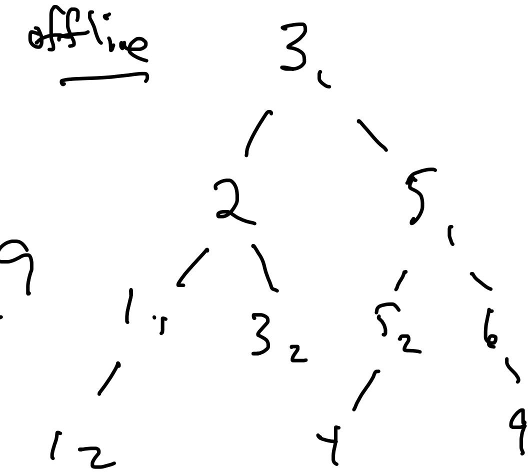
online



symmetric (inorder)

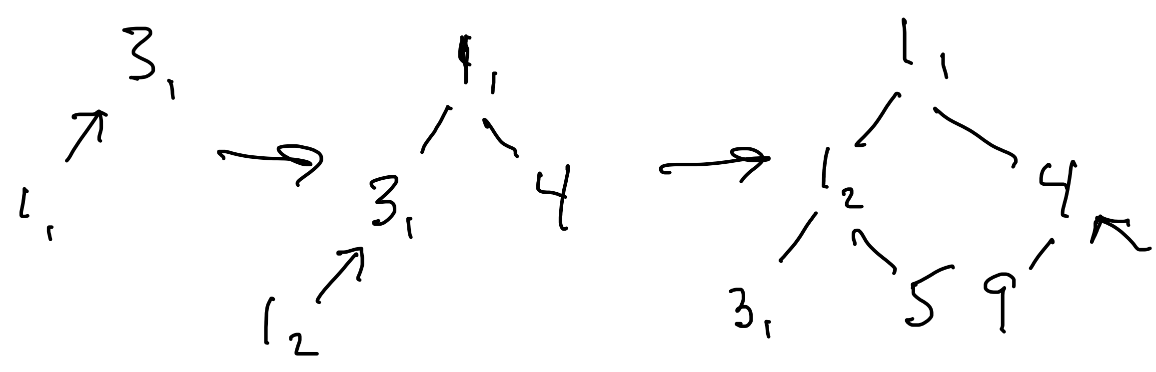
1_2 1_1 2 3_2 3_1 4 5_2 5_1 6 9

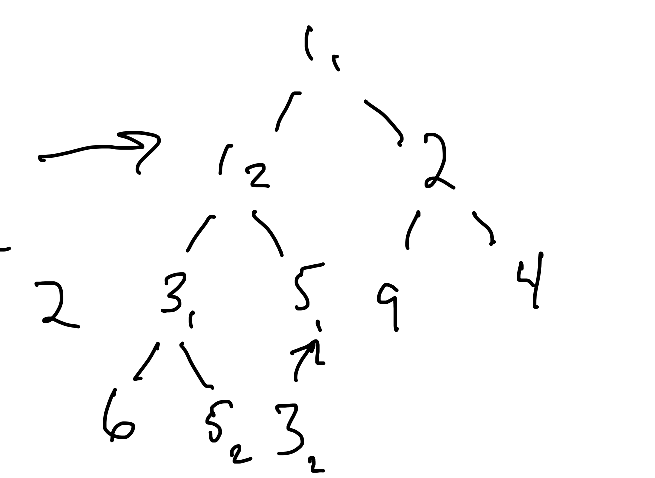
offline



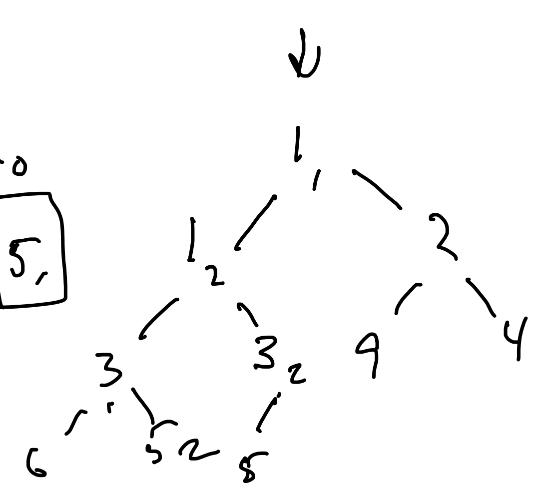
heap (top light)

3_1 1_1 4 1_2 5_1 9 2 6 5_2 3_2





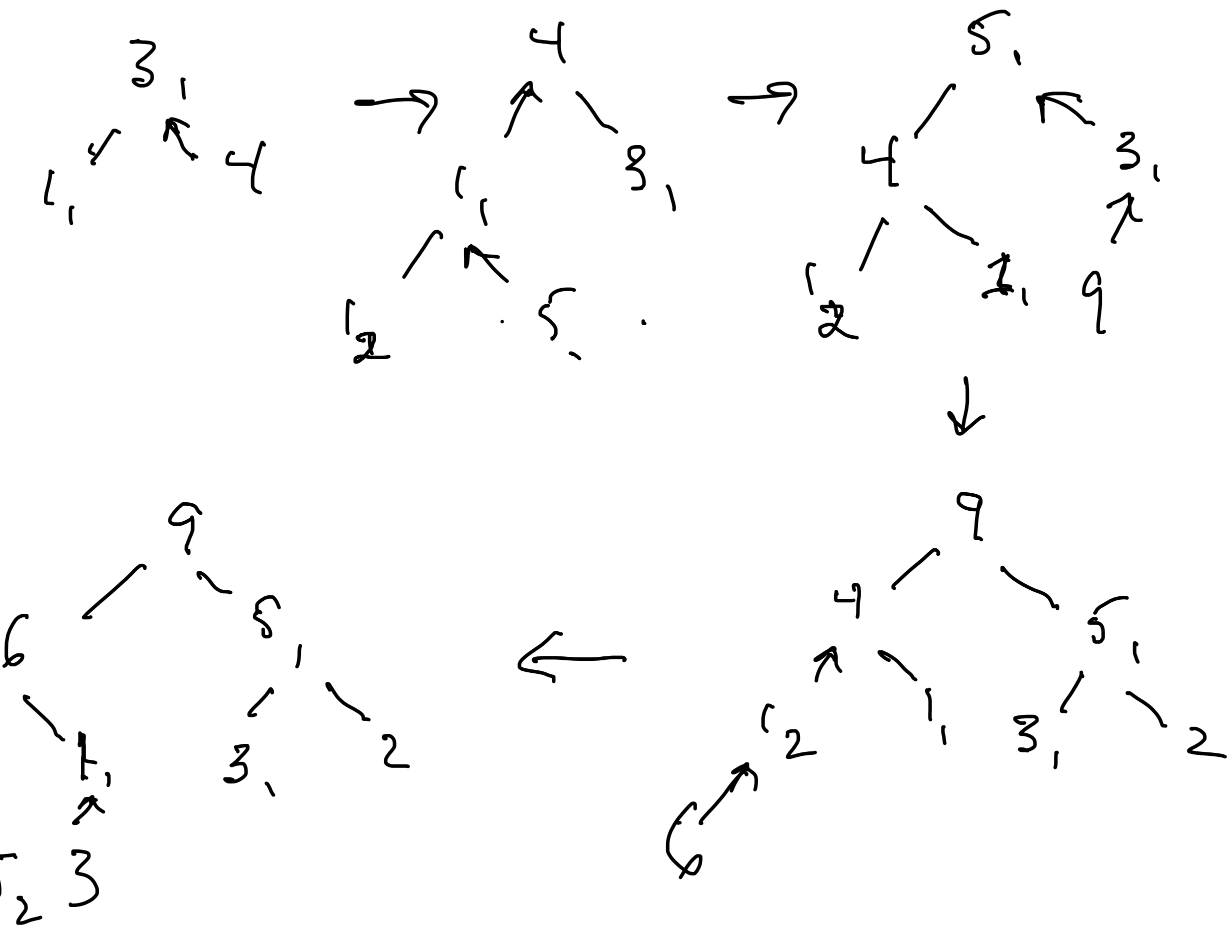
1	1	4	3	5	~~9~~	2	

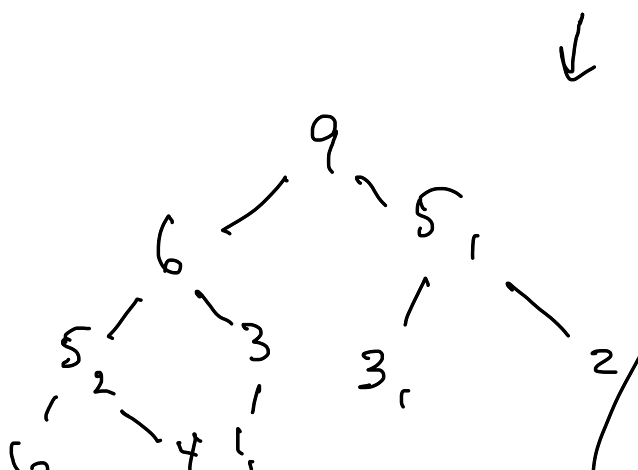


1	2	3	4	5	6	7	8	9	10
1_1	1_2	2	3_1	3_2	9	4	6	5_2	5_1

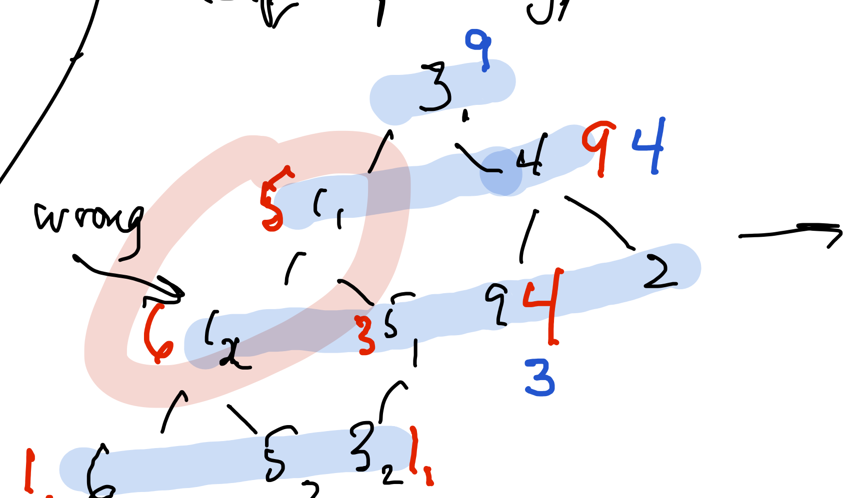
$3_1 \quad 1_1 \quad 4 \quad 1_2 \quad 5_1 \quad 9 \quad 2 \quad 6 \quad 5_2 \quad 3_2$

heap (top-heavy)
online



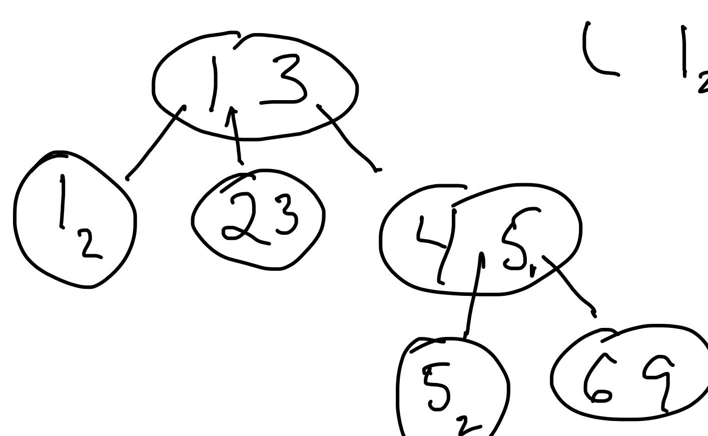


heap top heavy offline

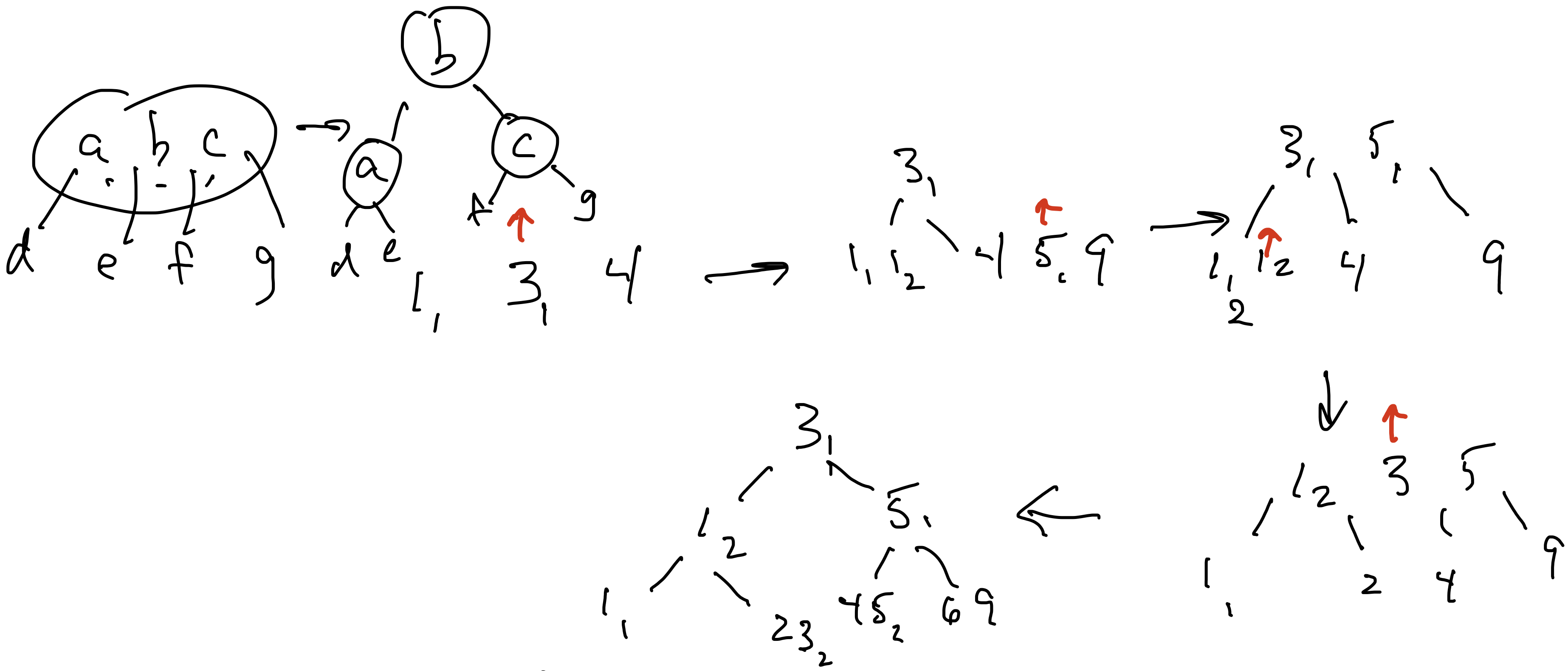


ternary tree (online) $3_1 \quad 1_1 \quad 4 \quad 1_2 \quad 5_1 \quad 9 \quad 2 \quad 6 \quad 5_2 \quad 3_2$

$(\ 1_2\ )\ 1_1 (\ 2\ \ 3\ )\ 3 (\ 4(5)\ 5(6\,9))$



2-3 tree (online) $3_1 \; 1_1 \; 4 \; 1_2 \; 5_1 \; 9 \; 2 \; 6 \; 5_2 \; 3_2$



pre-order traversal:

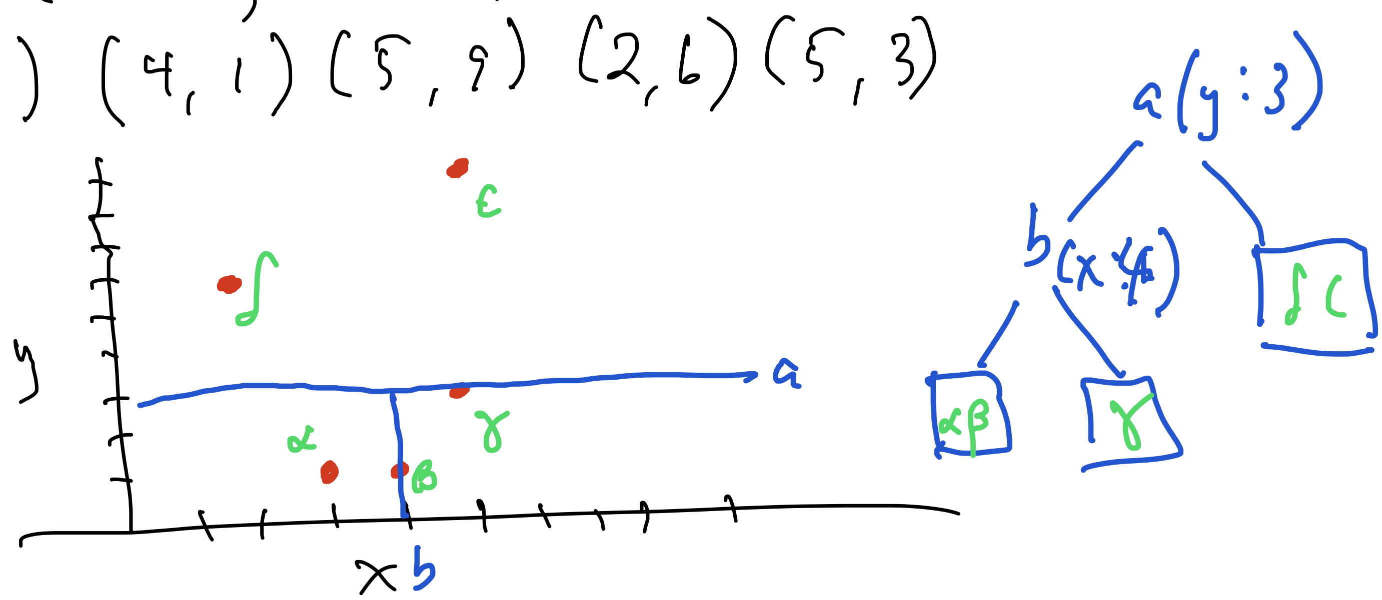
$3_1 \; 1_2 \; 1_1 \; 2 \; 3_2 \; 5_1 \; 4 \; 5_2 \; 6 \; 9$

post-order traversal:

$1_1 \; 2 \; 3_2 \; 1_2 \; 4 \; 5_2 \; 6 \; 9 \; 5_1 \; 3$

k-d tree (2 dim, offline, bucket size 2)

(3, 1) (4, 1) (5, 9) (2, 6) (5, 3)



Selection sort : 5 steps

$| \; 3_1\ 1_1\ 4\ 1_2\ 5_1\ 9\ 2\ 6\ 5_2\ 3_2$

$1_1 \; | \; 3_1\ 4\ 1_2\ 5_1\ 9\ 2\ 6\ 5_2\ 3_2$

$1_2 \; | \; 4\ 3_1\ 5_1\ 9\ 2\ 6\ 5_2\ 3_2$

$2 \; | \; 3_1\ 5_1\ 9\ 4\ 6\ 5_2\ 3_2$

$3_1 \; | \; 5_1\ 9\ 4\ 6\ 5_2\ 3_2$

$3_2\ 9\ 4\ 6\ 5_2\ 5_1$

$1_1\ 1_2\ 2\ 3_1\ 3_2 \; | \; 9\ 4\ 6\ 5_2\ 5_1$

Insertion sort

$3_1 \; | \; 1_1\ 4\ 1_2 \; | \; 5_1 \; | \; 9\ 2\ 6\ 5_2\ 3_2$

$1_1\ 3_1 \; | \; 4 \; |$

$1_1\ 1_2\ 3_1\ 4 \; | \; 5$

Quicksort (1 partitioning step, Lomuto)

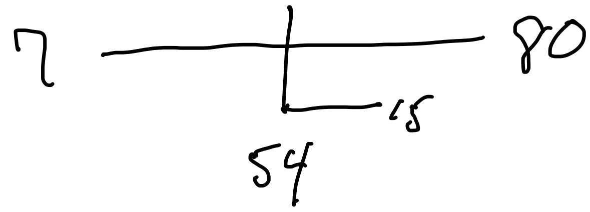
$3_1\ 1_1\ 4\ 1_2\ 5_1\ 9\ 2\ 6\ 5_2\ 3_2$

Pivot

$1_2 \quad 4 \quad 4 \quad 2 \quad 5 \quad 3$

$3\ 1\ 1\ 2 \quad 3 \quad 9\ 4\ 6\ 5\ 5$

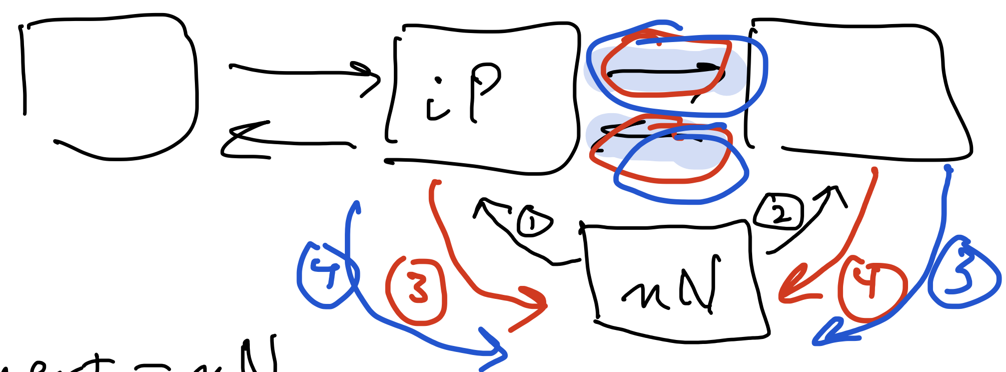
Score Range on Midterm



2b (423):

$\Theta(\ ) \Rightarrow \Theta(1)$

1a



③ iP → next = nN

④ nN → next → pred = nN

③ iP → next → pred = nN

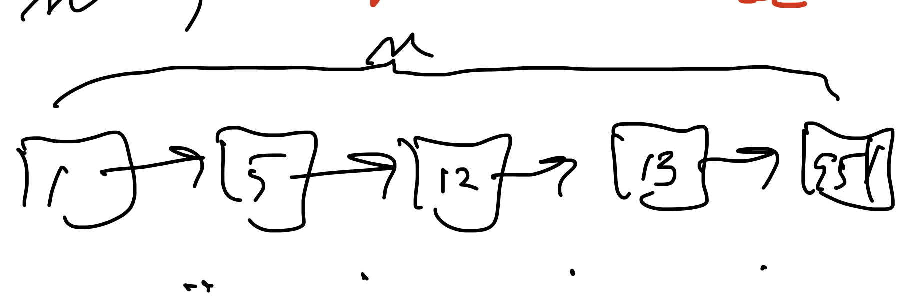
④ iP → next = nN

1b complexity: $\Theta(n)$ average or worst-case

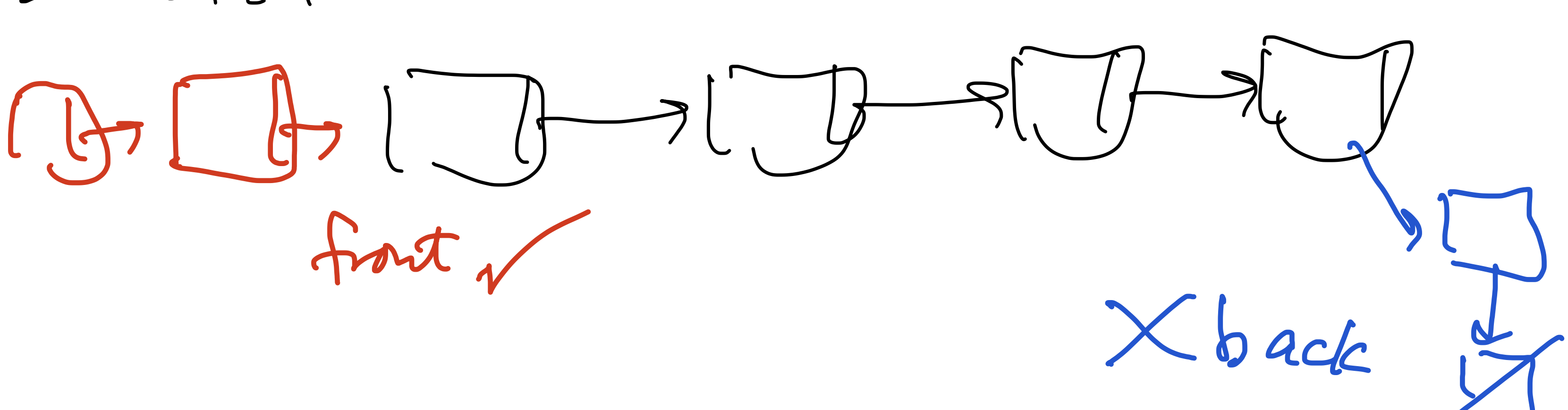
insertion:

sorted list

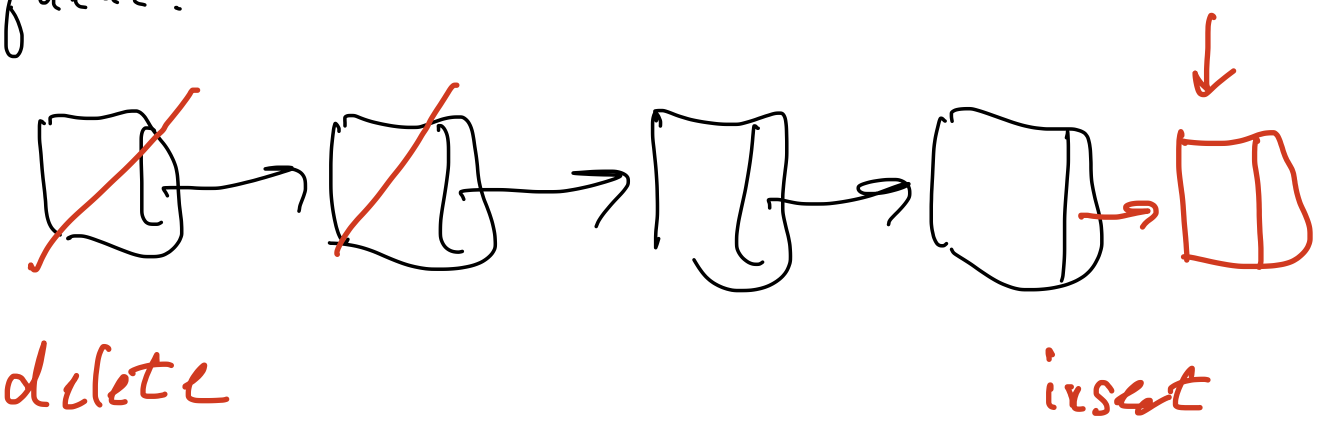
singly-linked



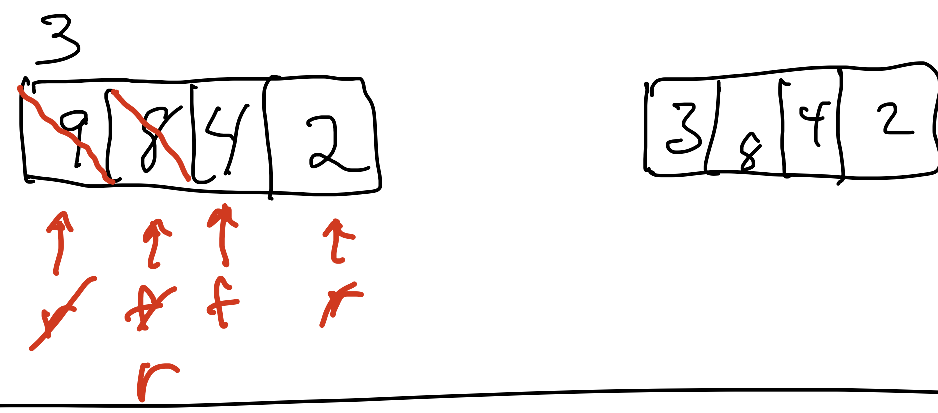
1c stack



1d queue.



2a



2b

~~7~~ ~~2~~ 4 ~~1~~ ~~8~~ ~~3~~ 6 9 5 ~~0~~

2 4   4 8   7

0 1 2 3 4

8 6 9 5 7

2c

7 2 4 1 8 3 6 9 5 0

comp

3   8 6   8   5   0   8   9

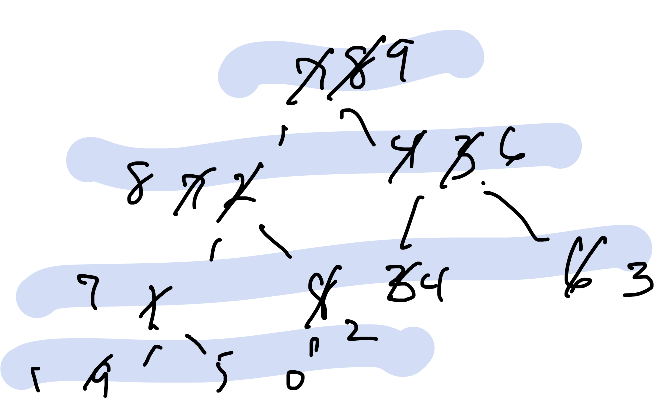
end of small values

~~7~~ 2 4 1 3 6 5 ~~0~~ 8 9

0   7

2d

7 2 4 1 8 3 6 9 5 0

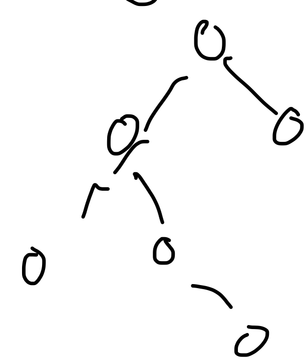


0	1										
∞	9	8	6	7	2	4	3	1	5	0	

3 a sorted binary tree

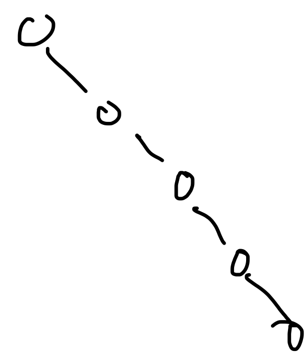
expected complexity $\Theta(\log n)$

insertion



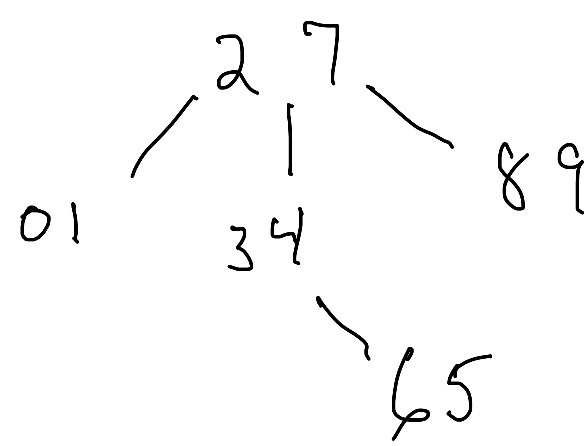
$\Theta(h)$

b worst case

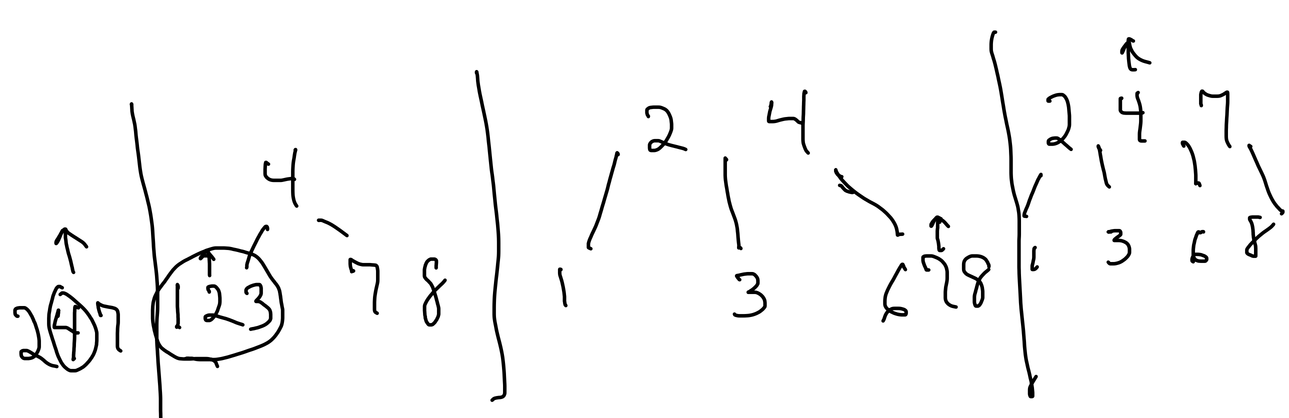


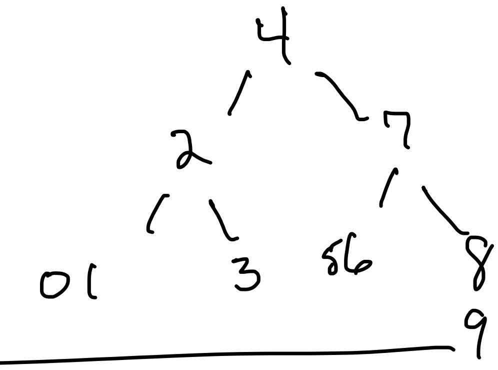
$\Theta(n)$

c. 7 2 4 1 8 3 6 9 5 0

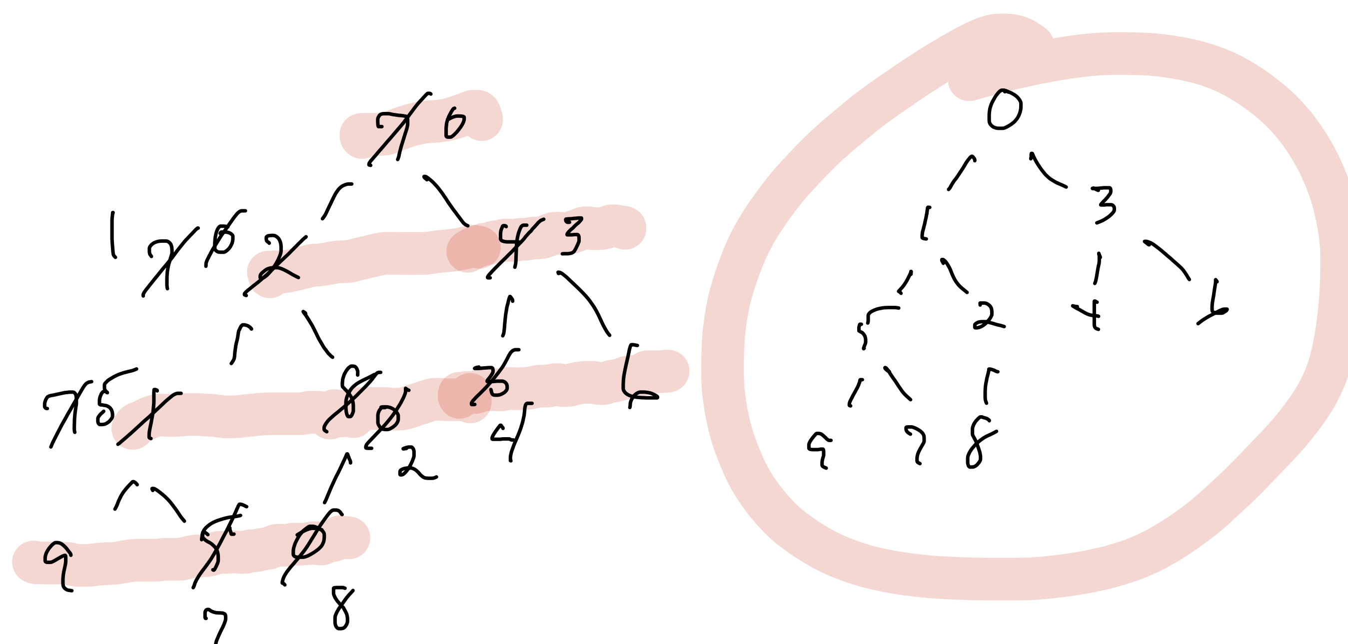


3d 7 2 4 1 8 3 6 9 5 0





3e 7 2 4 1 8 3 6 9 5 0

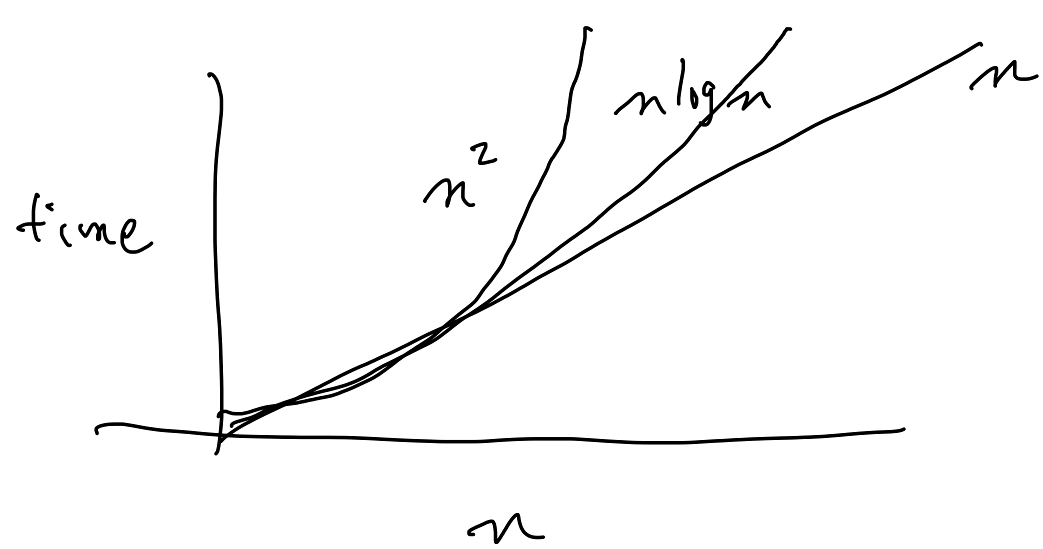


4 a) Why heap instead of bubble sort?

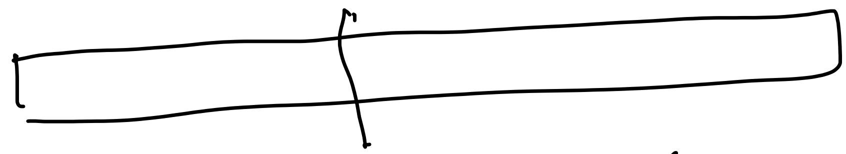
$\Theta(n \log n)$ guaranteed.

$\Theta(n^2)$

$\Theta(n^2)$



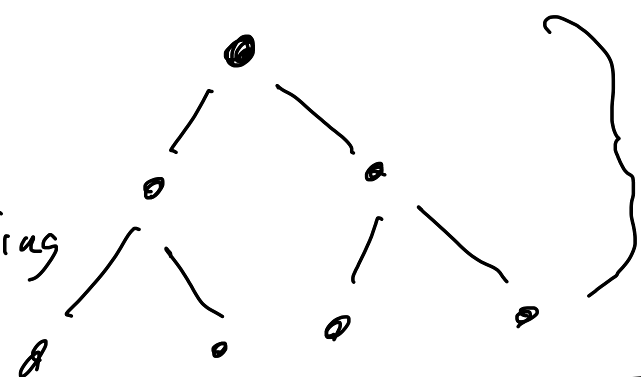
4b. why median of 3 (or 5)?



step 1: more likely to get a more balanced partition.

step 2: better balance → shorter navigation tree

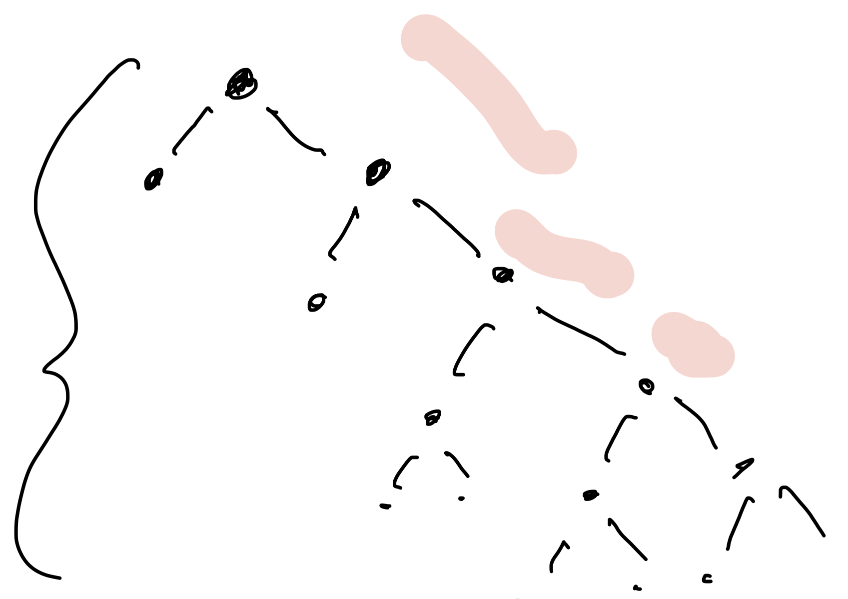
perfect partitioning



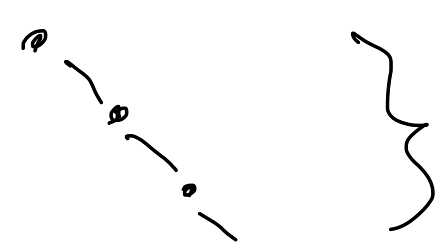
log n

ok partitioning

$\Theta(\log n)$



terrible partitioning



$\Theta(n)$

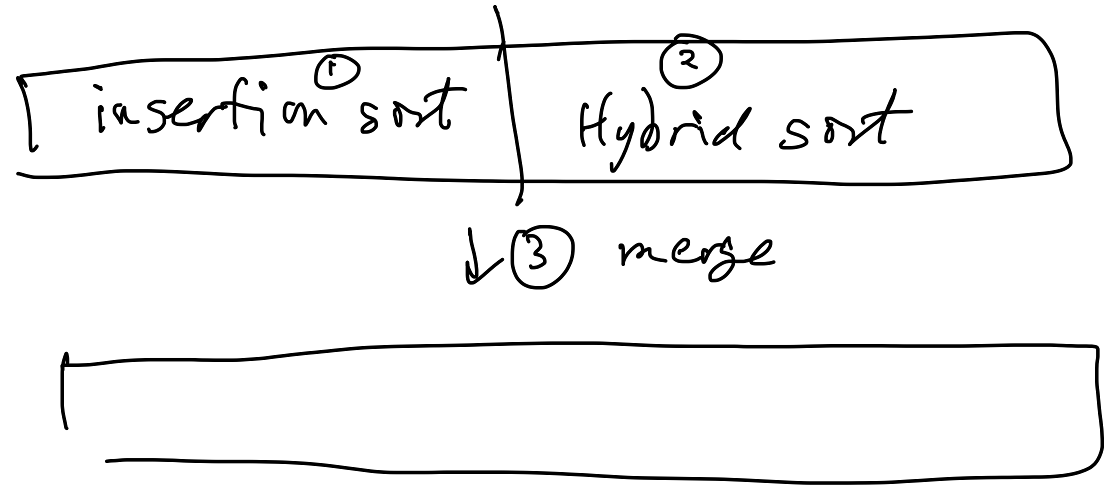
4c: 2nd largest element.

sort, index: $\Theta(n \log n)$

quick select $\Theta(n)$ expected.

1 pass: $\Theta(n)$

4d: Hybrid sort



$$C_n = f(n) + a\, C_{n/b}$$

$$n^2 + n + 1\, C_{n/2}$$

$$C_n = n^2 + C_{n/2}$$

$k = 2$
$a = 1$
$b = 2$

a	b^k
1	4

$\Theta(n^k)$

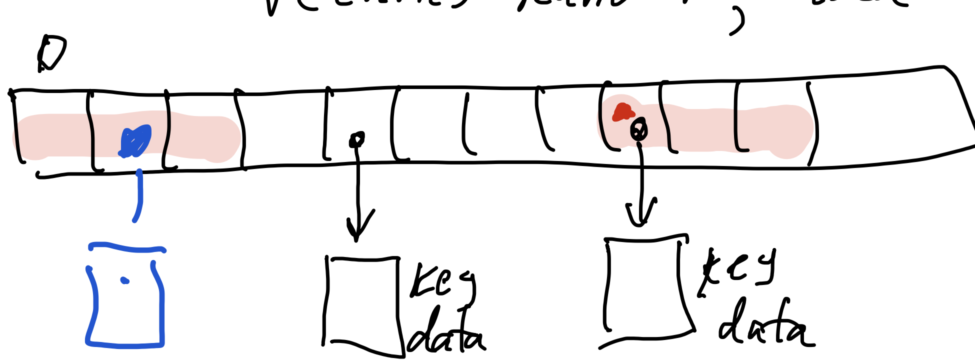
$\Theta(n^2)$

Hashing

Key (typically a string) k

apply a hash function $h(k)$

returns number, used as an array index



open addressing: if there is a collision, place the new key in some other cell.

method: linear probing

$$p_i = (h(k) + i) \bmod s \quad i = 0 \ldots$$

very bad: clusters

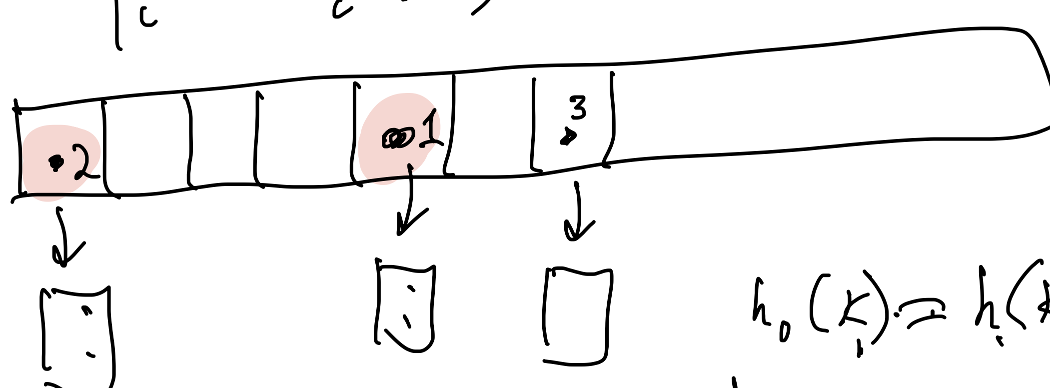
advice: $s \geq 3 \cdot (\max \# \text{ of entries})$

to search:

for each $p_i = (h(k)+i) \bmod s$ {
  if $A[p_i]$ is empty, return failure.
  if $A[p_i] \to key == k$, return success ($A[p_i] \to data$)
}
// very bad case: table is full
return failure.

method: family of hash functions $h_0(), h_1() \ldots$

$$p_i = h_i(k)$$

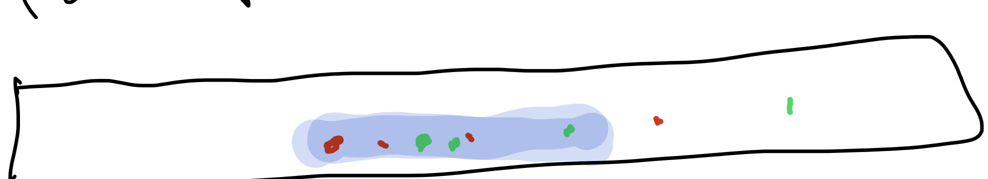


$h_0(k_1) = h_i(k_2) =$

$h_1 \quad \neq \quad \neq$

method: quadratic probing

$$p_i = (h(k) + i^2) \bmod s$$



still clustering, secondary

method: add-the-hash rehash

$$p_i = \left(h(k)\cdot(i+1)\right) \underline{\bmod} \; s$$

don't use 0 index of array.

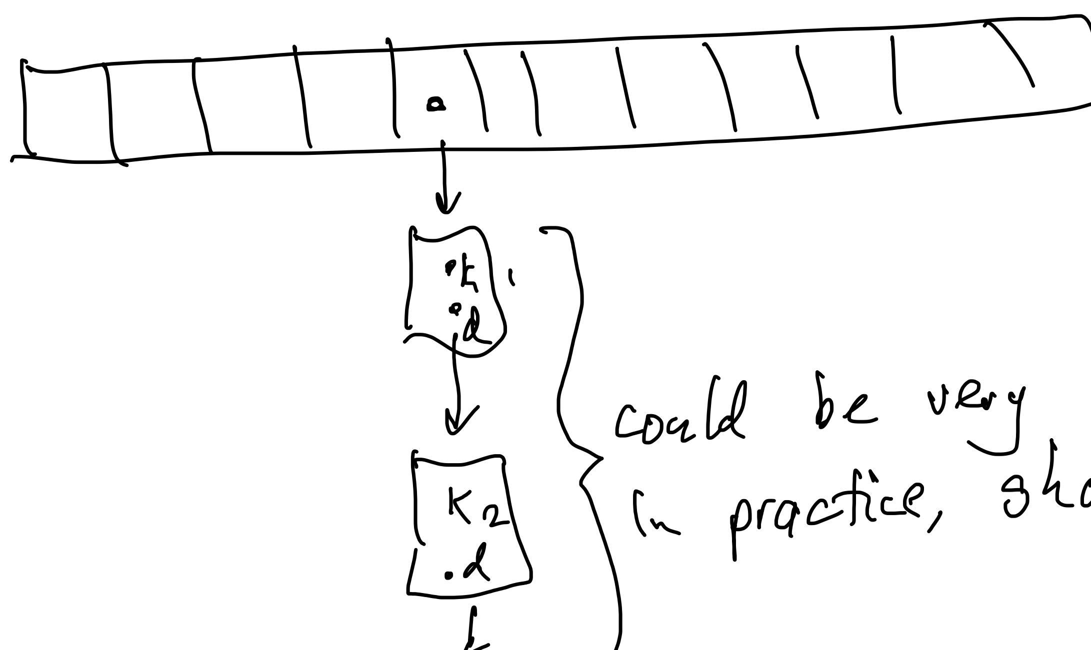
s should be prime so that p_i eventually try all cells.

method: double hashing

$$p_i = \left(h_1(k) + i\, h_2(k)\right) \underline{\bmod}\; s$$

$h_2(k)$ → never 0

Alternative to open addressing: external chaining



could be very long.

In practice, short.

$s \cong$ # of elements to insert

on average, each chain has length 1.

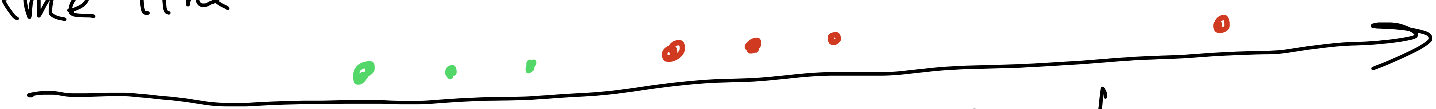
cost of insertion: $\Theta(1)$ (average)

to insert k: (without duplication)

let $i = h(k)$
for each node d in list headed by $A[h(k)]$
  verify $d.\text{key} \neq k$.

Build new node ⓐ = {k, data}
Insert ⓐ in list headed by A[h(k)]
at beginning, especially if search/insert
show "locality of reference"

time line



advice: insert at start of chain
when searching, move success node
to start of its chain

to search for k:
Let i = h(k)
for each node d in list headed by A[i] {
if d.key == k, return sucess
}
return failure.

Alternative to representing chain as a list
binary tree, 2-d tree, ...
better: use larger s.

what is a good h() ?

- fast.
- examine all of K. (up to a reasonable limit)
- uniform: 0 ... s-1 equally likely.

? spreading (not important for external chaining)
Similar key should hash to very different values.

In practice: key is a sequence of bytes. b_i

method: $\left(\sum_i b_i\right) \underline{\text{mod}}\ s$

fast if $s = 2^j$ for some j
mask the lower j bits
eg: $j = 4$ mask (binary) with 0000 1111

$j = \lceil \log_2 n \rceil$

expected capacity

method:
for b_i do:
value = value * 237 + b_i;
answer = value $\underline{\text{mod}}$ s;

method: $\left(\sum_i b_i \ll i\right) \underline{\text{mod}}\ s$.

method: $\left(\text{XOR}_i\ b_i \ll i\right) \underline{\text{mod}}\ s$

wisdom: not important what h( ) you use, as long as it's not silly.

How big should s (array size) be?

prime if using quadratic probing, ... (open addressing)

2^j so that $\underline{\text{mod}}$ is fast

if expect n elements, set $s \approx n$. (external chaining)

what if s turns out to be too small?

1) Live with it.
2) Rehash all elements into a bigger table (pause in computation)
3) extendible hashing: split chains
binary tree, trie (split on last bit)

Modern programming languages have hash tables built in.
(associative arrays)

Perl: $a\{k\} = data

JavaScript: a[1] a['string']
index key

Java: library HashMap

you don't specify S.

Python: Foo = dict()
Foo['this'] = 'that'
key data

Cryptographic hashes (digests)

h(text) = number (represented as a string of hex digits)

purpose: uniquely identify text.

goals:

fast computation

uninvertable

given h(k), ~~impossible~~ infeasible to derive k.

collision-proof:

infeasible to generate a collision

Examples: MD5 $h(x)$ is 128 bits.
Practical attack in 2008

SHA-1 160 bits
Generating a collision in 2^{69} operations. (2005)

SHA-2
variants: SHA256 (256 bits)
SHA512 (512 bits)

Uses:

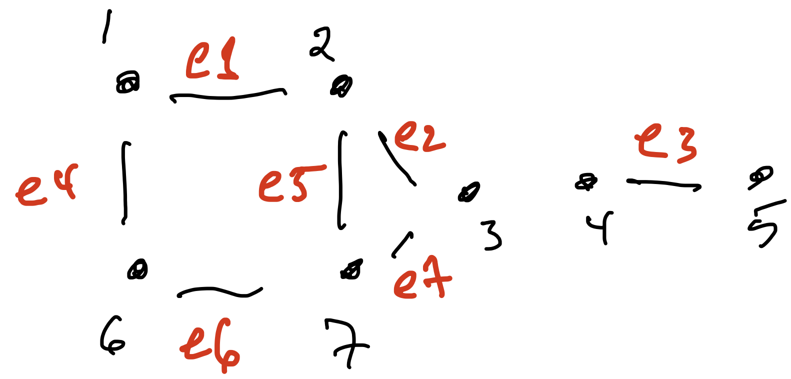
① Storing passwords. : store user name, digest (password)
/etc/passwd

② catching copying : verify that excerpts of submissions are unique.

③ authentication : want to prove that I sent message m.
shared secret S between sender (A) receiver (B)
$h(m + s)$

④ intrusion detection: store hash of every essential program in a protected file.
Compare h(file content) with stored value.
(tripwire) (the digest of a program is called its signature)

Graphs



vertex (vertices)
edge

graph properties: connected? (no)

directed? (no)

edges have a direction ⟶

weighted? (no)

edges have a numeric value (weight)

sparse? (somewhat) / dense (not very)

vertex properties: fanout / fanin   degree

$\underbrace{\text{fanout / fanin}}_{\text{directed}}$ $\underbrace{\text{degree}}_{\text{undirected.}}$

vertex 2
has degree 3.

graphs represent

1) streets in a city (vertices are corners, edges are streets)

want to compute paths. (sequence of edges)

2) airline routes.

weight of an edge: cost of ticket

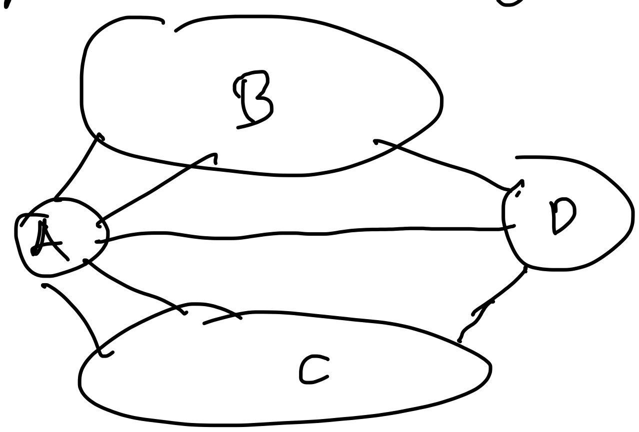
minimal-cost cycles

Hamiltonian cycle: visit every vertex once.

Eulerian cyle: visit every edge once.

1707-1783

3) Bridges of Königsburg (later Kaliningrad)

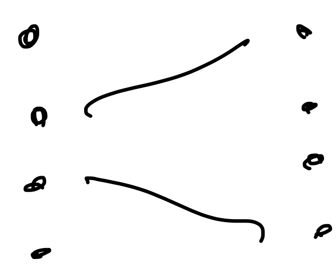


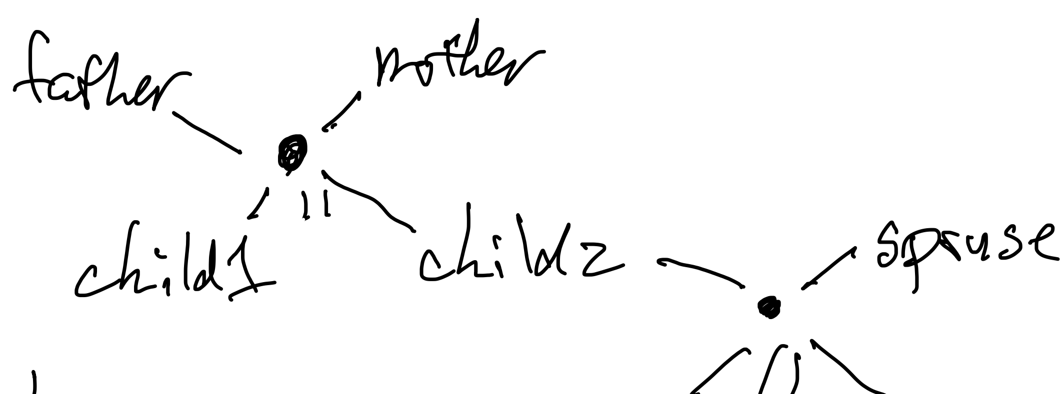
Eulerian cycle exists iff # vertices with odd degree 0 or 2, (?)

4) Family trees.

Bipartite graph: 2 kinds of vertices
Edges only connect vertices of different kinds.

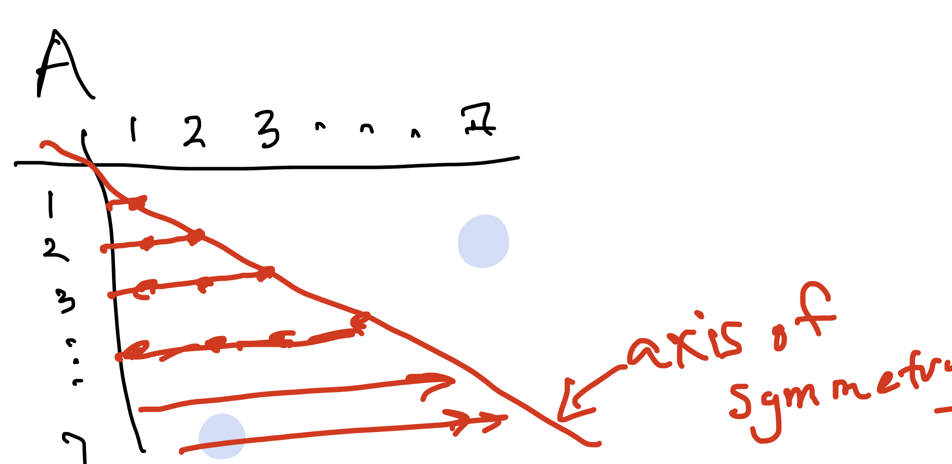
Two kinds: people
families





To represent a graph.

- Adjacency matrix



$A[i,j]$ = true iff $\exists$ edge connecting vertex i to vertex j.
use a 1-dimensional representation!

$A[i,j]$ stored at $A(\underline{i(i-1)/2 + j})$

1 dimensional.

$A[3,7]$ $\quad A\left[\frac{3\cdot 2}{2} + 7\right] = A[10]$

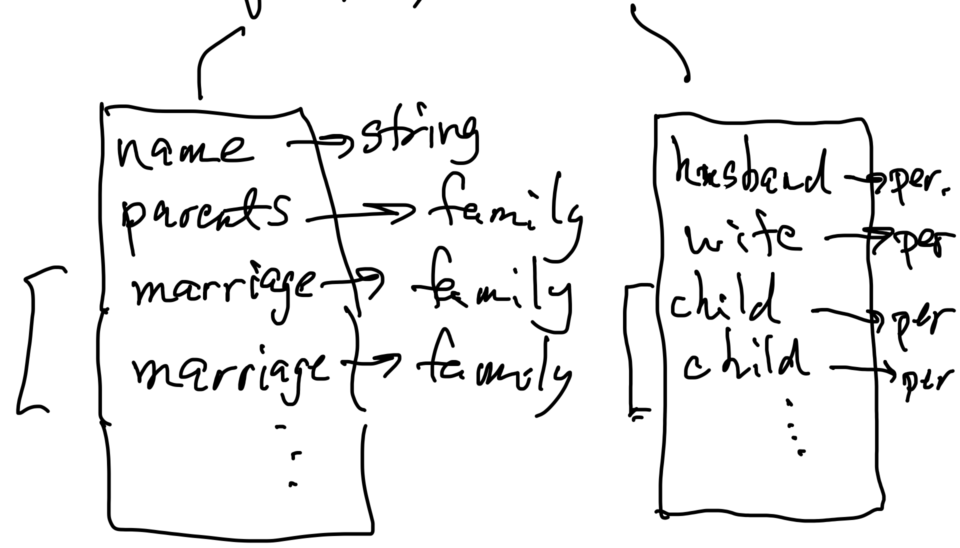
- Adjacency list

```
1 | → 2 → 6
2 | → 1 → 3 → 7
3 |   2   7
4 |   5
5 |   4
6 |   1   7
7 |   2   3   5
```

vertex $\qquad$ neighbors

- Specialty representations

family tree: structs for people, families



Compute the degree of all vertices

matrix $\quad$ <u>foreach</u> vertex (0 .. v-1) { $\quad$ number of vertices

degree [vertex] = 0;

<u>foreach</u> neighbor (0 .. v-1)

$\Theta(v^2)$

<u>if</u> A[vertex, neighbor]

degree [vertex] += 1;

}

Adjacency list

```
foreach vertex (0 .. v-1) {
    degree[vertex] = 0;
    for (neighbor = L[vertex];
         neighbor ≠ null;
         neighbor = neighbor -> next)
      degree[vertex] += 1
}
```

$\Theta(v+e)$

Connected component of vertex i

Algorithm: Depth-first search (DFS)

```
void DFS(vertex i)
    // assume visited[*] = false at start
    foreach neighbor(i) {
       visited[neighbor] = true
       if !visited[neighbor]
            DFS(neighbor);
    }
```