Midterm:

min: 15.5

max: 97.5

avg: 65.2

Q1.

Copy constr: creates new object

Cpy asoign.op : operator=(..)

— Gives to existing object

Q2

```
p3 = p1 < p2 ? p1 : p2;

if(p1.operator<(p2))
    p3.operator=(p1);
else
    p3.operator=(p2);
```

Q3

const_iterator } accessor

↑ derived from

iterator } accessor + mutator

derived class expands functionality of parent class

Q4 OOP

a) info hiding:

restrict access to (private) object components (data)

b) encapsulation:

grouping of data & methods in one aggregate (i.e. class)

c) template: provide
type-independence
(generic programming)

d) inheritance:
class extension (or
restriction) of a class
(by deriving a class)

e) polymorphism:
(e.g., overloading)

Q6

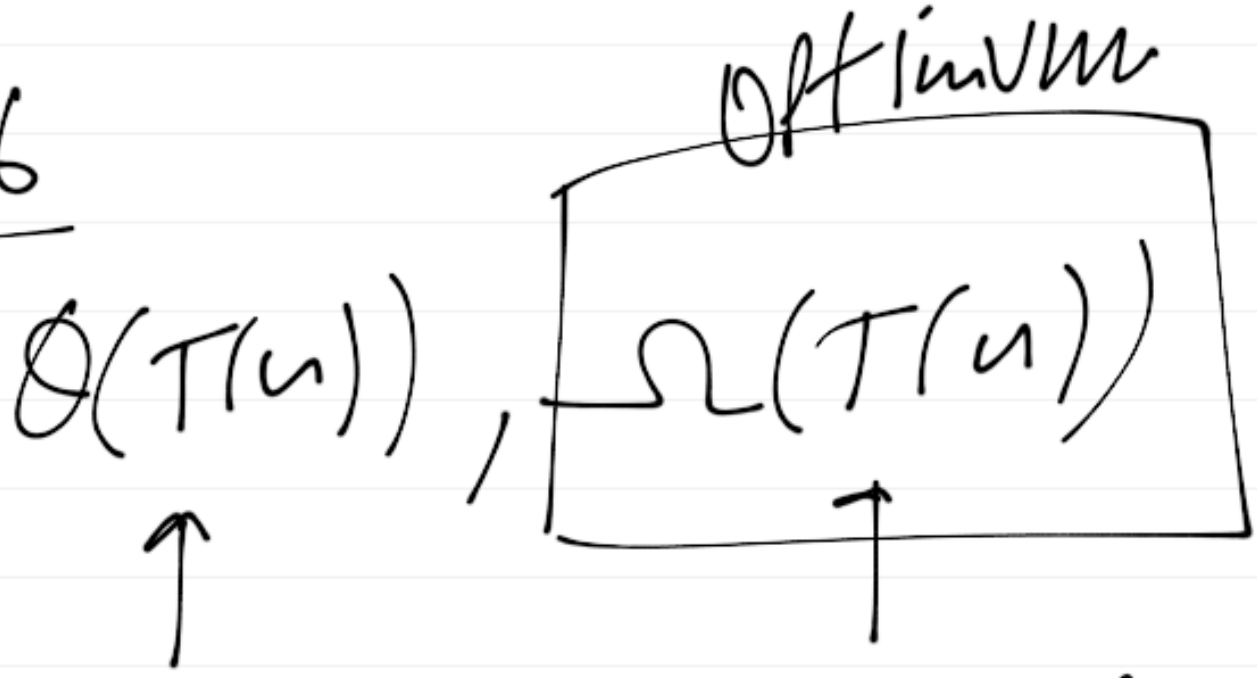
$T(n) = \Omega(G(n)) \mid \exists c, n_0 :$

$T(n) \geq c\, G(n), \forall n \geq n_0$

"big-omega": lower bound

(best case)

Q6



average running time

heuristic minimum running time

if $(\Theta(T(n)) = \Omega(T(n))$

alg. runs optimally on average

if $(\Theta(T(n)) = \Omega(T(n))$

alg. is optimal

Q7

Alg. 1: $\Theta(n) < \Omega(n \lg n)$

on avg. better than minimum

→ impossible; no sample

Alg. 2: $\Theta(n^2) \cdot O(n^2) > \Omega$

worst case = avg. case

e.g. bubble sort

Ay.3 $\Omega = \Theta = O$

optimal alg.

e.g. AVL, R-B, balanced tree alg.

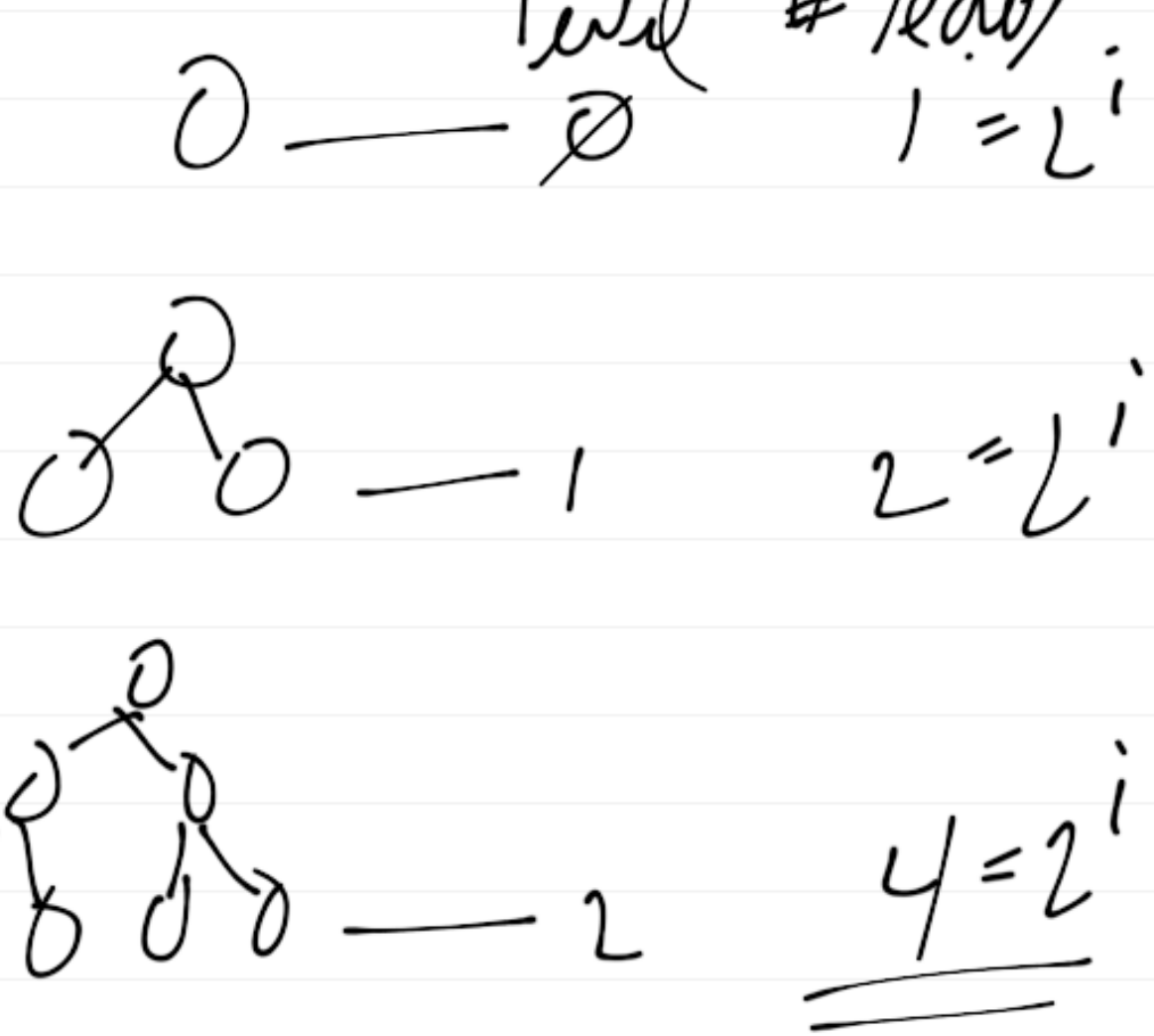
Ay.4 $\Omega = \Theta = n \lg n$

$O(n^2)$

dependents in worst case

e.g., binary search tree

Def



2^i nodes per level

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

leaves: 2^h

Q9 Heap insert: $O(\lg n)$

St. A | St. B | slightly better

$O(\lg n) \times n$

$= O(n \lg n)$

for Insert

+

$O(\lg n) \times n$

$= O(n \lg n)$

for DeleteMin

$O(2n \lg n) \in O(n \lg n)$

$O(1) \times n$

$= O(n)$

for Toss

+

$O(n) \times 1 = O(n)$

FixHeap

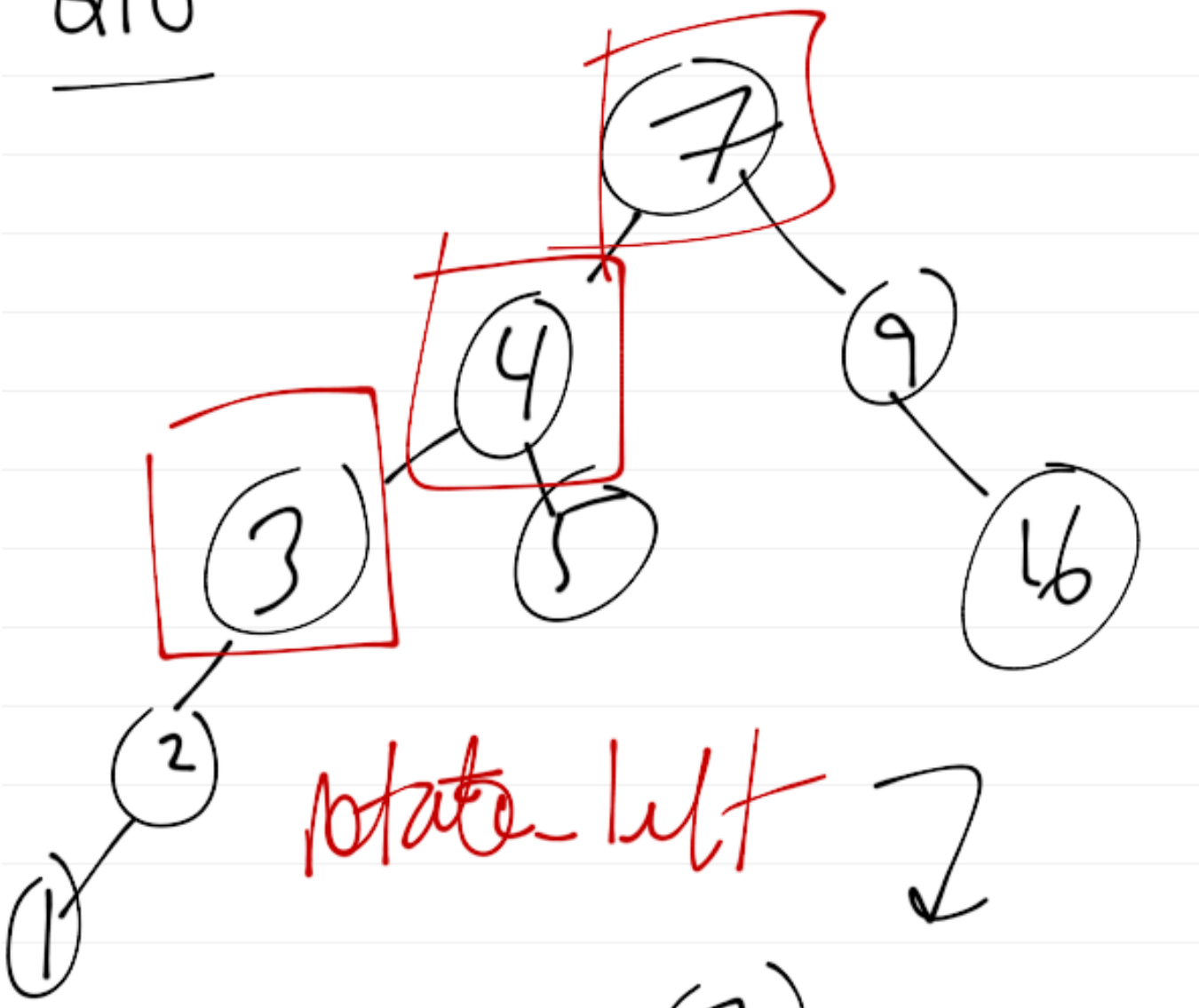
+

$O(n \lg n)$ DeleteMin

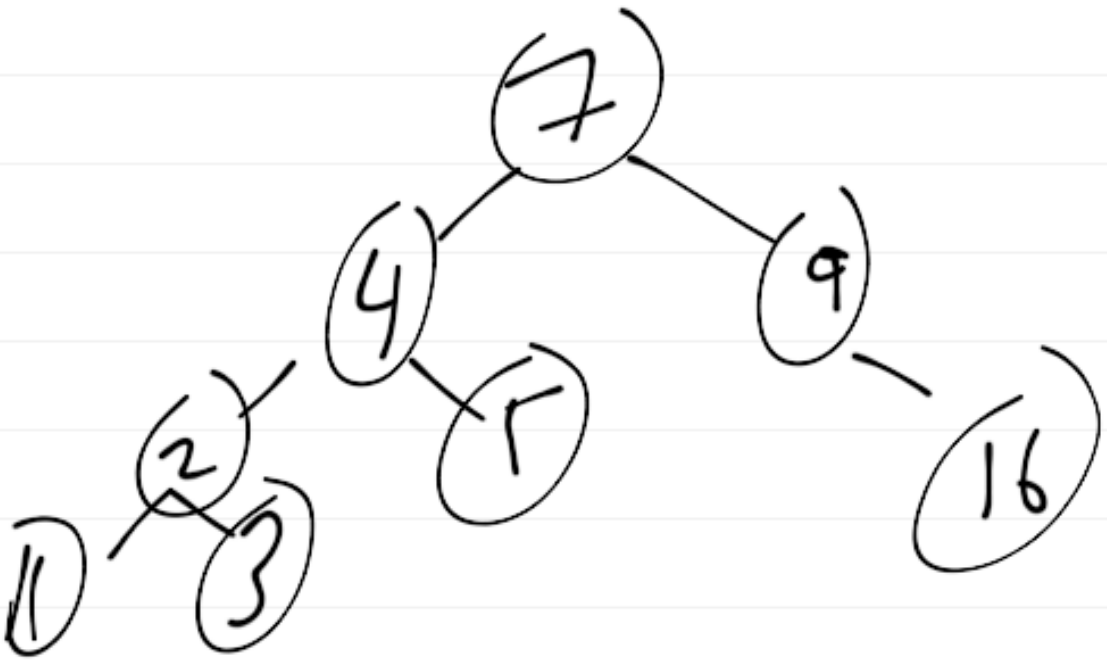
$O(n \lg n + 2n)$

$\in O(n \lg n)$

Q10



rotate left



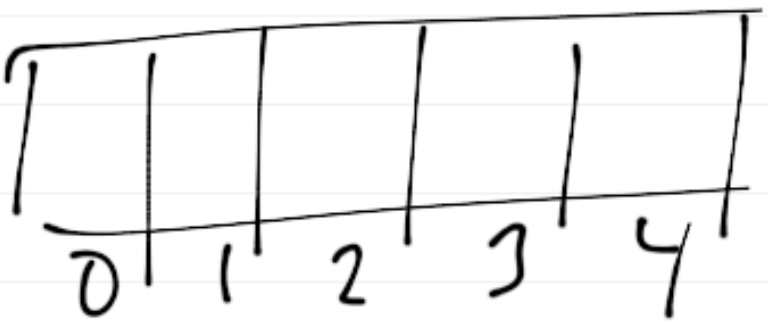
Hashing (Chp. 5)

- data structure is a hash table
- basic array that supports $\Theta(1)$ insertion, deletion, find (avg. constant time)

- given key (data element, e.g. a number, a string)

Compute its index into the array via hash function

E.g. a very small hash table:



hash table of size 5 ($n=5$)

given any number, e.g.,

19203, 32762, etc.,

simple hash function is

number $\boxed{\%}$ size

mod

any number % 5
will give index $\in [0, 4]$

— two major concerns:

1. if no. elements $>> n$,
   we'll get collisions

how to store:

2. performance of hash function, e.g., given {25, let's say all numbers are multiple of 5, 5, 10, 15, 20, ...

⇒ all go in index [0]

(just shared table)

- a good candidate for hash table size & hash function is the prime numbers

1, 3, 6, 7, 11, 13, 17,

19, 23, 24, 31, 37,

41, 43, 47, 53, 59,

... 10,007

- text gives good hash function for strings

Keys

$$h = \sum_{i=0}^{keysize-1} Key[keysize-i-1] * 37^i$$

via Horner's rule:

$h_k = K_0 + 37K_1 + 37^2 K_2 =$

$((K_2)37 + K_1)37 + K_0$

e.g.

```
h = 0                                   (++)
for(int i=0; i<key.length(); i
    h = 37 * h + key[i];
h = h % tablesize
if(h<0) h += tablesize
```

- implementing the hash table (doesn't seem to exist in C++ STL)

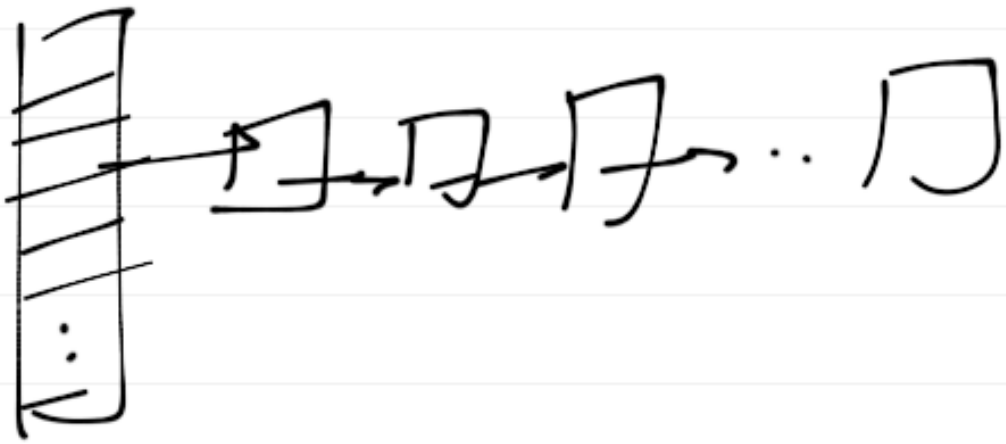
```
class hash_t
{
    .
    .
    .
private:
std::vector< std::list<T> >
        elements;
};
```

- Analysis is tricky:
  - need to consider average case $O()$

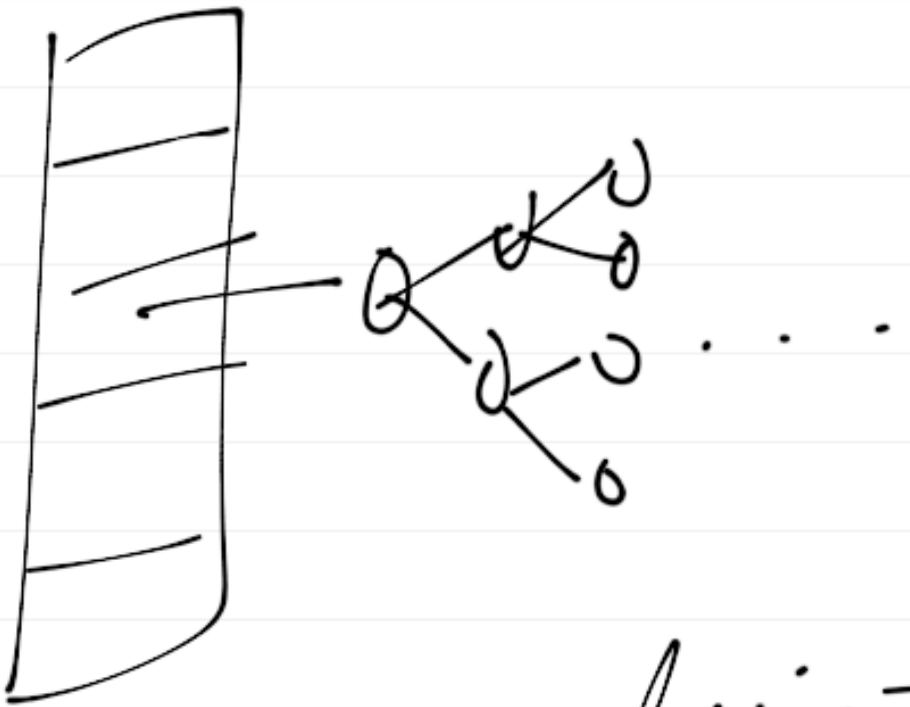
e.g.

$O(1)$ + $O\left(\frac{n}{size}\right)$

hash fn + search



- mapa do Heller:
  - hash table of
    hash tables
  - hash table of trees



- average-case analysis ≡
  amortized analysis