e-mail list

Dr.	Raphael	Finkel
Mr.	Ray	Goldstein
Prof.	Rafi	
~	רפאל	

raphael@cs.uky.edu

multilab:

cor
pen } .cs.uky.edu

ssh	VPN
putty	
⋮	

Basic building blocks

Data structures

way to represent information
so it can be manipulated
packaged with code to manipulate

ADT: Abstract Data Type

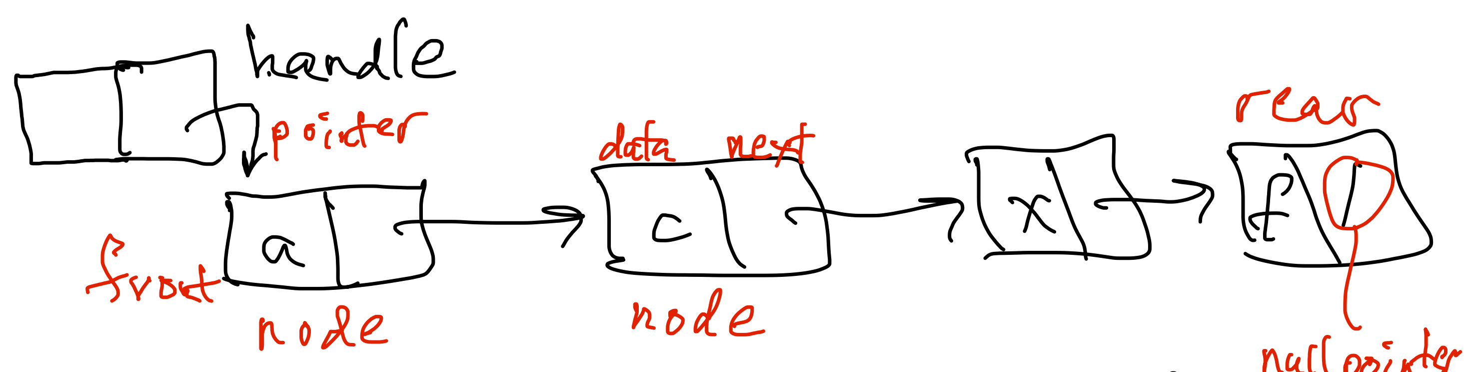
Tool:

$$\frac{\text{Use}}{\text{Implementation}}\ \text{Specification}$$

Software tool

$$\frac{\text{program (application)}}{\text{program: ADT}}\ \text{API (application programming interface)}$$

Singly-linked lists



operation	cost (complexity)
create empty list	$\Theta(1)$
delete list	
	big-Θ notation
	order of 1
insert new node at front	$\Theta(1)$
delete first node (return data)	$\Theta(1)$
count length	$\Theta(n)$
search for data	$\Theta(n)$
sort list	$\Theta(n \log n)$ . . .
	$\Theta(n^2)$

To speed up counting, keep a current count in the header.

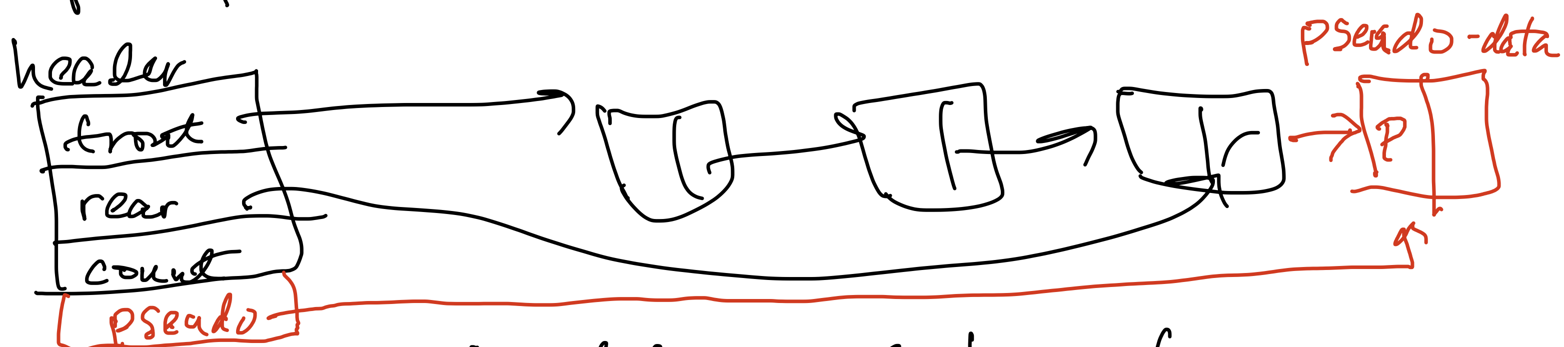
update count on every insert, delete.

counting length is now $O(1)$.

useful if you need the count often.

what if I want to insert at the rear?

Keep a pointer to the rear in header.

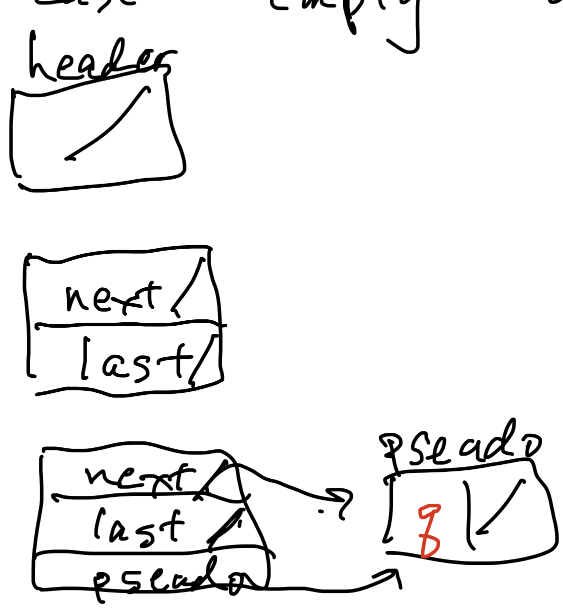


pseudo-data: extra data used to make searching faster.

Optimization (time)

1) If it's fast enough, leave it alone.
2) Maybe: wait a year.
3) Find out where time is spent; concentrate on that. (valgrind)
   a) better algorithm or data structure.
   b) local optimization (pseudo-data)

boundary cases: empty list

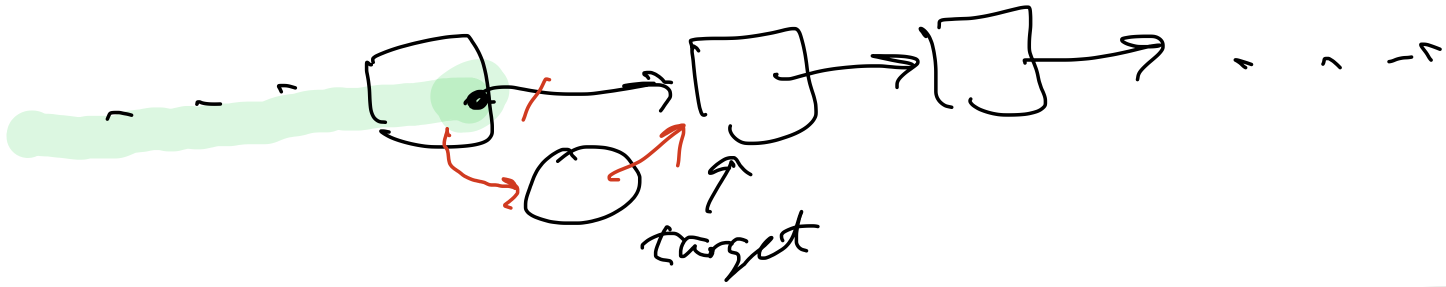


insert new node after a given node.

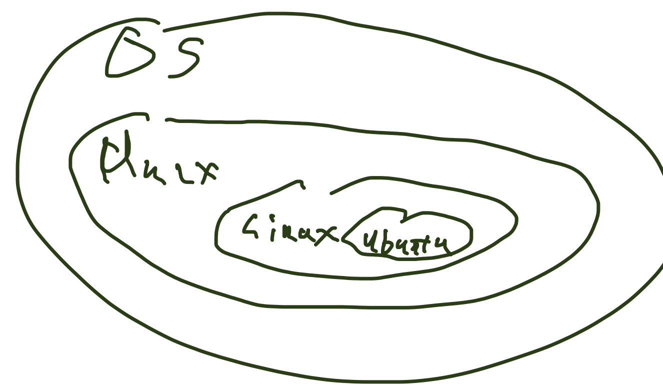
generate new node
put data in node
copy target→next to new→next

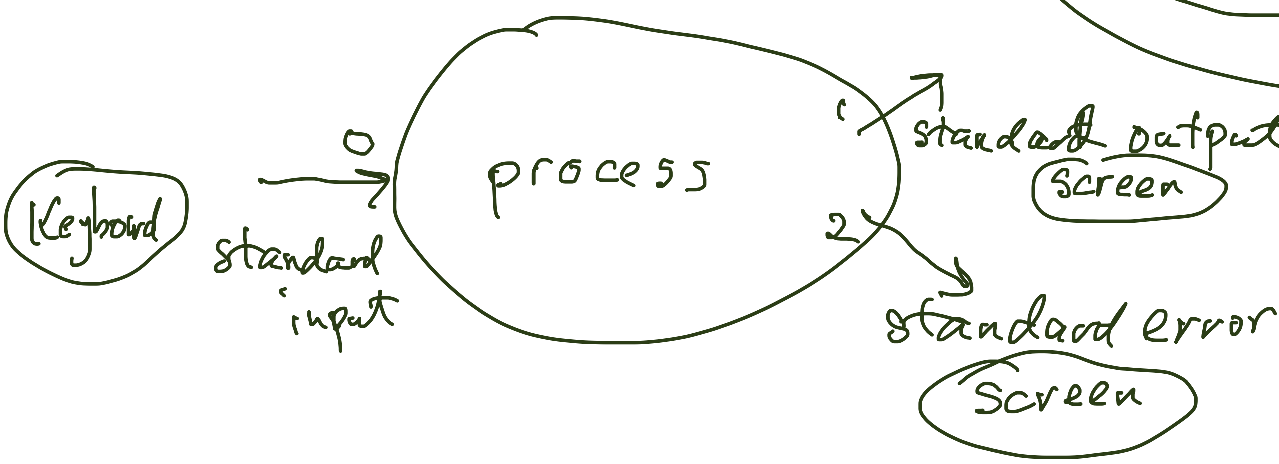
insert a new node
before a given node

replace target->next
with new.



Aside: Unix pipes

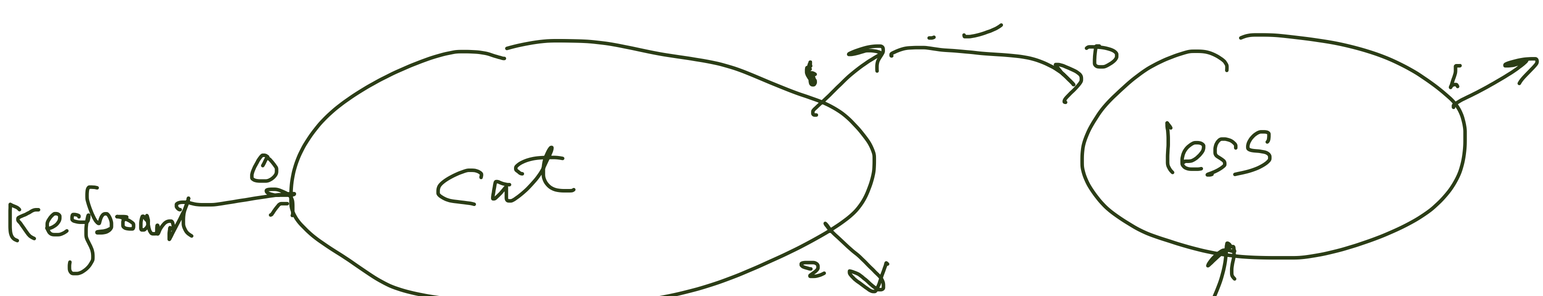




% cat (uses stdin, copies to stdout)

% cat fileName (open file, copy to stdout)

% cat fileName | less



% cat fileName | wc

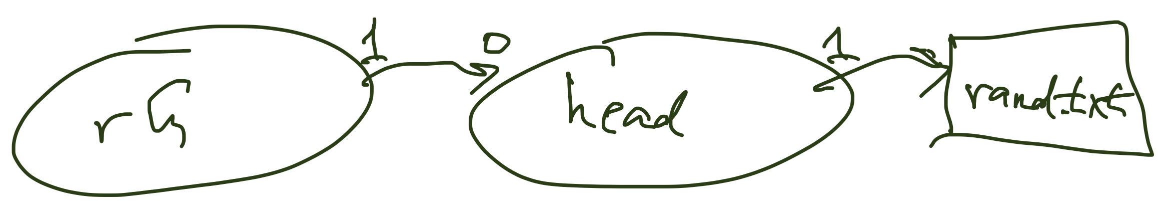
% ls | less

% ls | sort

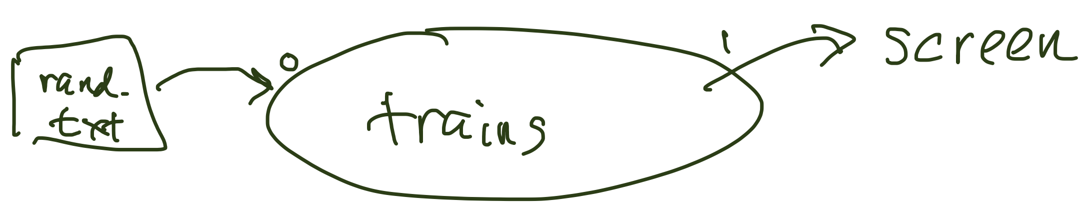
"creeping featurism"

```
% trains    (wait for random-number inputs)
% randGen.pl | trains
% randGen.pl     (don't do this)
% randGen.pl | trains | less
```


```
% randGen.pl | head -1000 > rand.txt
```




```
% trains < rand.txt
```



Makefile : recipe file

```
target : prereq.
    step
    step
```

Stacks, Queues, Dequeues.

Stack of integer

operations (API)

```
stack *makeEmptyStack( )
boolean isEmptyStack(stack * S)
int popStack(stack * S)
void pushStack(stack *S, int I)
```

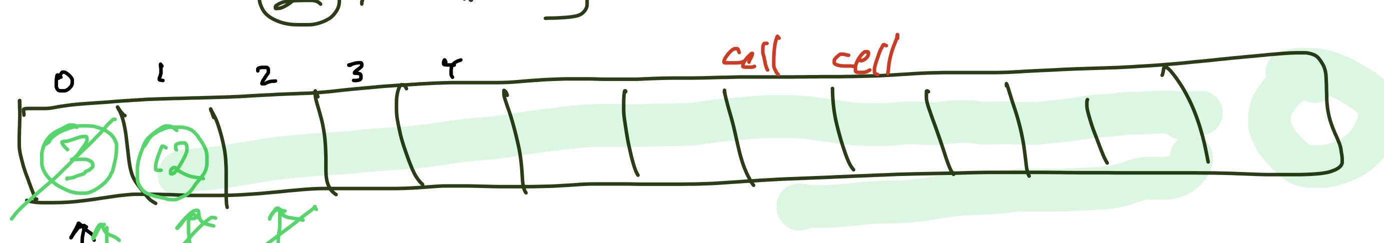
```
stack * myStack = makeEmptyStack();
pushStack(myStack, 3);
pushStack(myStack, 12);
print popStack(myStack); // 12
print popStack(myStack); // 3
print popStack(myStack); //error
```

---

implementation ① linked list (front of list = top of stack)

- makeEmptyStack: makeEmptyList
- isEmptyStack : isEmptyList
- pushStack : insertAtFront
- popStack : deleteFromFront

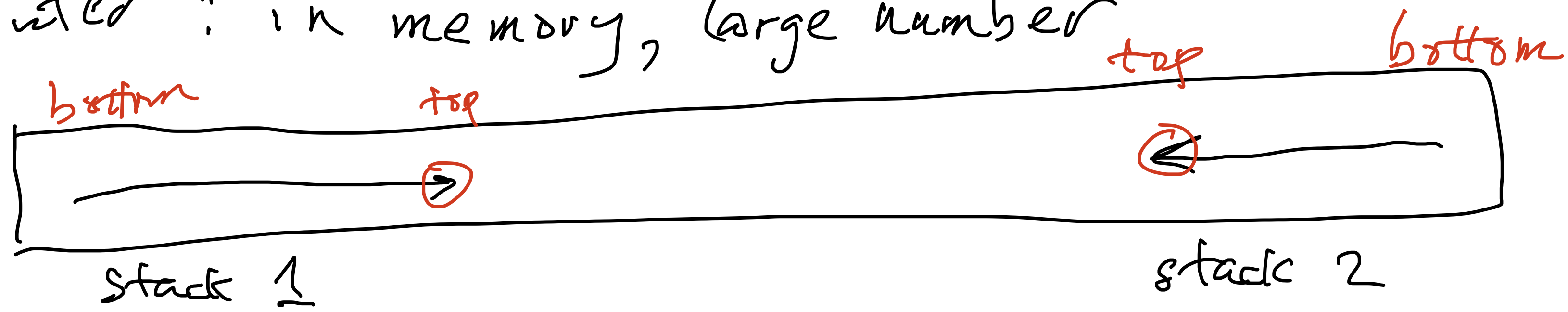
implementation ②: Array



count (index to insert at)

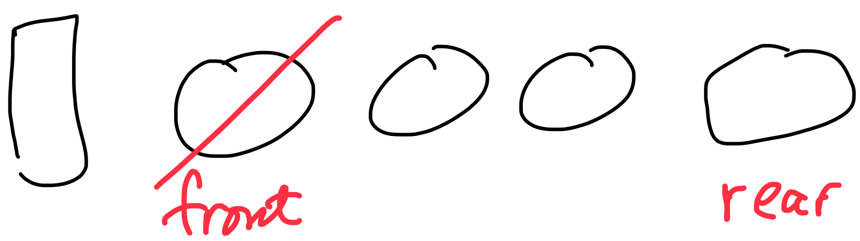
index : in an array, small number

pointer : in memory, large number

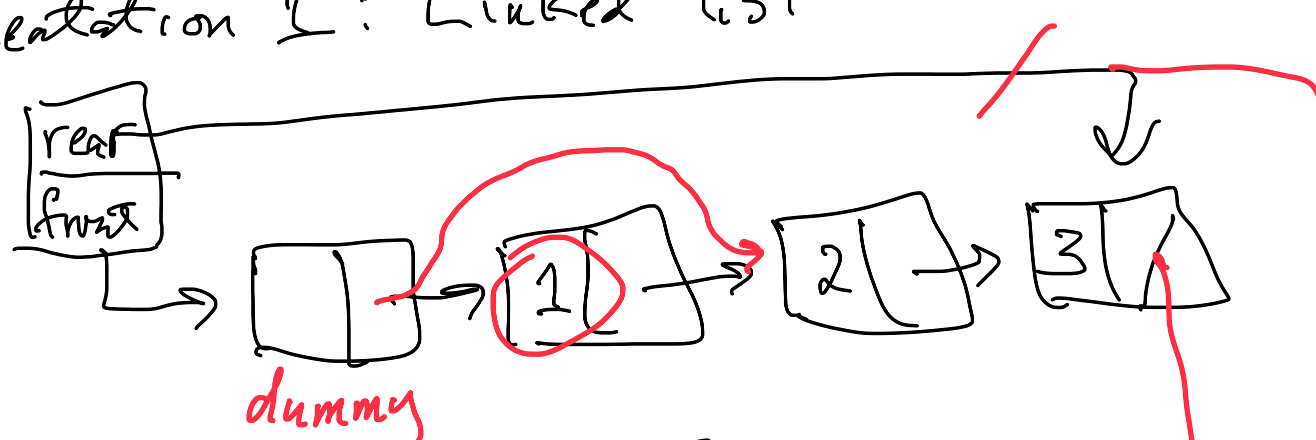


Queue of integers

Empty or result of insert either at rear or deleting from the front.



Implementation 1: Linked list

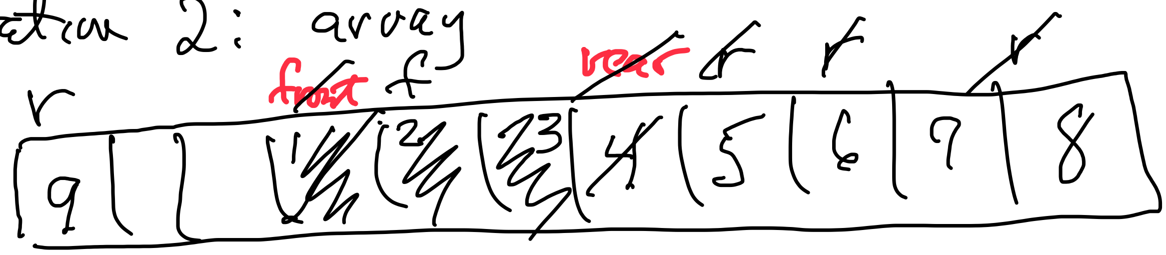


isEmptyQueue

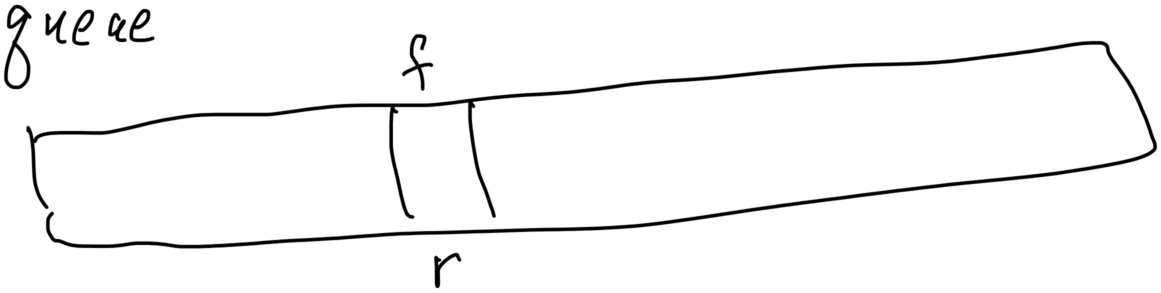
insert $\Theta(1)$

delete $\Theta(1)$

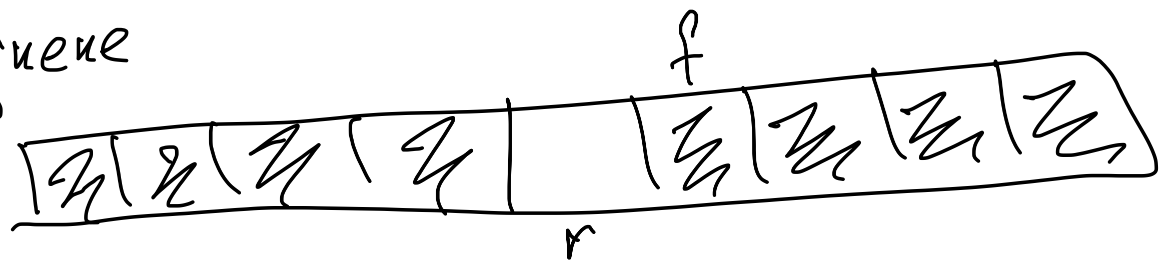
Implementation 2: array



empty queue



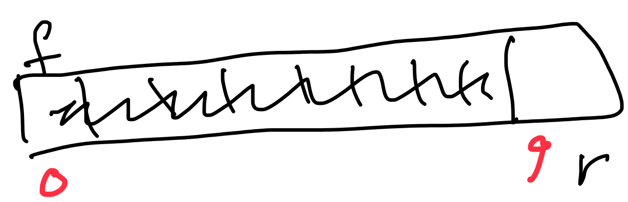
full queue



isEmpty: $f == r$?

isFull: $r+1 == f$

$\text{nextCell}(r) == f$



$\text{nextCell}(\text{index}) = (\text{index}+1) \% \text{size}$

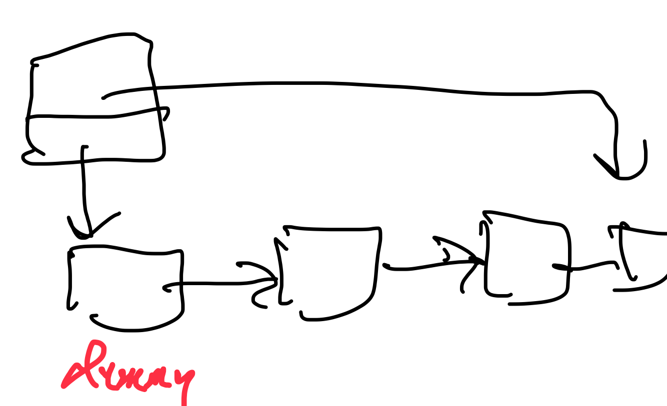
Dequeue /dɛk/ /dikju/

Either empty or the result of inserting at front or rear, or deleting from front or rear

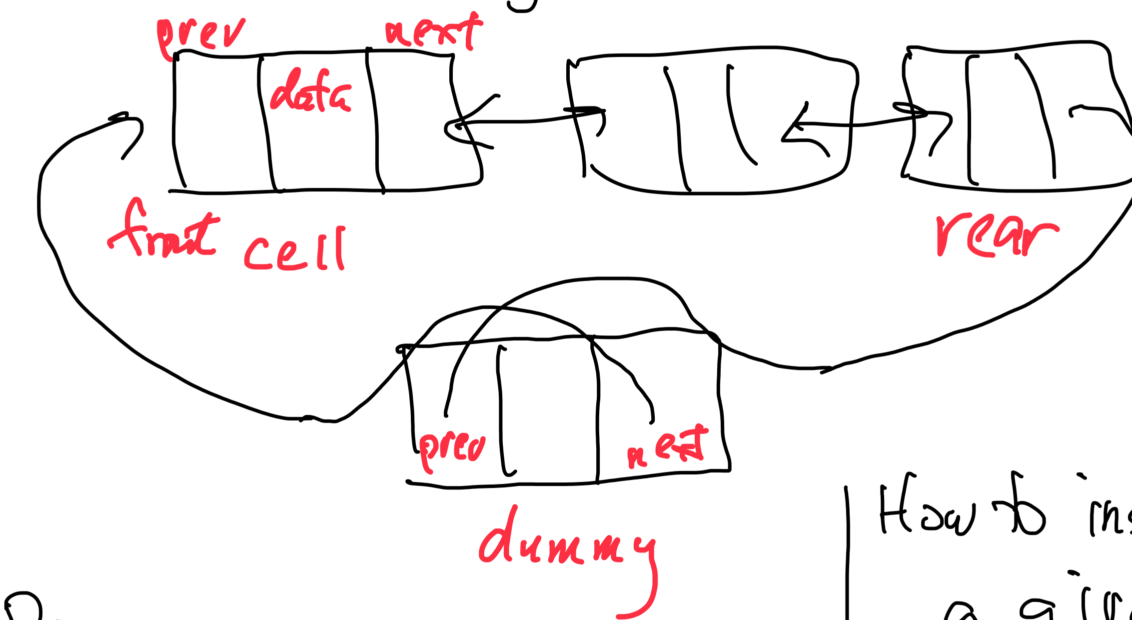
Operations

- $\Theta(1)$ makeEmptyDequeue
- $\Theta(1)$ bool isEmptyDequeue
- $\Theta(1)$ insertFrontDequeue
- $\Theta(1)$ insertRearDequeue
- $\Theta(1)$ deleteFrontDequeue
- $\Theta(n)$ deleteRearDequeue

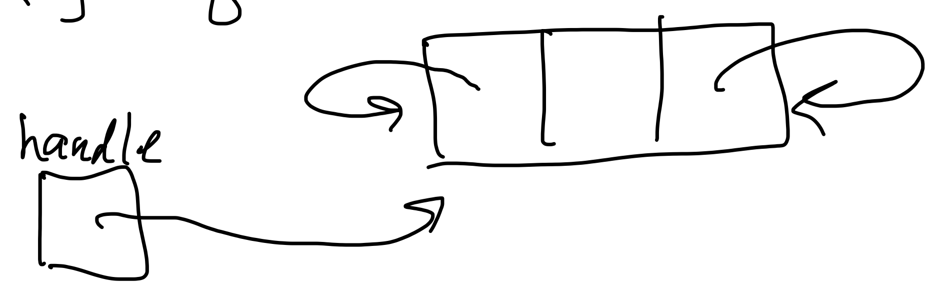
linked list



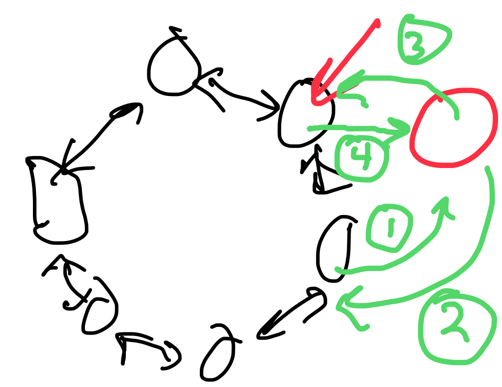
Implementation: Doubly-linked list



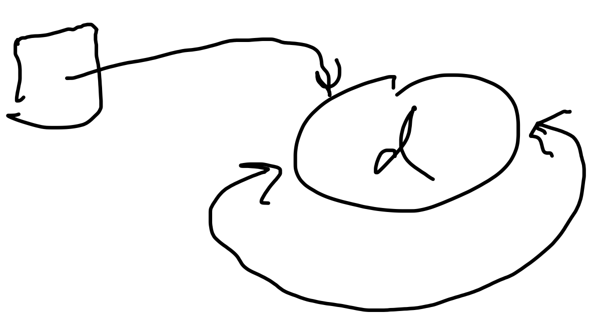
Empty Deque

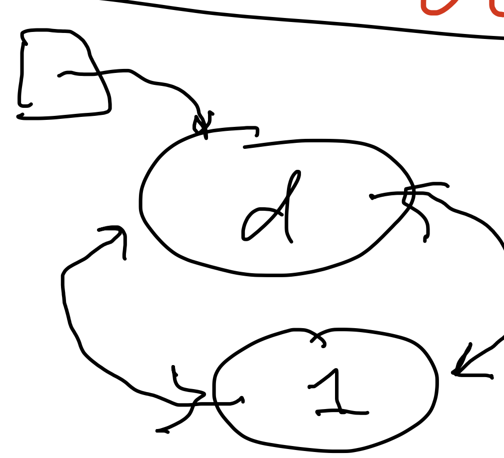


How to insert after a given node $\Theta(1)$



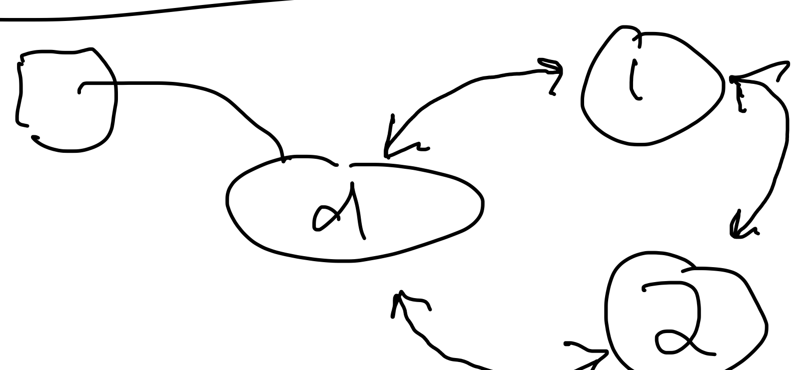
How to insert before a given node. $\Theta(1)$





clockwise: next

counter-clockwise: pred



permission violation

randGen.pl

% chmod +x randGen.pl

./workingTrains

Searching

n data elements

operations

void insert(int data, *D)

bool search(int data, *D)

more generally:

node *search(key_type key, *D)

struct — name

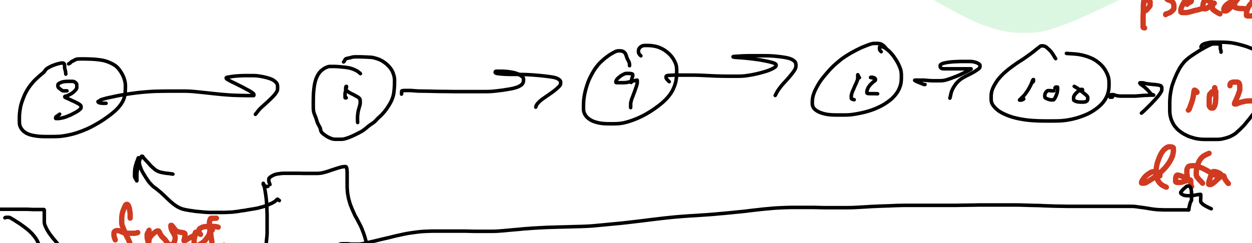
name
address
phone

Representation 1: Linked list

insert: put new element at front $\boxed{\Theta(1)}$

search:  $\boxed{\Theta(n)}$

Representation 2: Sorted linked list

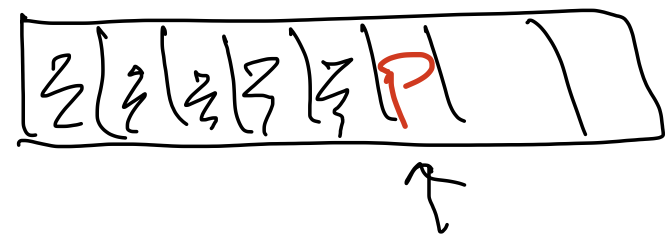
insert: 

$\boxed{\Theta(n)}$

Search: walk down list until too far
pseudo data at end with big value.

$\boxed{\Theta(n)}$

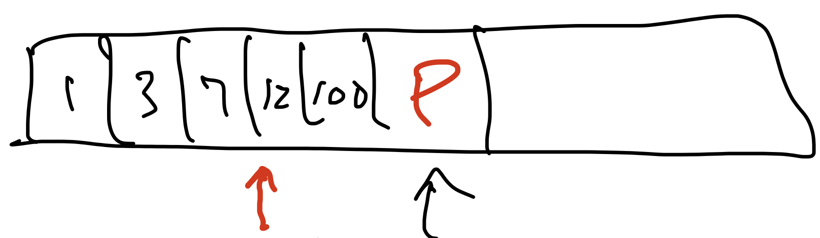
Representation 3: array



$\boxed{\Theta(1)}$ insert: place at end

$\boxed{\Theta(n)}$ Search: walk until found or not present

Representation 4: sorted array



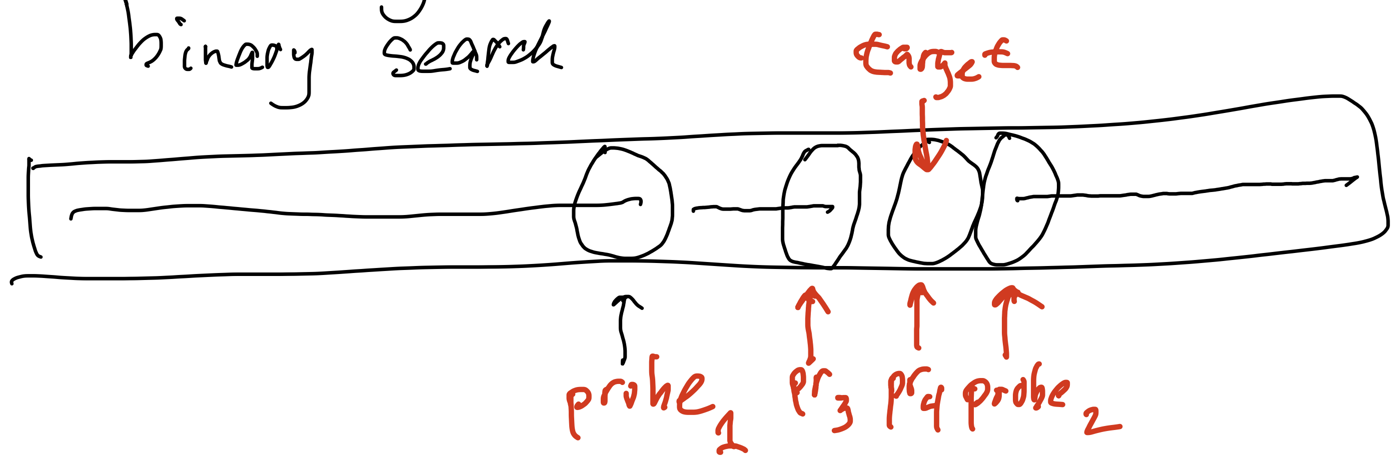
insert:

find right place $\Theta(n)$

$\boxed{\Theta(n)}$ shove remaining elements over $\Theta(n)$

search: $\boxed{\Theta(\log n)}$

binary search



Quadratic search

$\Theta(\log \log n)$

→ $\Theta(\log n)$ for both insert and search.

Binary search analysis.

Searching $C_n = 1 + C_{n/2}$ recurrence formula

Recursion theorem

$$C_n = \underset{\substack{\uparrow \\ 1}}{f(n)} + \underset{\substack{\uparrow \\ 1}}{a}\, C_{n/b} \leftarrow 2$$

where $f(n) = \Theta(n^k)$ ← $k=0$

↑ theta

when	C_n
$a < b^k$	$\Theta(n^k)$
$a = b^k$	$\Theta(n^k \log n)$
$a > b^k$	$\Theta(n^{\log_b a})$

$a = 1$
$b = 2$
$k = 0$
$b^k = 1$

$$C_n = \Theta(n^k \log n) = \Theta(n^0 \log n)$$
$$= \Theta(\log n)$$

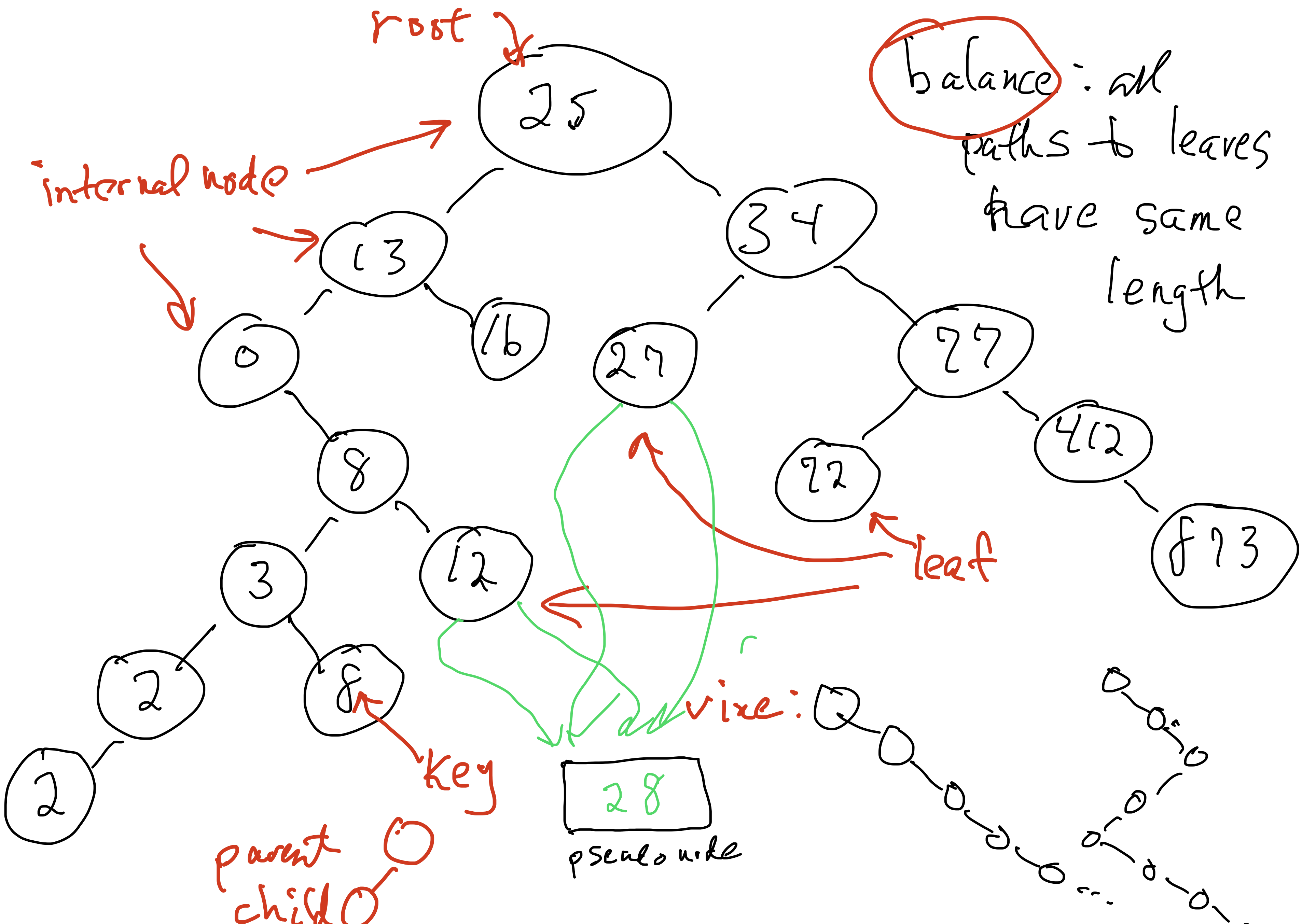
notation

$O(f(n))$ no worse than $f(n)$ = at most $f(n)$

theta $\Theta(f(n))$ no better or worse than $f(n)$ = exactly $f(n)$

omega $\Omega(f(n))$ no better than $f(n)$ = at least $f(n)$

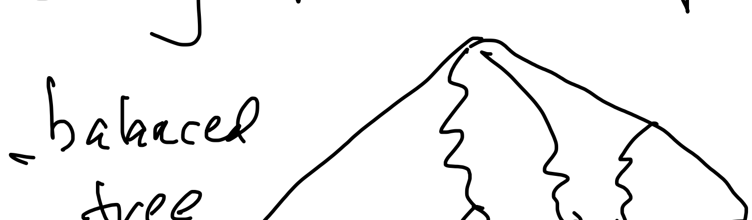
Representation 5: Binary tree

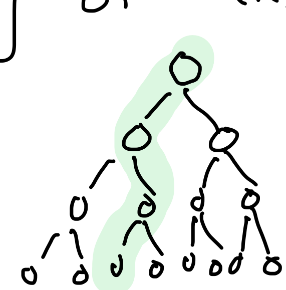


traversals: inorder (symmetric order): parent between children
preorder: parent before children
postorder: parent after children

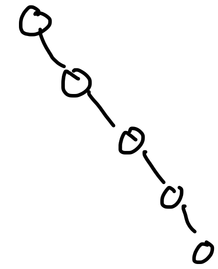
Representation 6: Hashing : later

Binary trees complexity of insert? $O(\log n)$ if balanced





n	depth
1	1
3	2
7	3
15	4
2^d-1	d
n	$\log(n+1)$



n	depth
1	1
2	2
3	3
n	n

$\Theta(n)$

build a binary tree using random keys,

longest depth is about $2 \cdot \log(n)$

$\Longrightarrow$ insert $\Theta(\log n)$

likewise for searching.

Finding the largest element in a set.

```
largest = -∞
foreach element in set {
    if (element.value > largest)
        largest = element.value;
}
return largest;
```

$\Theta(n)$

Finding the 2nd-largest element in a set

```
largest = -∞; nl = -∞; // next largest
foreach (element in set) {
    if (element.value > largest) {
        nl = largest;
        largest = element.value;
    } else if (element.value > nl) {
        nl = element.value;
    }
}
```

}

return nl;

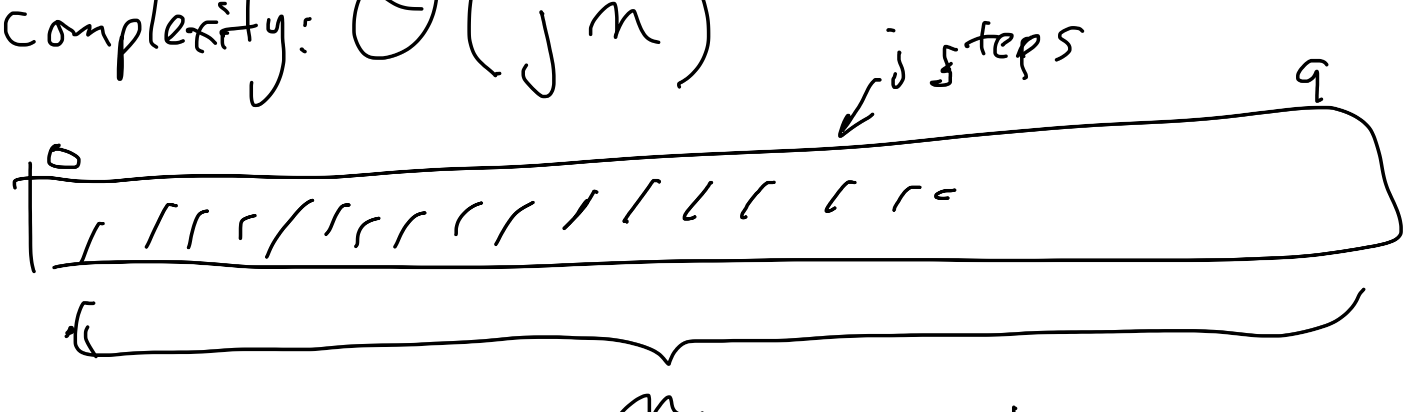
Complexity: $\Theta(n)$

---

finding the j th largest element in a set.

work inside the loop is $\sim j+1$

complexity: $\Theta(jn)$



median of n elements is the j th largest,

where $j = \lfloor n/2 \rfloor$ floor

$n = 10 \quad j = 5$

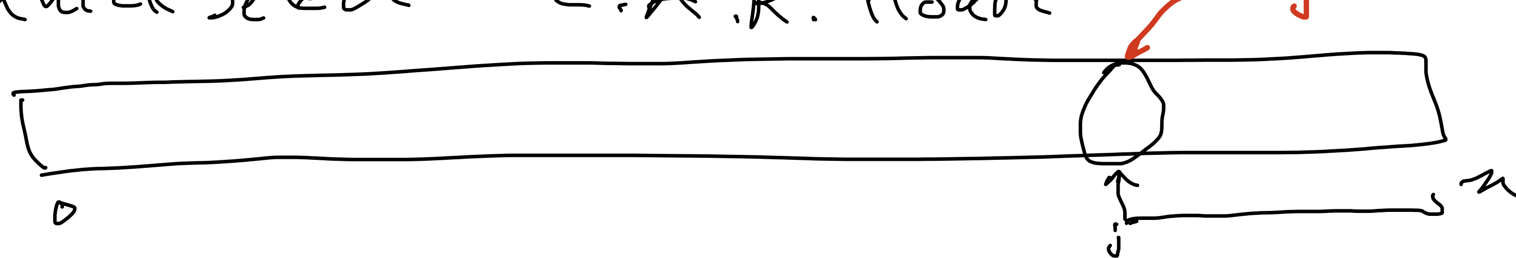
$n = 11 \quad j = 5$

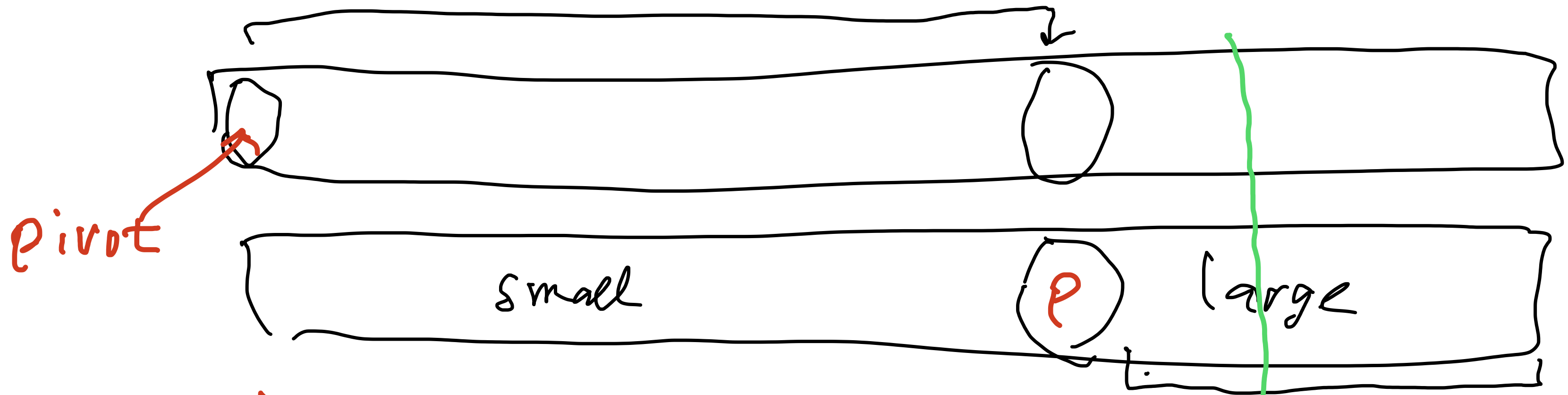
$$\Theta\left(\frac{n}{2} \cdot n\right) = \Theta(n^2)$$

$\Theta(1)$	constant	$\Theta(n^2)$	quadratic
$\Theta(\log n)$	logarithmic	$\Theta(n^3)$	cubic
$\Theta(n)$	linear	$\Theta(n^4)$	...
$\Theta(n \log n)$	—	$\Theta(2^n)$	exponential

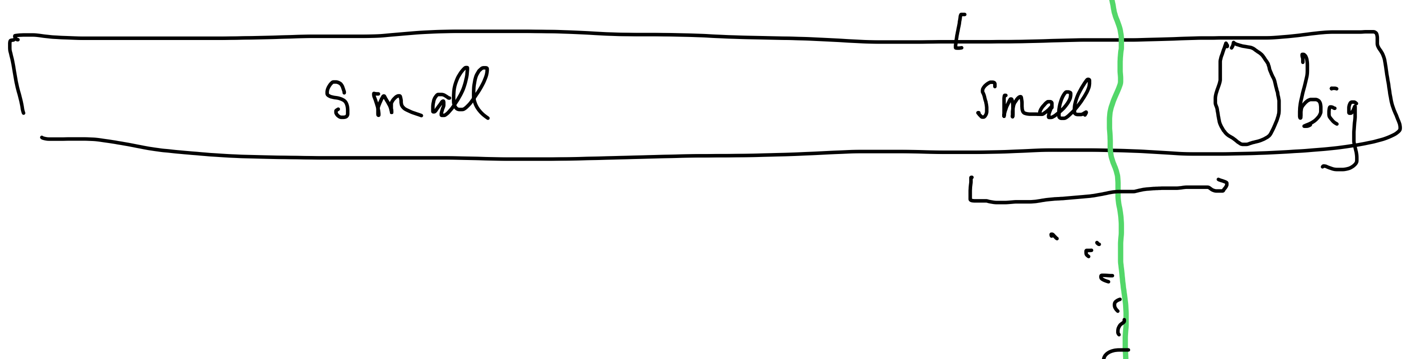
QuickSelect C.A.R. Hoare

target





partition: separate small, pivot, large in that order.



Complexity of finding median by repeated partitioning?

$$C_n = n + \frac{n}{2} + \frac{n}{4} + \cdots = 2n = \Theta(n)$$

(lucky) $C_n = n + C_{n/2} = n + \overset{a}{1} \cdot C_{n/\underset{b}{2}}$

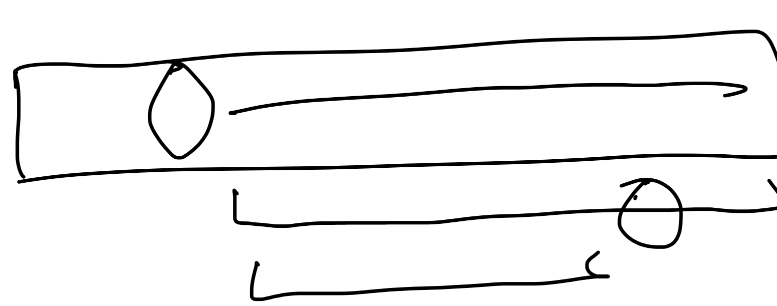
$f(n) = n^{1} \leftarrow k$

$$\begin{array}{cc} a & b^k \\ 1 < & 2^1 \end{array}$$

$$C_n = \Theta(n^k) = \Theta(n)$$

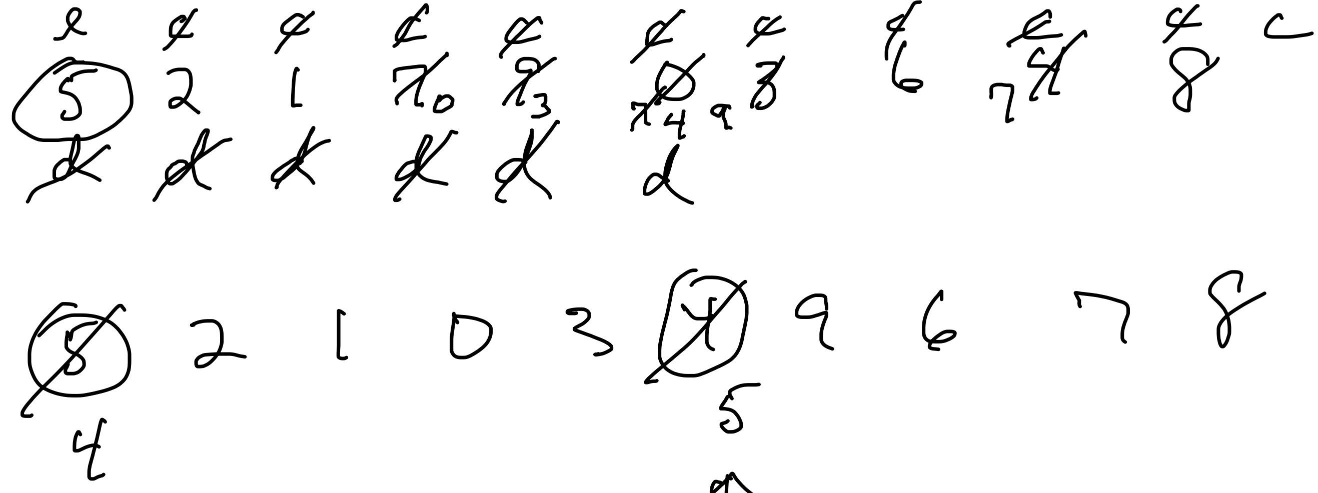
(unlucky) $C_n = n + 1 C_{\frac{3}{4}n}$

n^1, $k=1$; a ; $b = \frac{4}{3}$



$$\begin{array}{cc} a & b^k \\ 1 & \left(\frac{4}{3}\right)^1 \end{array}$$

$a < b^k$ $\quad C_n = \Theta(n^k) = \Theta(n)$



Complexity of partitioning : $\Theta(n)$

repeated partitioning to find jth : $\Theta(n)$

Sorting : n values (keys)

stable : ties stay in original order

in-place : no extra space (except $\Theta(1)$ memory)

complexity : $\Omega(n \log n)$

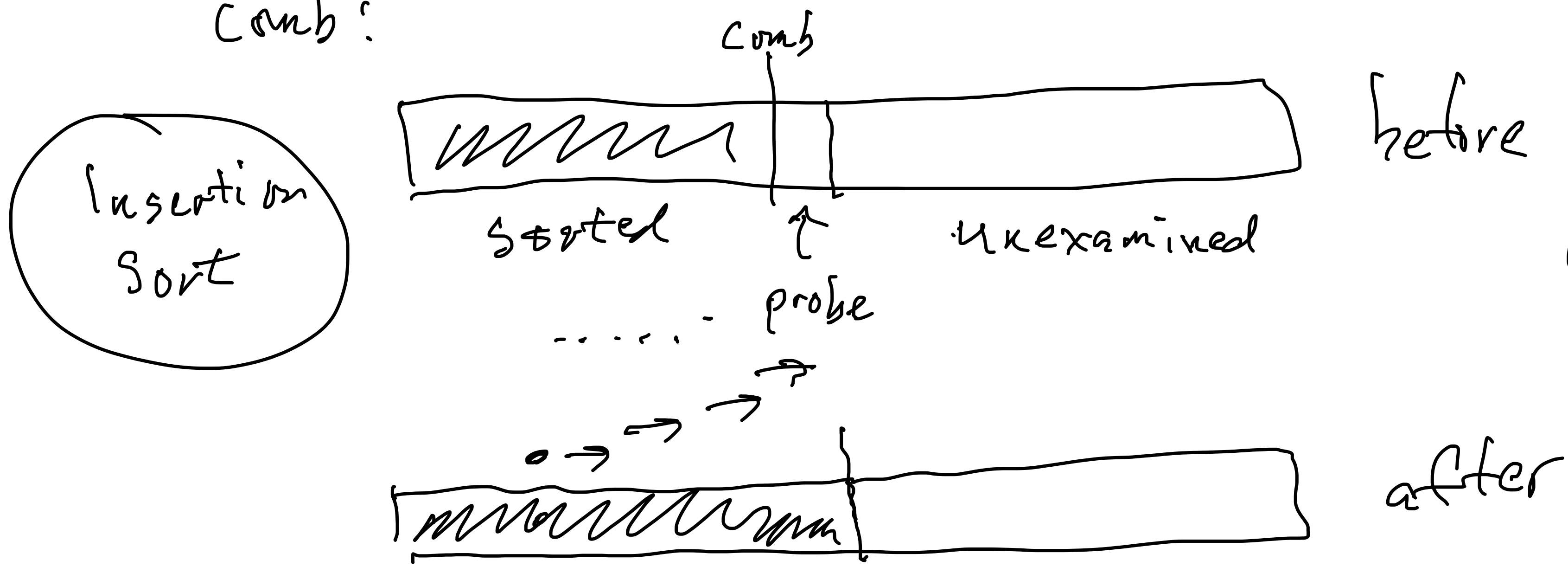
good : $\Theta(n \log n)$: quick, merge, heap, shell sort

bad : $\Theta(n^2)$

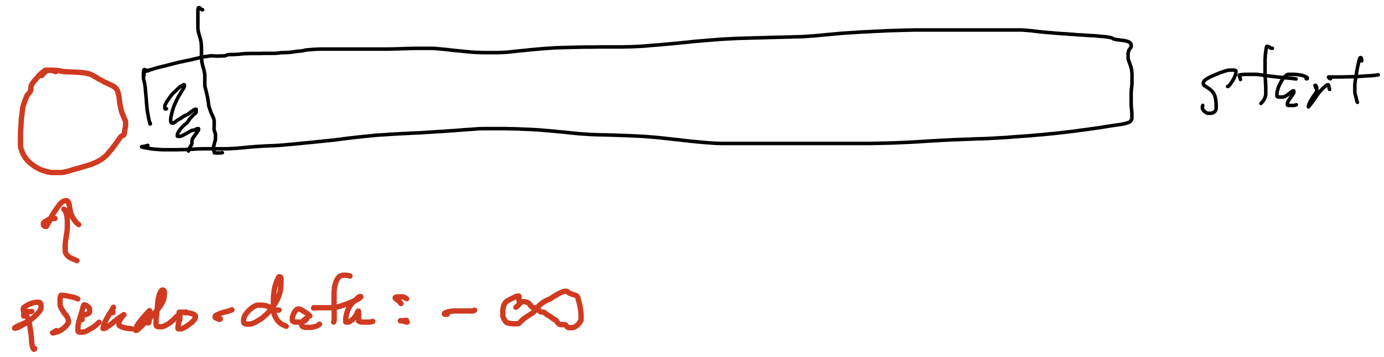
bubble, shaker, insertion, selection

---

comb :

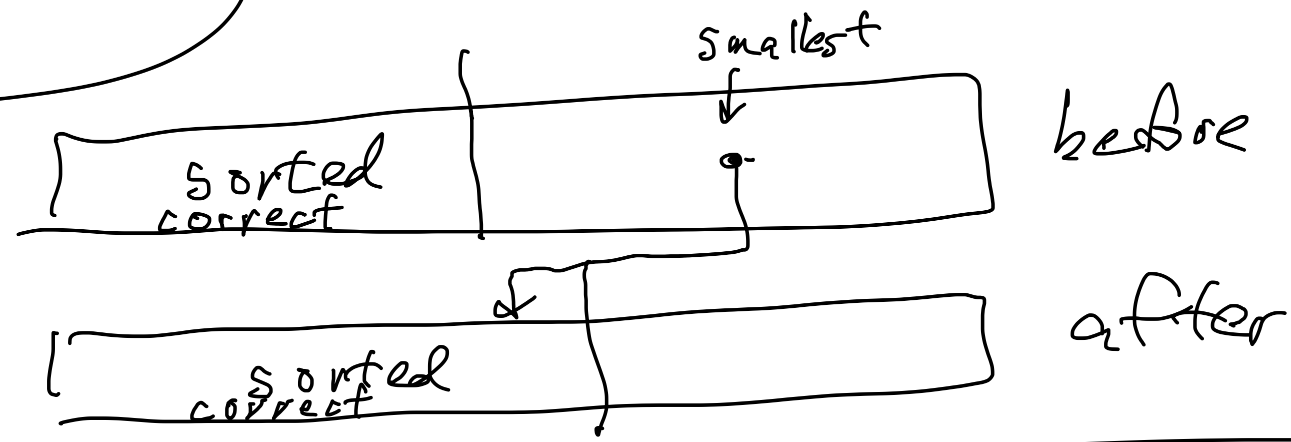


$\Theta(n^2)$

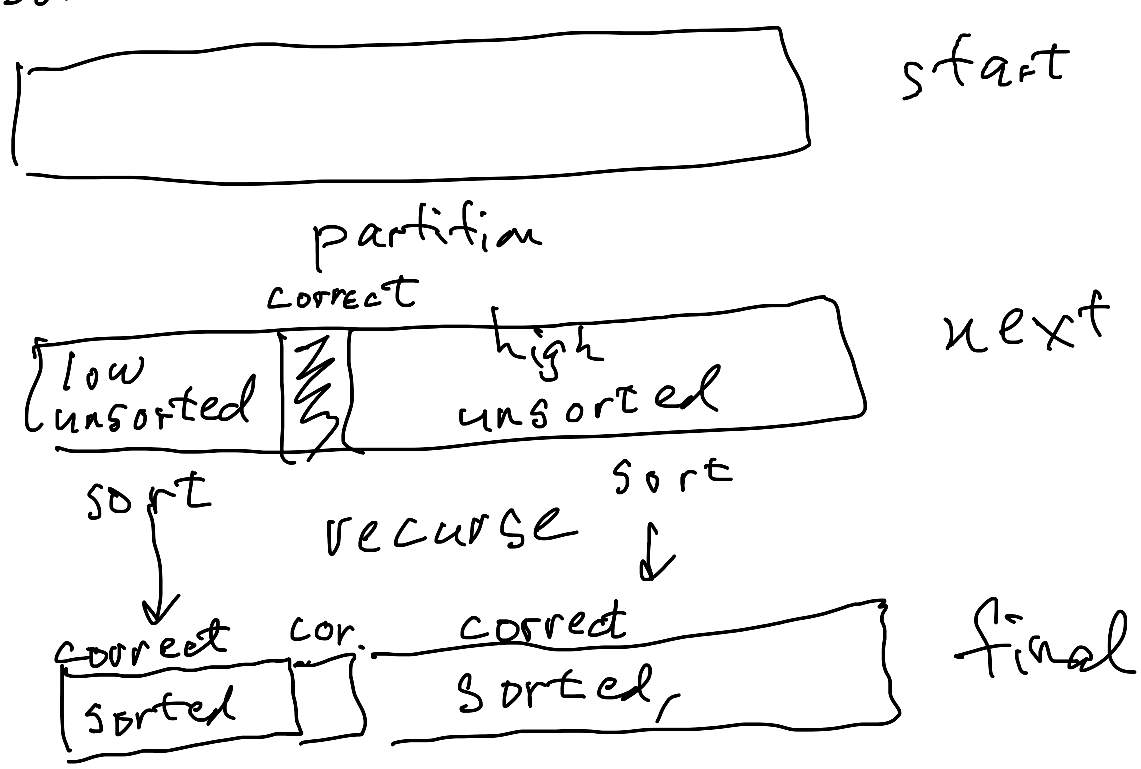


Selection sort

$\Theta(n^2)$



Quicksort

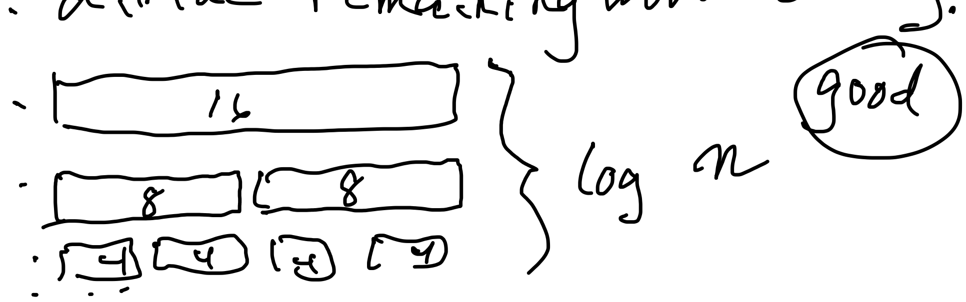


optimizations

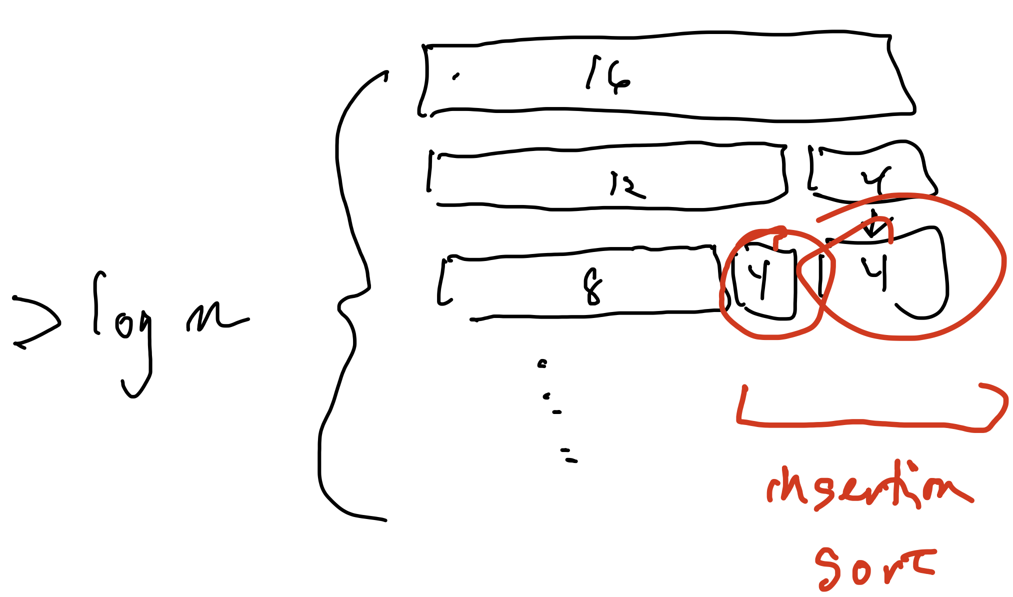
1) pick pivot: median of 3 elements

5

reason: divide remaining work evenly.



bad

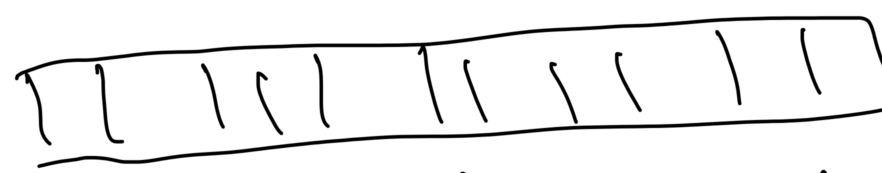


2) Don't recurse for regions smaller than S (span).

good S: depends on implementation

$10 \le S \le 100$

before insertion sort

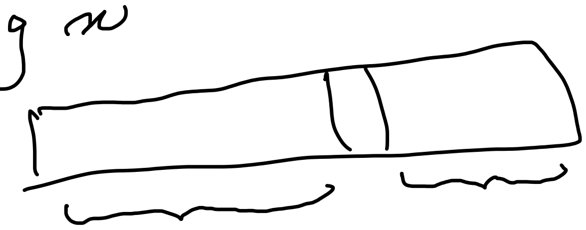


good for insertion sort, because no element moves more than S cells.

Analysis:

depth of recursion $\approx \log n$

at every level,



partitioning takes $\Theta(n)$

$\Rightarrow$ total time is $\Theta((\log n)n)$

$= \Theta(n \log n)$

Recursion theorem:

Lucky: $C_n = n^1 + 2C_{n/2}$

size of problem

$a = 2$

$b = 2$

$k = 1$

$a = b^k$ $\quad 2 = 2^1$

$C_n = \Theta(n^k \log n) = \Theta(n \log n)$

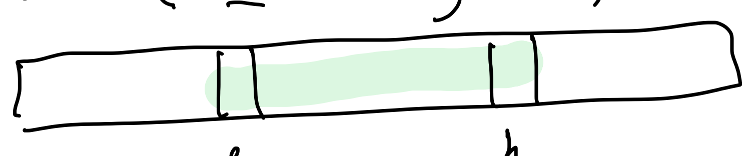
Unlucky: $C_n = n + C_{n/3} + C_{2n/3} < n + 2C_{2n/3}$

$a = 2$
$b = 3/2$
$k = 1$
$b^k = 3/2$

$a > b^k$

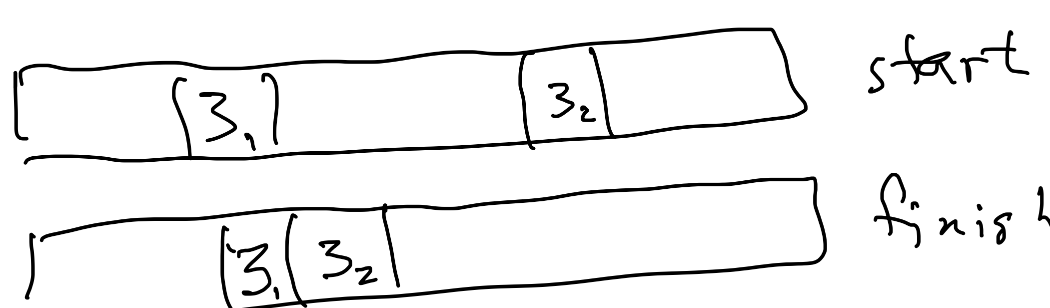
$$C_n < \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_{3/2} 2}\right)$$
$$= \Theta\left(n^{1.71}\right)$$

```
void quickSort(int array[], int low, int high)
{
    //
```




```
    if (high - low <= 0) return;
                   or span
    int mid = partition(array, low, high);
    quickSort(low, mid-1);
    quickSort(mid+1, high);
}
```

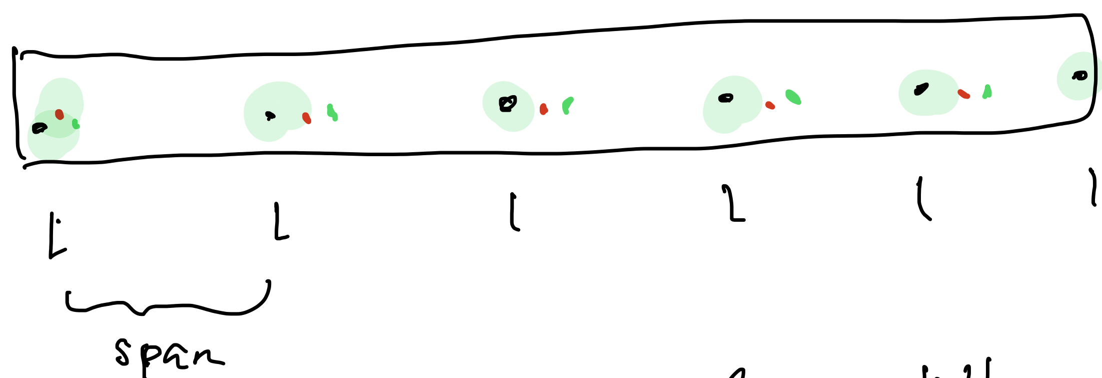
stability?



not stable. because partitioning is not stable.

Shell sort: Donald Shell (1959)

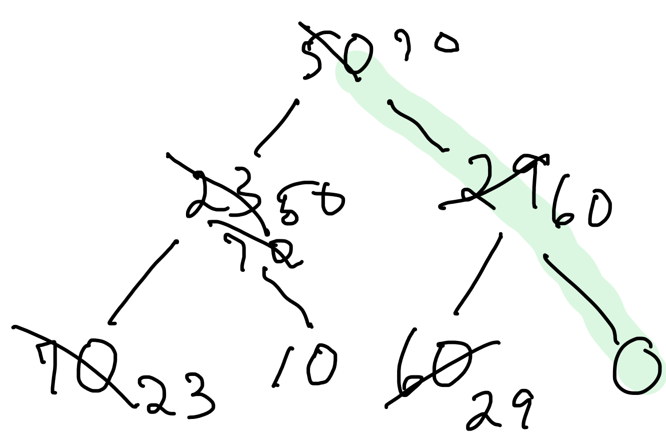
repeated insertion sorts




```
for span = (large, smaller, still smaller, 1) {
    for offset = (0 .. span-1) {
        insertionSort(offset, offset+span, offset+2span)
    }
}
```

---

heap: binary tree, balanced, strange sort criterion.



rule: parent $\geq$ child.

sift up

insert: $\Theta(\log n)$