Array-backed List Runtime Complexity

operations:

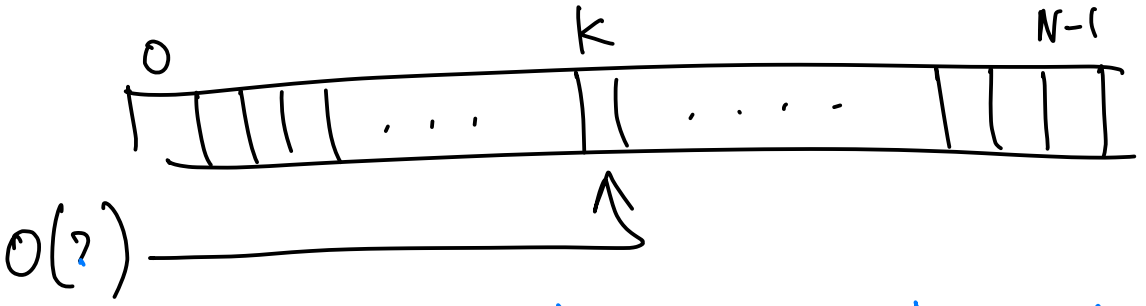
- indexing
- search (unsorted)
- search (sorted)
- deletion
- append
- insertion

} given list of N elements.

indexing

i.e., $x = lst[k]$ / $lst[k] = x$

(__getitem__) (__setitem__)



array indexing = compute base + displacement address

$= O(1)$

search (unsorted)

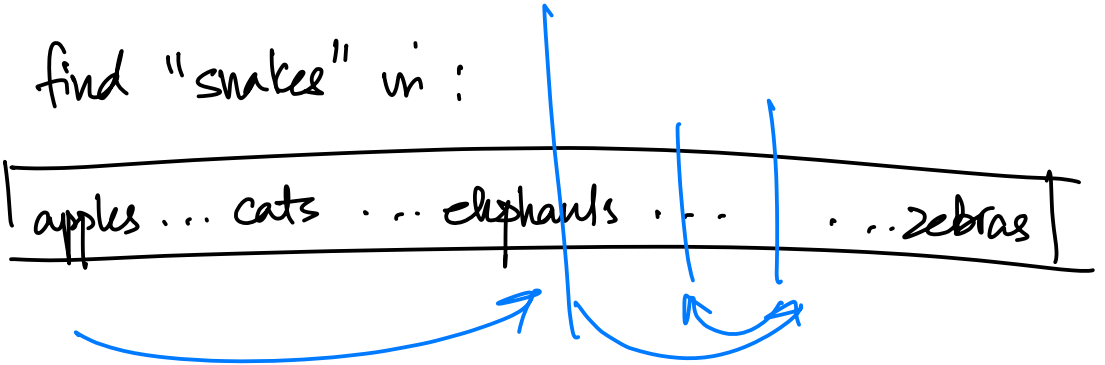
e.g., x in lst ?

(_ contains _)

$= O(N)$

search (sorted)

eg., find "snakes" in:



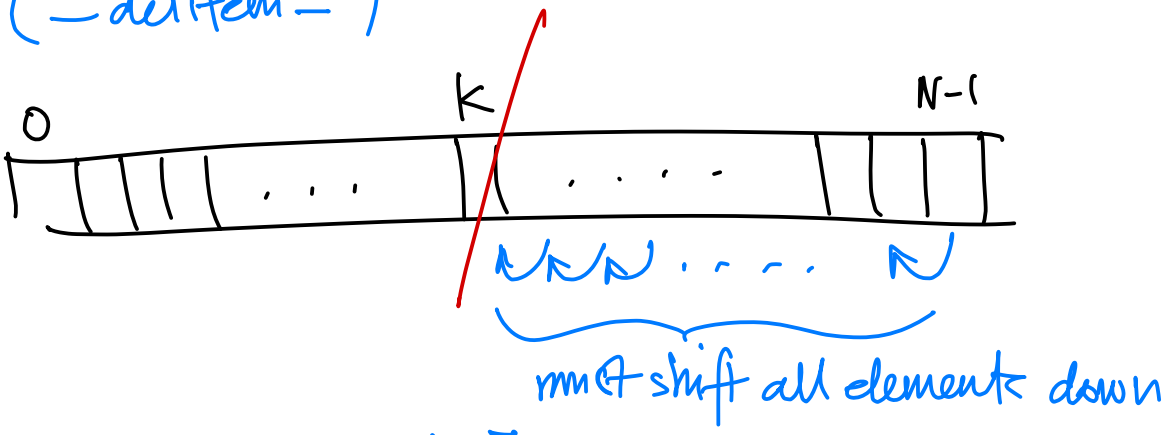
- can implement binary search! (able to jump into middle of list in O(1) time)

$= O(\log N)$

deletion

i.e., del lst[k]

(__delitem__)

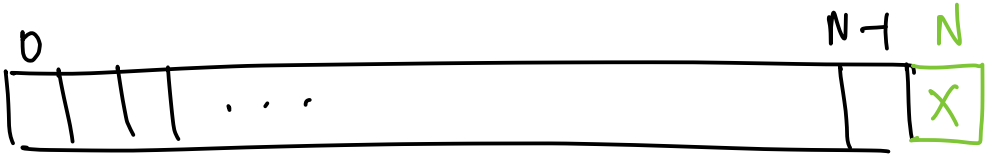


worst case: del lst[0]

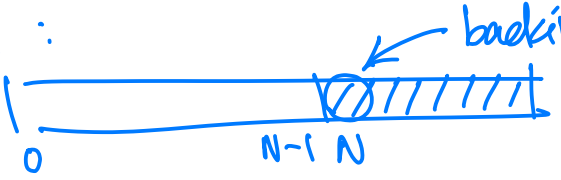
$= O(N)$

append

i.e., lst.append(x)

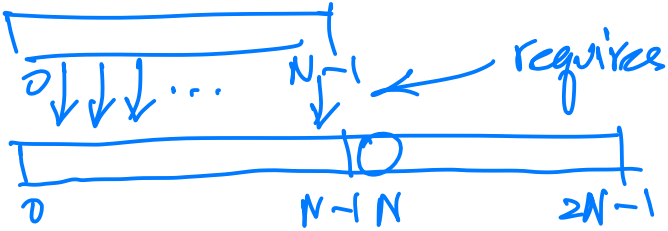


case 1 :



= O(1)

case 2 :

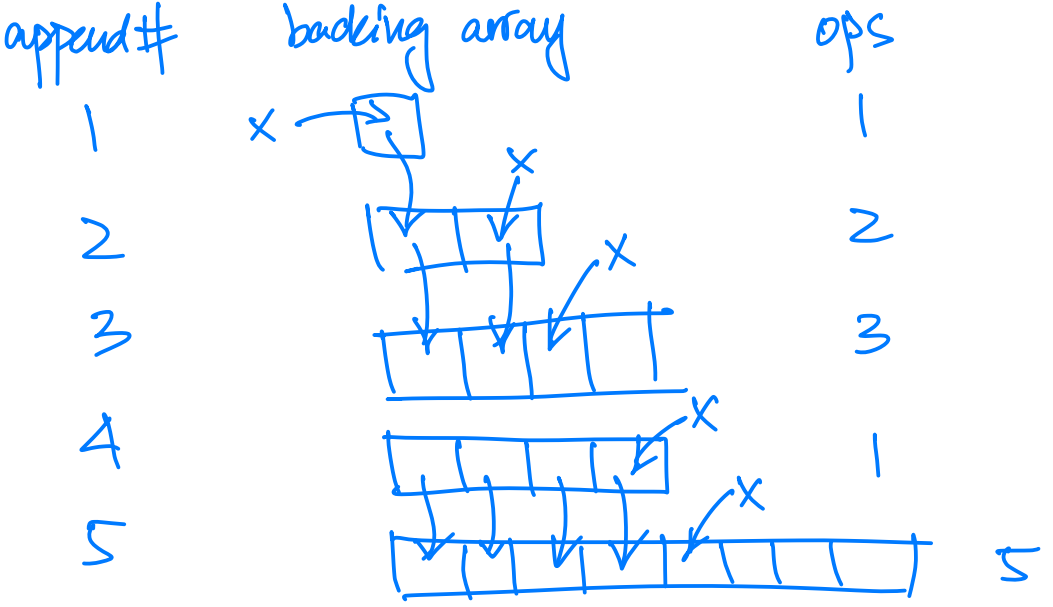


= O(N)

worst case!

append : a more granular analysis

- how many operations (only counting data movements) per append, starting w/ backing array of capacity 1, doubling it on each expansion?



append : a more granular analysis

- how many operations (only counting data movements) per append, starting w/ backing array of capacity 1, doubling it on each expansion?

append #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...
capacity	1	2	4	4	8	8	8	8	16	16	..	..	..	..	..	32	..	..	..
cost	1	2	3	1	5	1	1	1	9	1	1	..	..	..	..	17	1	..	..

append #	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 ...
capacity	1 2 4 4 8 8 8 8 16 16 .. .. 32 .. .. ..
cost	1 2 3 1 5 1 1 1 9 1 1 .. .. 17 1 .. ..

- we can try to amortize (spread out) the expensive appends over the cheap ones

- is there some fixed amortized cost that will work?

Amortization worksheet

try fixed cost of 2

append #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...
capacity	1	2	4	4	8	8	8	8	16	16	..	..	..	..	..	32	..	..	..
actual cost	1	2	3	1	5	1	1	1	9	1	1	..	..	..	..	17	1	..	..
amortized cost	2	2	2	2	2	..	..	..	..	..	..	..	..	..	..	..	..	..	..
Δ credits	+1	0	-1	+1	-3	+1	+1	+1	-7	..	..	..	..	..	..	..	..	..	..
credits saved	1	1	0	1	-2	-1	0	1	-6	..	..	..	..	..	..	..	..	..	..

← not sustainable!

Amortization worksheet

try fixed cost of 3

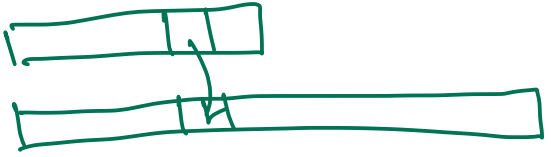
append #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...
capacity	1	2	4	4	8	8	8	8	16	16	..	..	..	..	..	32	..	..	..
actual cost	1	2	3	1	5	1	1	1	9	1	1	..	..	..	..	17	1	..	..
amortized cost	3	3	3	3	3	3	3	3	..	..	..	..	..	..	..				
Δ credits	+2	+1	0	+2	-2	+2	+2	+2	-6	+2	+2	..	..	..	..		+2	-14	
credits saved	2	3	3	5	3	5	7	9	3								17	3	

ok!

Intuition behind cost of 3:

- one credit is paid to copy element into the array.

- one credit is paid to move the element to a new array for the first time



- one credit is paid to move the element at $i - 2^{\lfloor \log_2 i \rfloor}$

append is O(1) amortized