Announcements

- S27 (amortized) → HW11. Only need 1x to get it to count
- Midterm Sat—Sun S12—now (not S25)

Today

- Doubling array
- Amortized analysis

} (S27)

Array API

new array[n]

A[i].get(i) — get item @ position i

Come back to this — A[i]=x — set new item @ position i

} O(1)-time

(List) add(x) — adds x to the end

baseArray — array, fixed size, allocated in contiguous block of memory

length — int, keeps track of # items in our DoublingList

```
new DL:
    baseArray = new array[2]
    length = 0
```

arbitrary

time: O(1)

```
get(i):
    if i ≤ length-1:
        return baseArray[i]
    else
        error
```

O(1)

```
add(x):
    if length < len(baseArray):
        baseArray[length] = x
        length = length + 1
    else: // baseArray full
        nextArray = new array[2 * length]
        for i ∈ [0.. length-1]:
            nextArray[i] = baseArray[i]
        (deallocate baseArray)
        baseArray = nextArray
        baseArray[length] = x
        length += 1
```

"doubling"

O(1) OR Θ(n)

n = # items in list

e.g.:

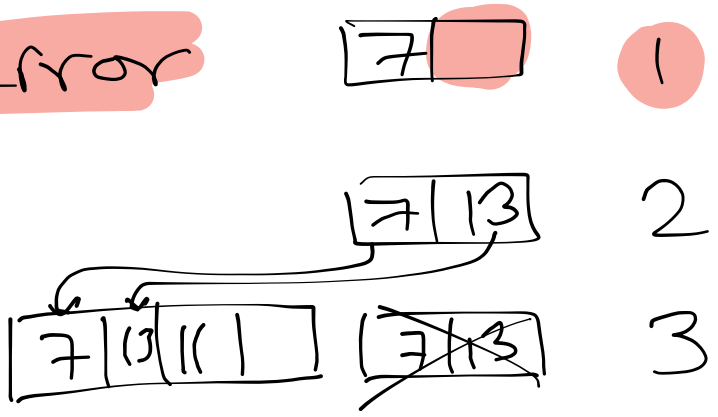
```
list = new DL()
list.add(7)
```

memory	length
[ ]	0
[7]	1

list.get(1) ⟶ error

list.add(13)

list.add(11)



Amortized Analysis

Data structure

Sequence of T operations

$\#$ ops

(Want to know: worst-case total runtime of operations, as a function of T)

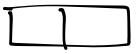
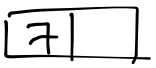
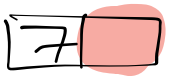
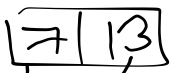
Amortized runtime (sequence of ops) =

$$\frac{\text{total runtime}}{\#\text{ ops}}$$

Worst-case sequence of operations

report amortized time for that sequence.

e.g.

	memory	length	runtime
`list = new DL()`		0	c
`list.add(7)`		1	c
`list.get(1)` → error		1	c
`list.add(13)`		2	c
`list.add(11)`		3	3c
			n+1
			7c

T = # ops = 5

amortized cost of this sequence = $\frac{7c}{5}$

What is a worst-case sequence of operations for Dubling List?

try:

	time
new DL	O(1)
add	O(1)

T-2 { get O(1)
get O(1)
.
.
.
get O(1) }

$$\frac{+\ O(1)}{T \cdot O(1)}$$

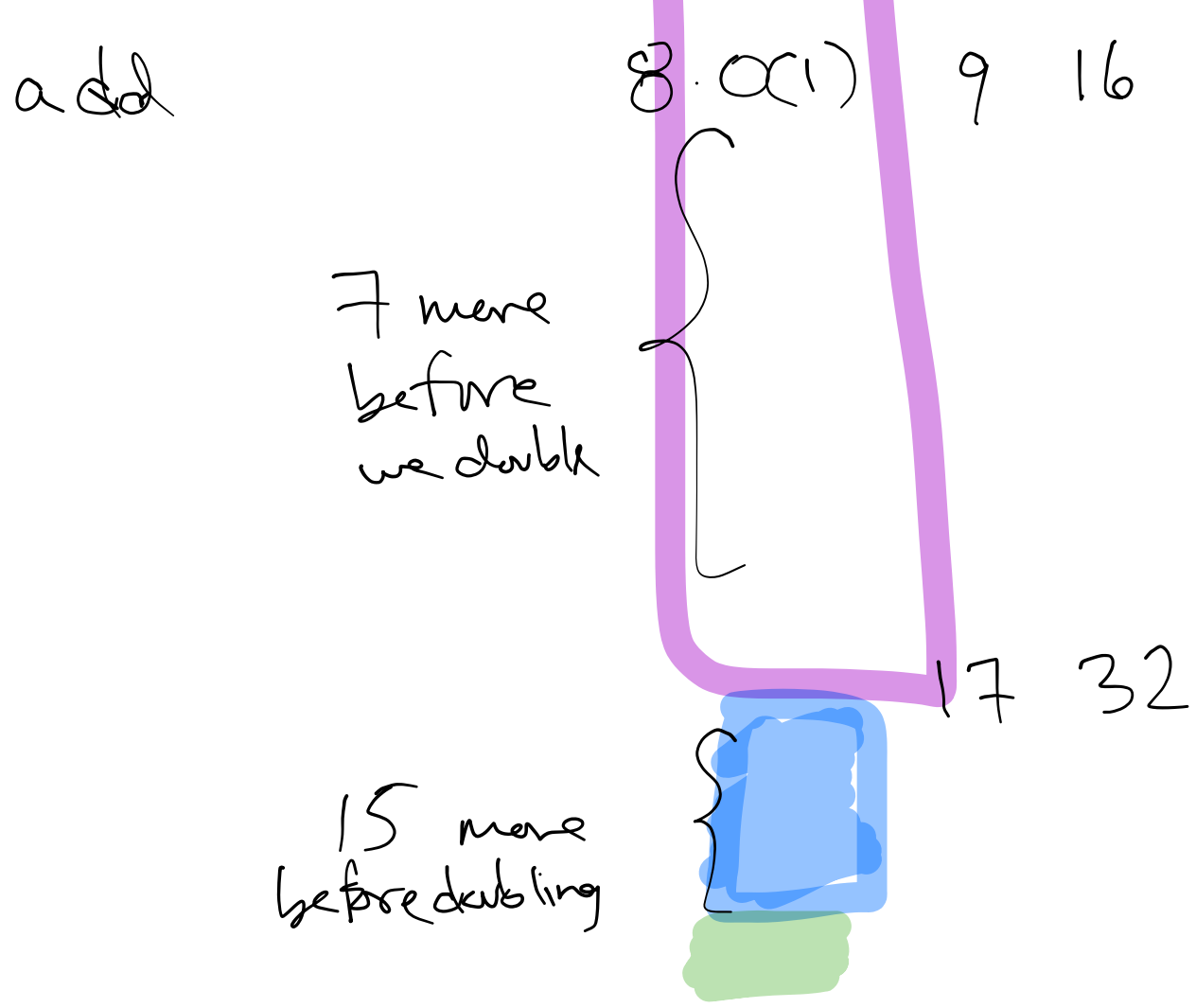
$$\text{amortized} = \frac{T \cdot O(1)}{T} = O(1)$$

WORST-case

new DL

T-1 { add
add
add
.
.
.
add }

	time	length	len(barArray)
	O(1)	0	2
	O(1)	1	2
	O(1)	2	2
2 ·	O(1)	3	4
1	O(1)	4	4
4 ·	O(1)	5	8
	O(1)	6	8
3	O(1)	7	8
	O(1)	8	8

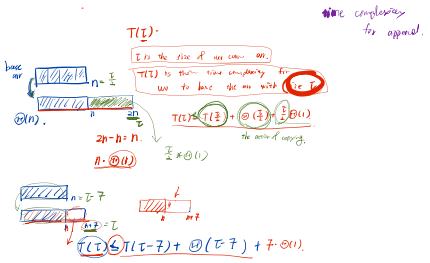


$T(\tau)$:= total time of τ add operations

worst case is when last op is a doubling

doubling list above

$$T(\tau) \leq \underbrace{O(\tau)}_{\text{for last}} + \underbrace{\frac{\tau}{2} \cdot O(1)}_{\frac{\tau}{2} \text{ adds}} + T\left(\frac{\tau}{2}\right)$$



doubling operation

between last two doubling ops

$$\leq c \cdot T + c \cdot \frac{T}{2} + T\left(\frac{T}{2}\right)$$

$$= \frac{3}{2} c T + T\left(\frac{T}{2}\right)$$

unroll

$$= \frac{3}{2} c T + \frac{3}{2} c \left(\frac{T}{2}\right) + T\left(\frac{T}{4}\right)$$

$$= \frac{3}{2} c \left(T + \frac{T}{2}\right) + T\left(\frac{T}{4}\right)$$

Unroll

$$= \frac{3}{2} c \left(T + \frac{T}{2}\right) + \frac{3}{2} c \frac{T}{4} + T\left(\frac{T}{8}\right)$$

$$= \frac{3}{2} c \left(T + \frac{T}{2} + \frac{T}{4}\right) + T\left(\frac{T}{8}\right)$$

$$= \frac{3}{2} c \left(T + \frac{T}{2} + \frac{T}{4} + \frac{T}{8}\right) + T\left(\frac{T}{16}\right)$$

$$= \frac{3}{2} c T \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^i + T\left(\frac{T}{2^k}\right)$$

Solve for base case $\frac{T}{2^k} \leq 1$

$$\log_2 T \leq k$$

plug back in

$$= \frac{3}{2} cT \sum_{i=0}^{\log_2 T} \left(\frac{1}{2}\right)^i + \underbrace{\text{base case}}_{O(1)}$$

Note: $\sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = 2$

$$\leq 3cT \leq O(T)$$

Total time of T adds $\leq O(T)$

Amortized time $\leq \frac{O(T)}{T} = O(1)$

$$\frac{T(T)}{T} = \frac{O(T)}{T} = O(1)$$

List

creation	$O(1)$	worst-case
get	$O(1)$	worst-case
add	$O(1)$	amortized

Using data structure w/ better amortized runtime to get algorithm w/ better worst-case runtime.

Dijkstra's Algorithm (shortest paths)

Priority queue
- add V times
- update priority E times

Balanced binary tree — $\Theta(\log V)$ runtime for all ops

$\rightsquigarrow O((V+E)\log V)$ time for algo

Fibonacci heap
- add $O(\log n)$ # items = V
- update amortized $O(1)$ time. same

$\Rightarrow$ E updates take $O(E)$ time

$\Rightarrow$ total runtime $O(V \log V + E)$

		Dijkstra BBT	Dijkstra Fib heap
sparse case	$E = O(V)$	$O(V \log V)$	$O(V \log V)$
dense case	$E = \Omega(V^{1+\varepsilon})$	$O(V^{1+\varepsilon} \log V)$	$O(V^{1+\varepsilon})$