

# Lecture 15: Sequences

- work: index access is  $O(1)$  work  
Fast

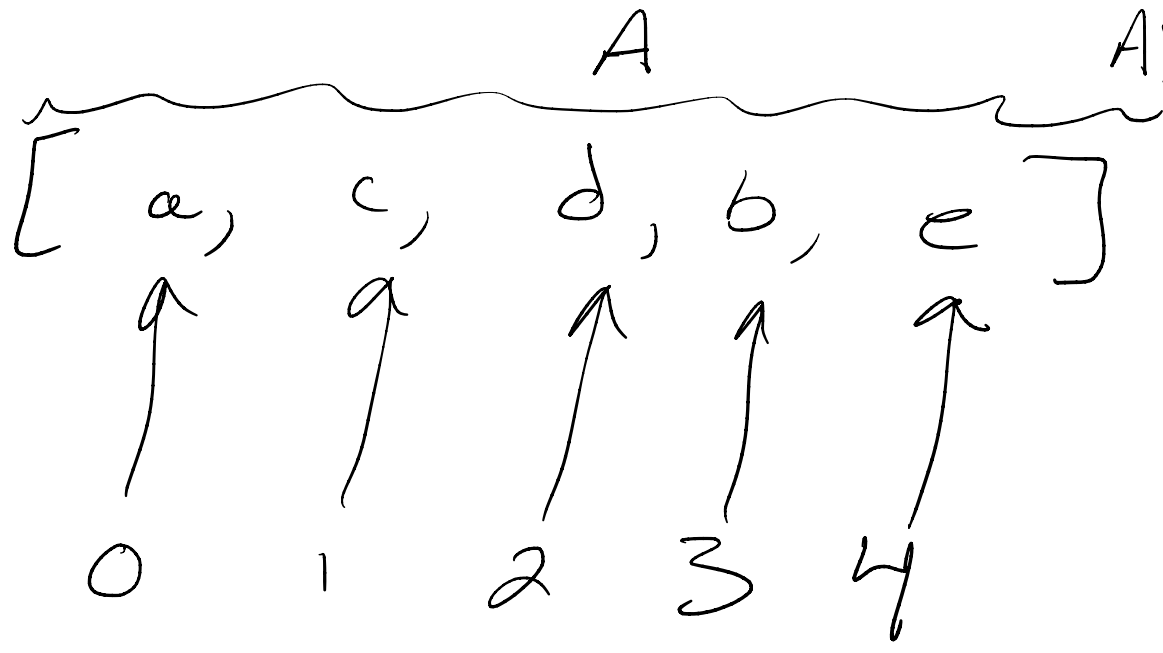
- span: map is  $O(1)$  span

# Array-backed sequences

index calculations

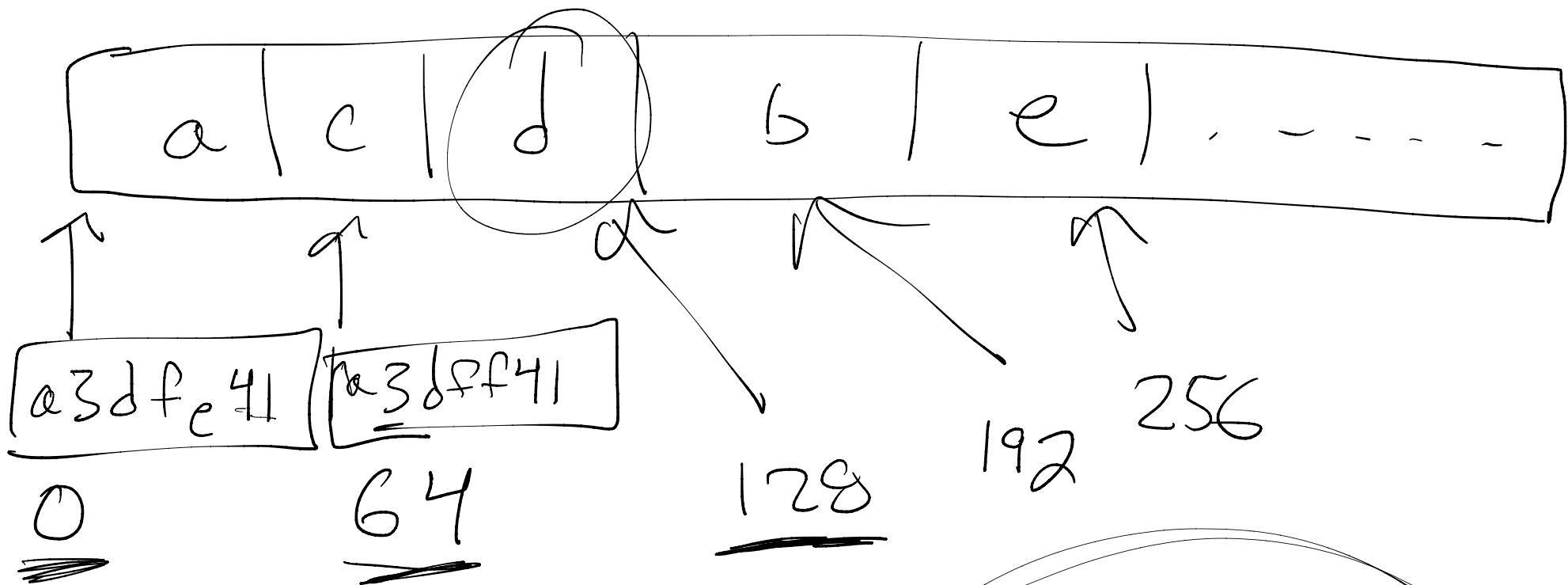
$A[i]$  gets elt at pos.  $i$   
 $A[i] = f$  } update

elements



index

"0-indexed"



$A[2]$

bounds:

$A[i] \quad 0 \leq i < \text{length } A$

work of  $A[i]$  is  $O(1)$

fun nth (l: 'a list, n: int) = 'a =  
case (n, l)

(0, x::\_) => x

| (\_, x::xs) => nth(xs, n-1)

$O(n)$  work

n either n  
or  
length(l)

assume  $0 \leq n < \text{length}(l)$

accessing elt. at position  $i$

arrays :  $O(1)$

lists :  $O(n)$

trees :  $O(\log n)$

↑  
(balanced)

[a, c, d, f, b]

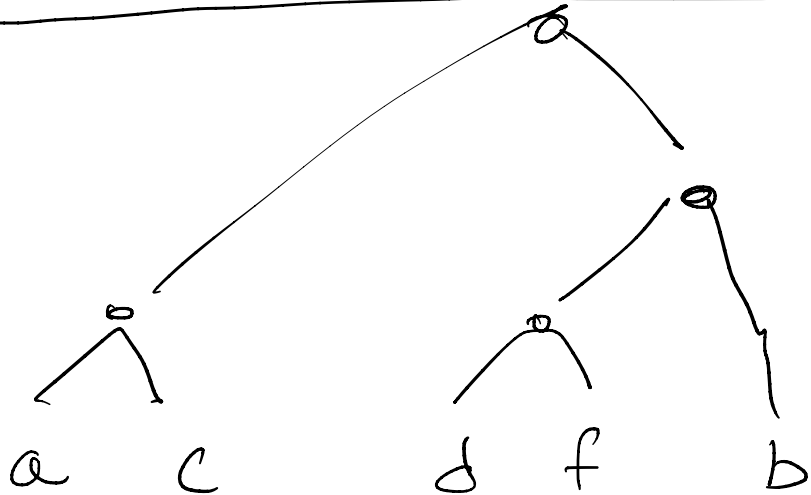
↓ map capitalize

list:

span is

$O(n)$

[A, C, D, F, B]

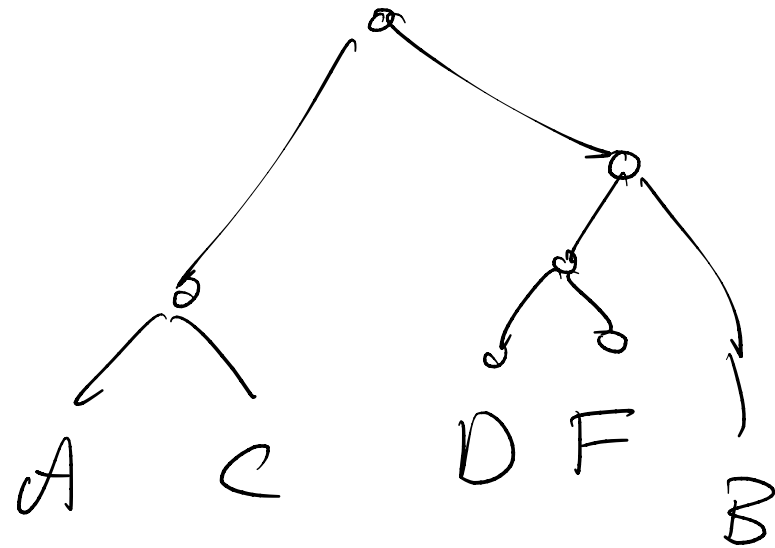


$O(\log n)$

span



if  
balanced



Conceptually:

$\langle a, c, d, f, b \rangle$

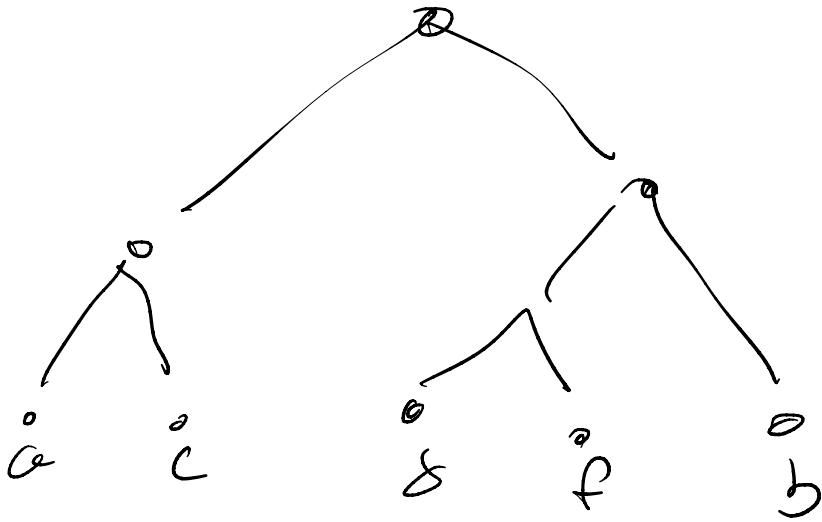


$\langle A, C, D, F, B \rangle$

should  
be  
OCI) !  
span

"ordered collection of elts"

① [a, c, d, f, b]



③ [ a | c | d | f | b ]



type 'a Seq.seq

array-backed

val Seq.map: ('a → 'b \* 'a Seq.seq) → 'b Seq.seq

val Seq.length: 'a Seq.seq → int

val Seq.nth: int \* 'a Seq.seq → 'a

val Seq.reduce: (('a \* 'a → 'a) \*  
'a \*  
'a Seq.seq)  
→ 'a

⋮

# Abstract specification of

① behavior

② work and span

① Behavior

$$\text{map}(f, \langle x_0, x_1, \dots, x_{n-1} \rangle) \\ = \langle f x_0, f x_1, \dots, f x_{n-1} \rangle$$

Seg:   
↳ list   
↳ Seg.seg

② work: sum of  $W_f$  on  $x_i$  for all  $i$

span: max of  $S_f$  on  $x_i$  for all  $i$

if  $f$  is constant time      Work is  $O(n)$        $n = \text{length}$    
span is  $O(1)$                       of input

Upper: char  $\rightarrow$  char  
length  $n$

SeqMap f (upper, <a, c, d, f, b>)

$O(n)$  work

$O(1)$  space

# Seq. length

① Behavior:

$$\text{Seq. length } \langle x_0, x_1, \dots, x_{n-1} \rangle \\ = \\ n$$

② work }  $O(1)$   
span }  ~~$O(1)$~~

Seq. nth

"A[i]"

① Behavior

$\text{Seq.nth}(i, \langle x_0, x_1, \dots, x_{n-1} \rangle)$

$= x_i$  if  $0 \leq i < n$

or raise Range otherwise

② work }  
space }  $O(1)$

Reduce  $= ((a * a \rightarrow a) * \dots * a \text{ seq seq})$  Combine  
 $\rightarrow a \text{ seq seq}$  base case

① Behavior:

$$\begin{aligned} & \text{reduce}(c, b, \langle x_0, x_1, \dots, x_{n-1} \rangle) \\ &= ((x_0 \odot x_1) \odot (x_2 \odot x_3 \odot \dots)) \odot \dots \odot x_{n-1} \end{aligned}$$

balanced tree parenthesization

or  $= b$  if  $\langle \rangle$

$$x_1 + (x_2 + (x_3 + x_4))$$

$$(x_1 + x_2) + (x_3 + x_4)$$

$$((x_1 + x_2) + x_3) + x_4$$

associativity

① assume associative; all =  
(but: floating point)

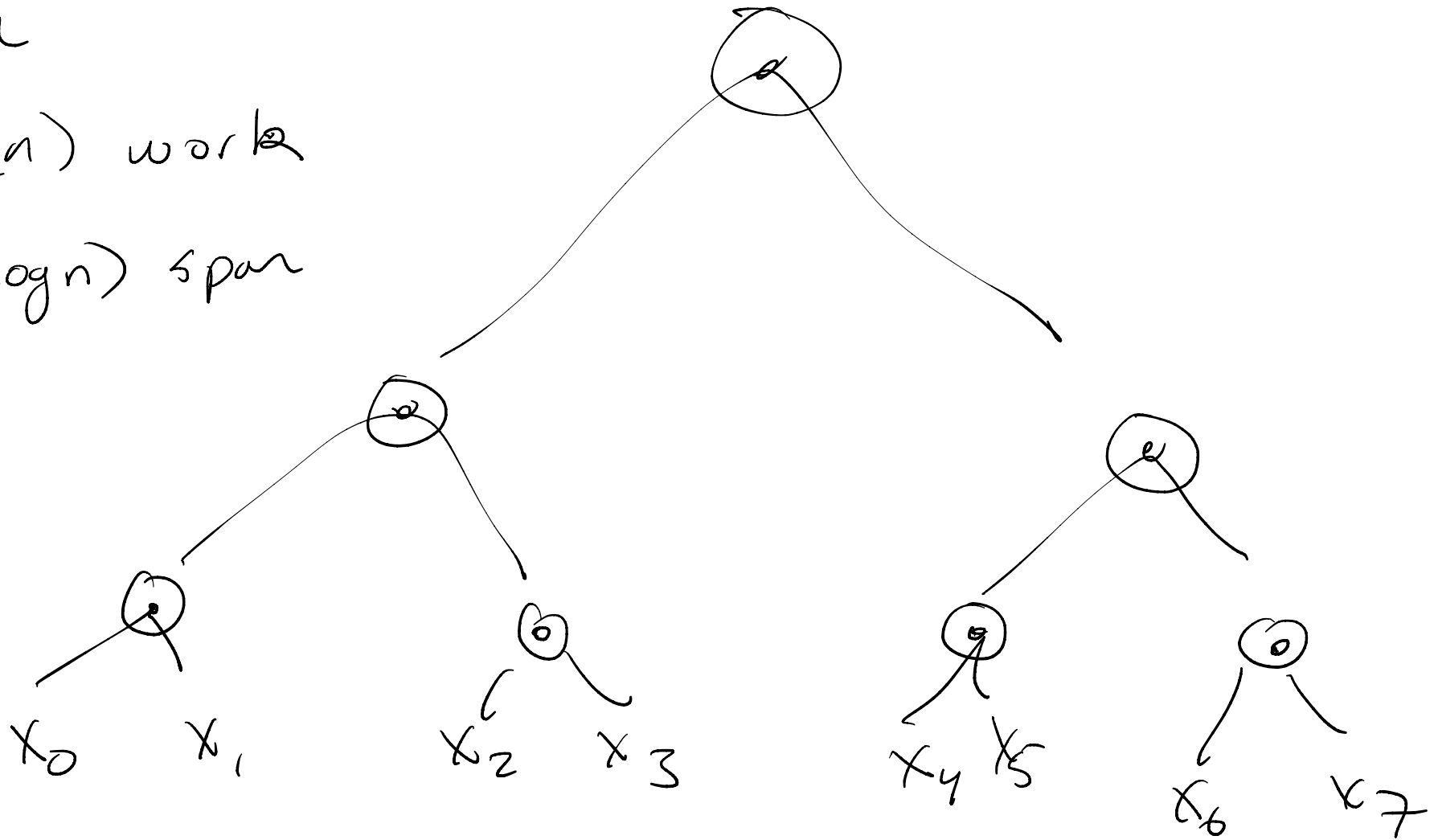
② fix a parenthesization

If  $\Theta$  constant time

then

$O(n)$  work

$O(\log n)$  span





Analyze reduce

: write  
+ solve  
recurrence

$$\text{reduce}(0, b, \langle s \rangle) = b$$

$$\text{reduce}(0, b, s) =$$

⊙ ( reduce on left,  
reduce on right )

calculating  
left + right  $O(n)$

first half  
of S

second  
half of S

fun sum (s: int Seq.seq) : int =  
 Seq.reduce (fn (x, y) => x + y,  
 0,  
 s)

$O(n)$   
 work  
 $O(\log n)$   
 span

$$\text{sum } \langle 1, 2, 4, 8 \rangle = 15$$

$$1 + 2 + 4 + 8 = 15$$

$$\text{sum } \langle \rangle = 0$$

count



```
fun count(s: (Int Seq, Seq) Seq, seq) : Int =  
  sum( Seq ◦ map(sum, s) )
```

count

$\langle \langle 1, 0, 1, 0, 0 \rangle \rangle$

$\langle \langle 0, 1, 0, 0, 0 \rangle \rangle$

$\xrightarrow{\text{sum}} \langle \text{sum} \langle 1, 0, 1, 0, 0 \rangle, \text{sum} \langle 0, 1, 0, 0, 0 \rangle \rangle$

$\xrightarrow{\text{sum}} \langle 2, 1, 0, 0 \rangle$

$\xrightarrow{\text{sum}} 3$

Assume  $n \times n$

|           | Input size   | Work     | Span        |
|-----------|--------------|----------|-------------|
| Inner sum | always $n$   | $O(n)$   | $O(\log n)$ |
| map       | $n \times n$ | $O(n^2)$ | $O(\log n)$ |
| outer sum | $n$          | $O(n)$   | $O(\log n)$ |
| Overall   |              | $O(n^2)$ | $O(\log n)$ |

Diagram illustrating the complexity analysis of a matrix operation, showing the relationship between Input size, Work, and Span across different stages.

The table is structured as follows:

- Input size:** The size of the input data at each stage.
- Work:** The total amount of work performed at each stage.
- Span:** The maximum number of processors that can be active simultaneously at each stage.

Key observations from the diagram:

- The **map** stage has the highest work complexity,  $O(n^2)$ .
- The **map** stage has a span of  $O(\log n)$ .
- The **map** stage's work complexity is derived from the **Inner sum** stage's work complexity,  $O(n)$ .
- The **map** stage's span is derived from the **Inner sum** stage's span,  $O(\log n)$ .
- The **map** stage's work complexity is the sum of the **map** stage's work complexity and the **outer sum** stage's work complexity,  $O(n)$ .
- The **map** stage's span is the sum of the **map** stage's span and the **outer sum** stage's span,  $O(\log n)$ .
- The **Overall** work complexity is  $O(n^2)$ .
- The **Overall** span is  $O(\log n)$ .

Brent's principle:

time on  $p$  processors

is  $O\left(\max\left(\frac{W}{p}, S\right)\right)$

can use  $\approx \frac{W}{S} \approx \frac{n^2}{\log n}$

Processors

