Lecture 4

Algorithms & Data Structures

Goldsmiths Computing

October 22, 2018

Outline

Introduction

Linked lists

Stacks

Queues

Outline

Introduction

Linked lists

Stacks

Queues

Lecture

- Dynamic arrays
  - handy data structure
  - generalization of one-dimensional array

Lecture

- Dynamic arrays
  - handy data structure
  - generalization of one-dimensional array
- The random access machine
  - simple model of a computer
  - discard irrelevant details
  - similar *enough* to real machines
- big-O notation
  - describe growth of functions
  - discard irrelevant details
  - large n only

Lecture

- Dynamic arrays
  - handy data structure
  - generalization of one-dimensional array
- The random access machine
  - simple model of a computer
  - discard irrelevant details
  - similar *enough* to real machines
- big-O notation
  - describe growth of functions
  - discard irrelevant details
  - large n only
- Analysis of vector operations
  - using RAM and big-O to see:
    - access to storage the same regardless of implementation strategy
    - length operation is is $O(\mathbf{1})$ for length-data but $O(N)$ for sentinel implementation

Lab

- Be a data structure implementor
  1. implement dynamic arrays

Lab

- Be a data structure implementor
  1. implement dynamic arrays
  2. ... without using your language's dynamic array data structure
- Measure implementation choices
  1. how many operations for particular strategies?
  2. how much unused space?

VLE activities

big-O quiz

Statistics so far:

- 245 attempts: average mark 5.72
- 82 students: average mark 6.35
  - 16 under 4.00, 34 over 6.99, 13 at 10.00

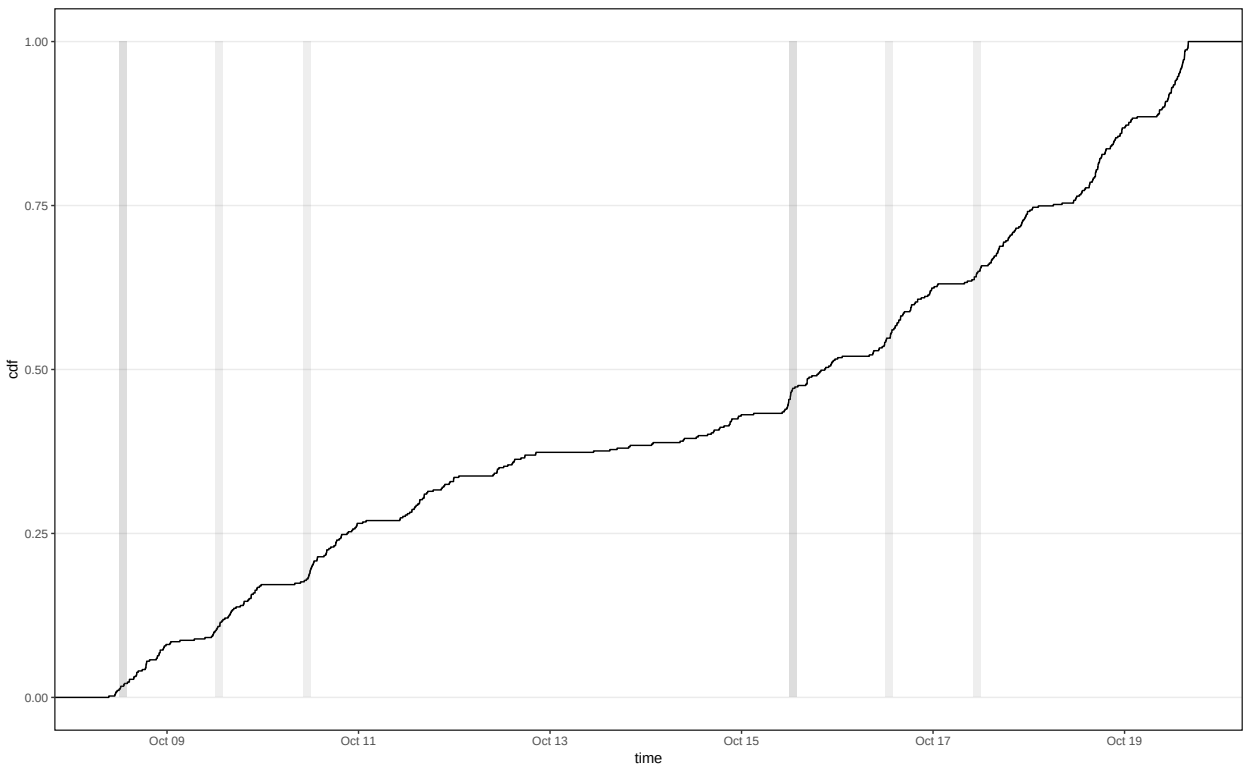
Quiz closes at 16:00 on Friday 26th October

- no extensions
- grade is
  - 0 (for no attempt)
  - $30 + 70 \times (\text{score}/10)^2$

VLE activities (cont'd)

Pairs and vectors quiz

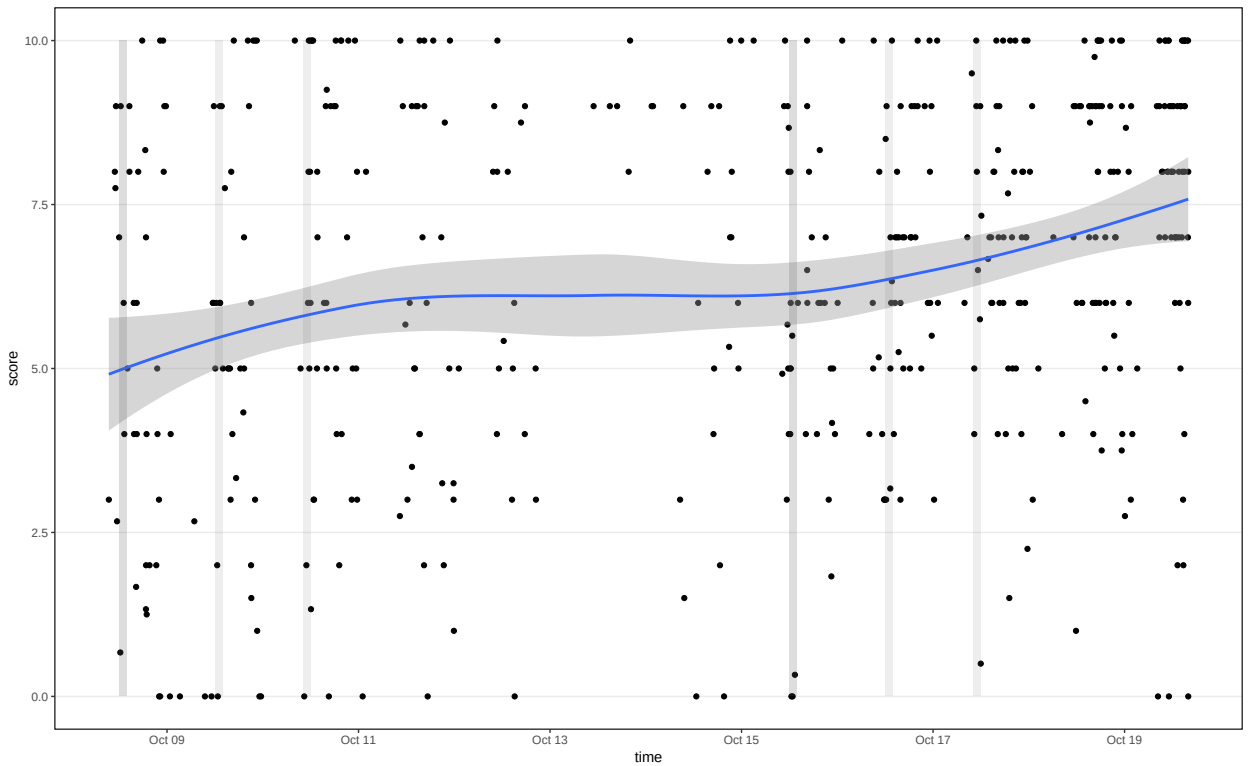
- 471 attempts: average mark 6.38
- 132 students: average mark 8.74
  - 4 under 4.00, 119 above 6.99, 64 at 10



VLE activities (cont'd)

Pairs and vectors quiz

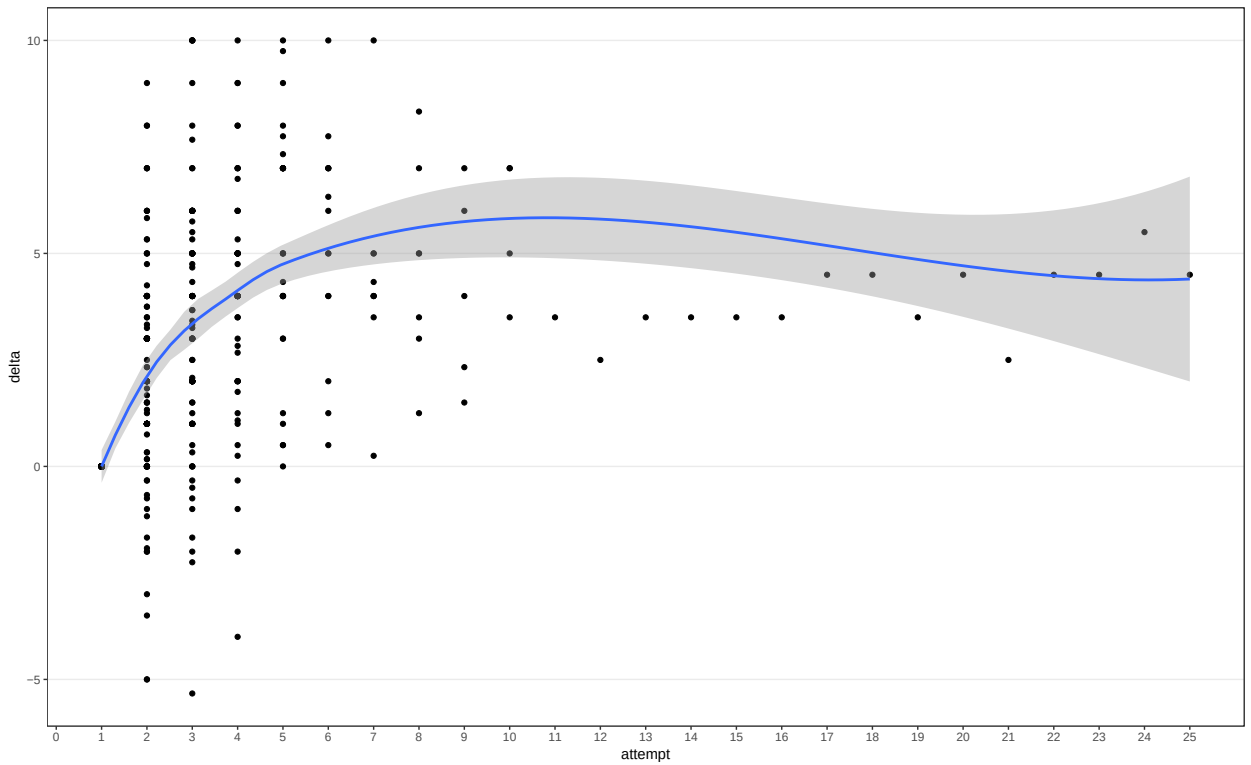
- 471 attempts: average mark 6.38
- 132 students: average mark 8.74
  - 4 under 4.00, 119 above 6.99, 64 at 10



VLE activities (cont'd)

Pairs and vectors quiz

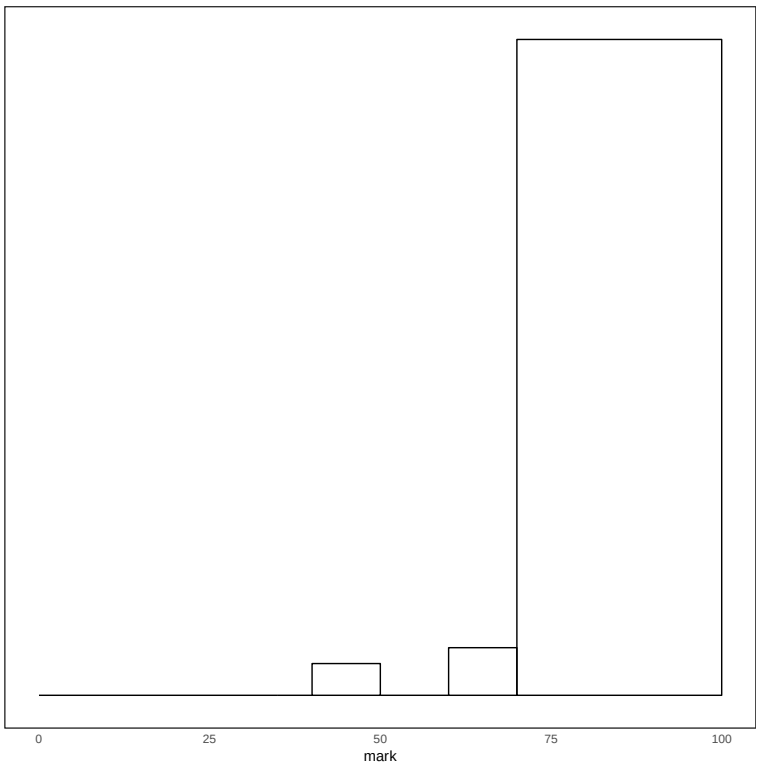
- 471 attempts: average mark 6.38
- 132 students: average mark 8.74
  - 4 under 4.00, 119 above 6.99, 64 at 10



VLE activities (cont'd)

Hello world lab activity

- 129 final uploads: average mark 96.71



Outline

Introduction

Linked lists

Stacks

Queues

Motivation

- simple application of pairs

Motivation

- simple application of pairs
- building block for more complex data structures:
  - stacks
  - queues

Motivation

- simple application of pairs
- building block for more complex data structures:
  - stacks
  - queues
- useful for considering issues in algorithm design:
  - complexity and scaling
  - iteration and recursion

Definition

A linked list is a sequential collection of data

Operations

first return the first element of the list

Operations

first	return the first element of the list
rest	return the list with the first element removed

Operations

first	return the first element of the list
rest	return the list with the first element removed
cons[o]	return a new list whose first is o and whose rest is the list

Operations

first	return the first element of the list
rest	return the list with the first element removed
cons[o]	return a new list whose first is o and whose rest is the list
set-first![o]	set the first of the list to o

Operations

first	return the first element of the list
rest	return the list with the first element removed
cons[o]	return a new list whose first is o and whose rest is the list
set-first![o]	set the first of the list to o
set-rest![l]	set the rest of the list to l

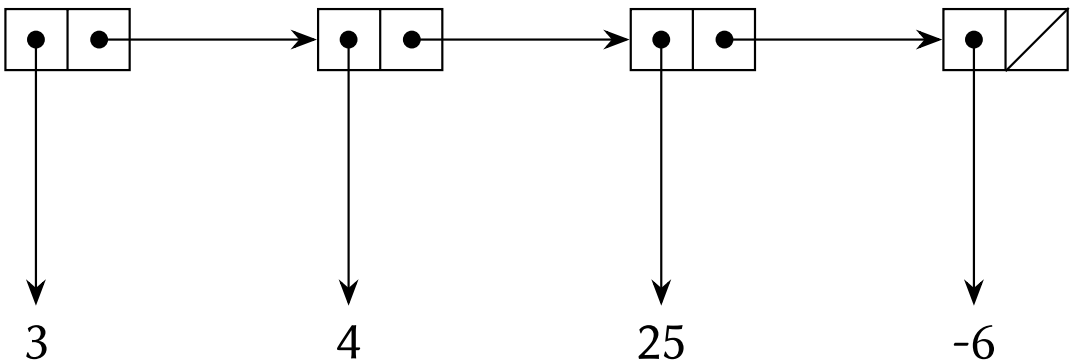
Operations

first	return the first element of the list
rest	return the list with the first element removed
cons[o]	return a new list whose first is o and whose rest is the list
set-first![o]	set the first of the list to o
set-rest![l]	set the rest of the list to l
null?	return `true` if the list is the *empty list*

Special value

NIL the immutable empty list

Implementation



Complexity analysis

first, rest, set-first!, set-rest!

1. pointer read (first, rest) or write (set-first!, set-rest!)

$$\Longrightarrow \Theta(1)$$

Complexity analysis

first, rest, set-first!, set-rest!

1. pointer read (first, rest) or write (set-first!, set-rest!)

$$\Longrightarrow \Theta(1)$$

cons

1. fixed-size (two word) allocation
2. two pointer writes

$$\Longrightarrow \Theta(1)$$

Complexity analysis

first, rest, set-first!, set-rest!

1. pointer read (first, rest) or write (set-first!, set-rest!)

$$\Longrightarrow \Theta(1)$$

cons

1. fixed-size (two word) allocation
2. two pointer writes

$$\Longrightarrow \Theta(1)$$

null?

1. single comparison

$$\Longrightarrow \Theta(1)$$

Complexity analysis

construct a linked list

... with N elements

1. construct N nodes
2. set-first! N times
3. set-rest! $N - 1$ times

$$\Longrightarrow \Theta(N)$$

Complexity analysis

construct a linked list

... with N elements

1. construct N nodes
2. set-first! N times
3. set-rest! $N - 1$ times

$$\Longrightarrow \Theta(N)$$

or

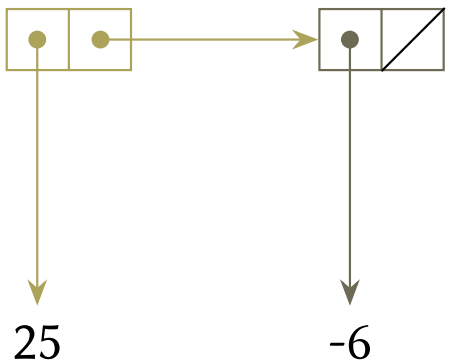
Let the time for constructing a linked list with k elements be T_k

1. construct a list of length $N - 1$: T_{N-1}
2. construct a node: $\Theta(1)$

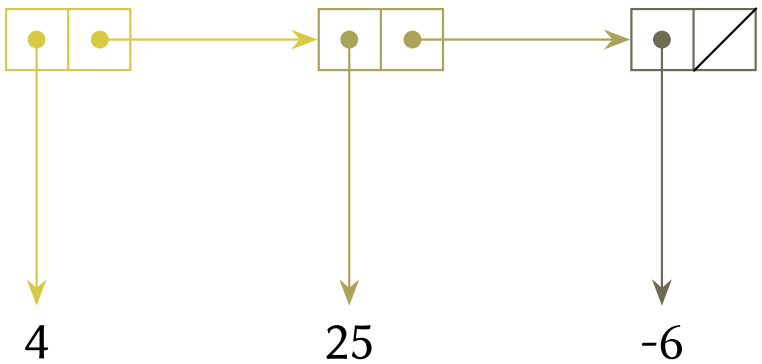
$$\begin{aligned} \Longrightarrow T_N &= T_{N-1} + \Theta(1) \\ &= \Theta(N) \end{aligned}$$

Recursive Data Structure

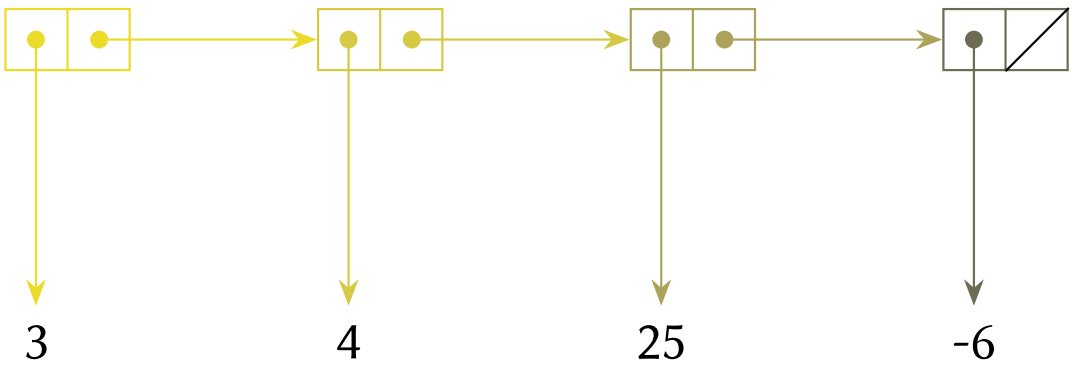
Recursive Data Structure



Recursive Data Structure



Recursive Data Structure



Linear linked list algorithms

Base case

what is the answer for the empty list?

Linear linked list algorithms

Base case

what is the answer for the empty list?

Otherwise

1. compute the answer for the rest of the list
2. modify that answer based on the current node

Example: length

Base case

what is the length of the empty list?

Otherwise

1. what is the length of the rest of the list?
2. how does the length of this list relate to the length of the rest of the list?

Outline

Introduction

Linked lists

Stacks

Queues

Motivation

- last-in first-out collection
- useful for modelling (computational) state:
  - function calls / activation records
  - function-local temporary storage area

Definition

A stack is an extensible linear collection of data whose top element (only) is accessible

Operations

push![o] add o to the top of the stack

Operations

push![o]	add o to the top of the stack
top	return the top element of the stack

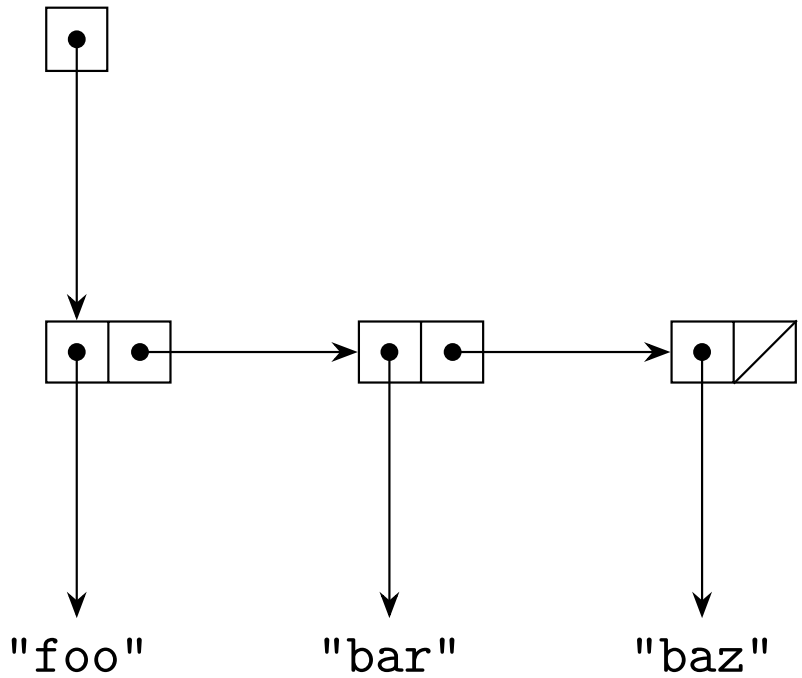
Operations

push![o]	add o to the top of the stack
top	return the top element of the stack
pop!	remove and return the top element of the stack

Operations

push![o]	add o to the top of the stack
top	return the top element of the stack
pop!	remove and return the top element of the stack
empty?	return *true* if the stack has no elements

Implementation



Complexity analysis

top

Two pointer reads $\Longrightarrow \Theta(1)$

Complexity analysis

top

Two pointer reads $\Longrightarrow \Theta(1)$

pop!

Three pointer reads, one pointer write $\Longrightarrow \Theta(1)$

Complexity analysis

top

Two pointer reads $\Longrightarrow \Theta(1)$

pop!

Three pointer reads, one pointer write $\Longrightarrow \Theta(1)$

push!

One pair allocation, one pointer write $\Longrightarrow \Theta(1)$

Complexity analysis

top

Two pointer reads $\Longrightarrow \Theta(1)$

pop!

Three pointer reads, one pointer write $\Longrightarrow \Theta(1)$

push!

One pair allocation, one pointer write $\Longrightarrow \Theta(1)$

empty?

One pointer read, one equality comparison $\Longrightarrow \Theta(1)$

Dynamic arrays, revisited

Can use a dynamic array directly as a stack

push[o]!, pop!	directly supported
top	select[length-1]
empty?	length = 0

Compare with linked list implementation:

- Space complexity:

linked list	2 words per item (+ 1 word overhead)
dynamic array	1 word per item (+ 2+k words overhead)

  dynamic array up to twice as space efficient

- Time complexity:

linked list	operations $\Theta(1)$ in all cases
dynamic array	operations $\Theta(1)$ amortized

Outline

Introduction

Linked lists

Stacks

Queues

Motivation

- first-in first-out collection
- useful for mediating access to resources
  - wait your turn
  - (but see priority-queues)

Definition

A queue is an extensible collection of data where removal of data happens at the head of the queue (maintaining the order of remaining elements), and addition of data happens at the tail of the queue (again maintaining order of other elements)

Operations

head return the element at the head of the queue

Operations

head	return the element at the head of the queue
dequeue!	return and remove the element at the head of the queue

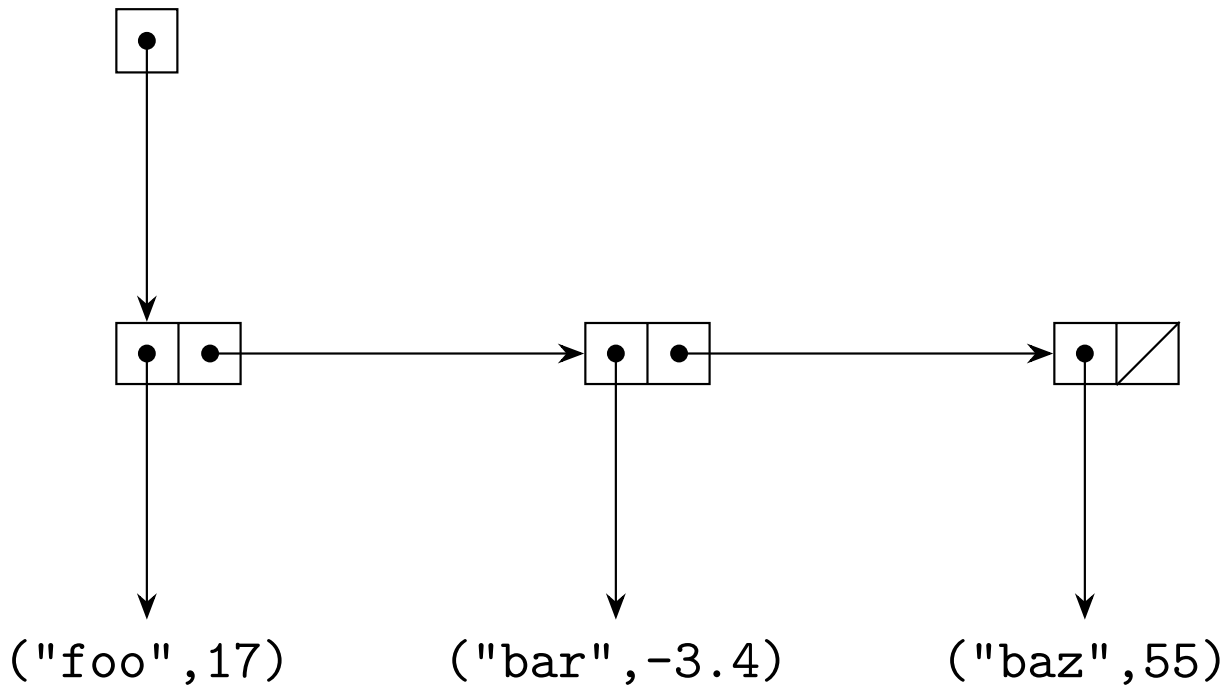
Operations

head	return the element at the head of the queue
dequeue!	return and remove the element at the head of the queue
enqueue![o]	add o to the tail of the queue

Operations

head	return the element at the head of the queue
dequeue!	return and remove the element at the head of the queue
enqueue![o]	add o to the tail of the queue
empty?	return *true* if the queue has no elements.

Implementation



Complexity analysis

head, dequeue!, empty?

Exactly the same as stack operations top, pop!, empty?

$$\Longrightarrow \Theta(1)$$

Complexity analysis

head, dequeue!, empty?

Exactly the same as stack operations top, pop!, empty?

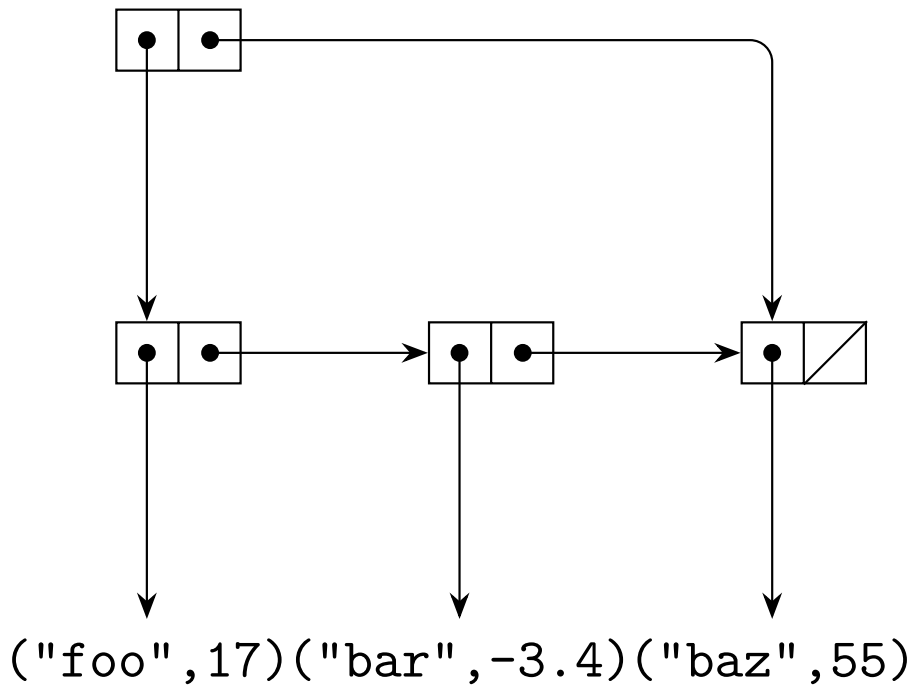
$$\Longrightarrow \Theta(1)$$

enqueue!

Add to the end of the underlying list: N pointer reads

$$\Longrightarrow \Theta(N)$$

Better implementation



Complexity analysis

head, dequeue!, empty?

As before, exactly the same as stack operations top, pop!, empty?

$$\Longrightarrow \Theta(1)$$

enqueue!

One pair allocation, two pointer writes

$$\Longrightarrow \Theta(1)$$

Work

1. implement queues using both variants described above (one with a tail pointer and one without). Count the operations it takes to create queues by successive enqueueing with 10, 100, 1000, 10000 and 100000 elements. Does this support the lecture discussion about complexity analysis?
2. do the stacks and queues quiz on the VLE
   - open from 22nd October 2018
   - closes 2nd November 2018