Week 05 - Lecture 1 Slides

Dr. Yeganeh Bahoo

Created: 2023-10-04 Wed 11:07

Table of Contents

Lecture 1: FIFO & LIFO

Learning objectives: By the end of this lecture you should be able to

- recognize problem properties where stacks, queues, and deques are appropriate data structures
- implement the stack, queue

Queues

- Provides the simplest way for implementing scheduling mechanisms.
- Ordering principle: **first-in first-out (FIFO)**, i.e., the element deleted is the one that has been stored for the longest time
- Applications:
  - printing queues
  - OS control processes

Queues: using an array

Initial array:

head and tail locations			head			tail
array indexes	0	1	2	3	4	5
items			4	12	17	
# of items: 3						

Enqueue 11:

head and tail locations	tail		head			
array indexes	0	1	2	3	4	5
items			4	12	17	11
# of items: 4						

Dequeue:

head and tail locations	tail			head		
array indexes	0	1	2	3	4	5

items			4	12	17	11
# of items: 3						

Exercise

Given the initial array below, what values would be in the array and which ones would represent the head and tail of the queue if the following operations are sequentially executed

```
ENQUEUE 11
ENQUEUE 7
DEQUEUE
DEQUEUE
```

head and tail locations			head			tail
array indexes	0	1	2	3	4	5
items			4	12	17	
# of items: 3						

Solution

head and tail locations			head			tail
array indexes	0	1	2	3	4	5
items			4	12	17	
# of items: 3						

ENQUEUE 11

head and tail locations	tail	head

array indexes	0	1	2	3	4	5
items			4	12	17	11
# of items: 4						

ENQUEUE 7

head and tail locations		tail	head			
array indexes	0	1	2	3	4	5
items	7		4	12	17	11
# of items: 5						

Solution

(cont.)

DEQUEUE

head and tail locations		tail		head		
array indexes	0	1	2	3	4	5
items	7		4	12	17	11
# of items: 4						

DEQUEUE

head and tail locations		tail			head	
array indexes	0	1	2	3	4	5

items	7	4	12	17	11

# of items: 3

Queues: implementation

```
(defparameter *size* 4)

(defstruct queue
  (data (make-array *size*))
  (head 0)
  (tail 0)
  (count 0))

(defun is-empty (q)
  (and (= (queue-head q) (queue-tail q))
       (= (queue-count q) 0)))

(defun is-full (q)
  (and (= (queue-head q) (queue-tail q))
       (= (queue-count q) *size*)))

(defun enqueue (q x)
  (unless (is-full q)
    (incf (queue-count q))
    (setf (aref (queue-data q) (queue-tail q)) x)
    (if (= (queue-tail q) (1- *size*))
        (setf (queue-tail q) 0)
        (setf (queue-tail q) (1+ (queue-tail q))))))

(defun dequeue (q)
  (unless (is-empty q)
    (decf (queue-count q))
    (let ((x (aref (queue-data q) (queue-head q))))
      (if (= (queue-head q) (1- *size*))
          (setf (queue-head q) 0)
          (setf (queue-head q) (1+ (queue-head q))))
      x)))
```

Analysis of DEQUEUE and ENQUEUE

Array-based implementation:

- ENQUEUE and DEQUEUE: $O(1))$
- dynamic resizing of the array requires memory reallocation, $O(n)$

List-based implementation:

- ENQUEUE: $O(1))$
- DEQUEUE: $O(n)$
- Requires a garbage collection mechanism
    - lisp does that effectively for built-in lists

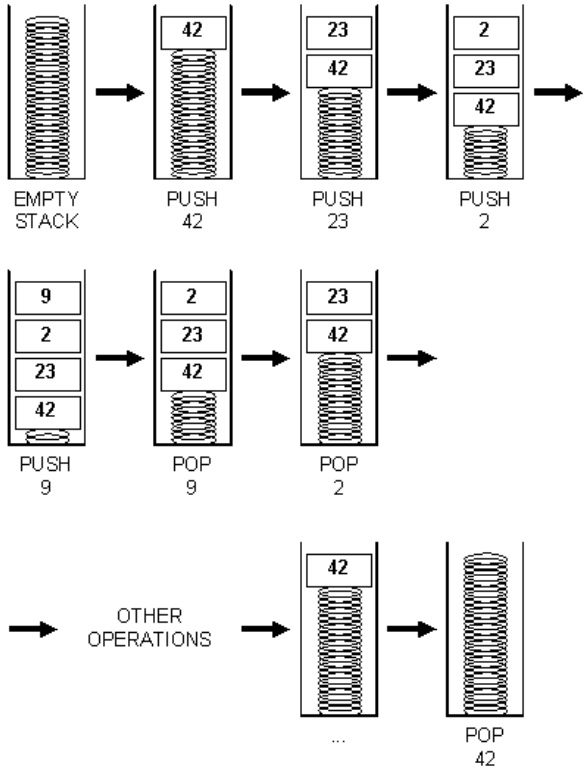
Example of a list-based implementation:

```
(defun enqueue (q x)
  (cons x q))

(defun dequeue (q)
  (subseq q 0 (1- (length q))))
```

```
CL-USER> (enqueue '(3 4 5) 1)
(1 3 4 5)
CL-USER> (dequeue '(1 3 4 5))
(1 3 4)
```

Stack, aka "Push-down stack"



- Provides the simplest way of saving information in a temporary storage location
- Ordering principle: **last-in first-out (LIFO)**
- Applications:
  - backward navigation in web browsers
  - mathematical expression evaluation: `1+3*(2-1)`
  - management of recursive subroutine calling

The stack data structure and interface

- Very simple to implement using a list
- Interface: PUSH, POP, PEEK, IS-EMPTY

Example: using lisp's PUSH and POP functions.

```
RTL-USER> (defvar *stack* '())
*STACK*
RTL-USER> (push 11 *stack*)
(11)
RTL-USER> (push -31 *stack*)
(-31 11)
RTL-USER> (pop *stack*)
-31
RTL-USER> *stack*
(11)
RTL-USEAR> (defun peek (s) (car s))
PEEK
RTL-USER> (peek *stack*)
11
```

Example

Parenthesis checker - a simple application for a stack. Fill in the blanks.

Input	Stack	Stack operation
`[[]]]`		push [
`[]]]`	`[`	push [
`]]]`	`[[`	pop
`]]`	`[`	pop
`]`		

```
(defun par-checker (list)
  "Returns T if the [s in list are matched with their ]s"
  (let ((s nil)) ; initialize stack s
    (dolist (symbol list (null s))
      (case symbol
        ([ (push symbol s))
        (] (if (null s) (return) ; unbalanced []s, breaks from loop returning NIL
               ...))
        (otherwise (return))))))) ; invalid symbol, breaks from loop returning NIL
```

Solution

```
(defun par-checker (list)
  "Returns T if the [s in list are matched with their ]s"
  (let ((s nil)) ; initialize stack s
    (dolist (symbol list (null s))
      (case symbol
        ([ (push symbol s))
        (] (if (null s) (return) ; unbalanced []s
               (pop s)))
        (otherwise (return))))))) ; invalid symbol
```

```
CL-USER> (par-checker '([] []))  ; the symbol [] does not match the cases; has to be [ ]
NIL
CL-USER> (par-checker '([ ] [ ]))
T
CL-USER> (par-checker '([ ] [ [ ] ]))
T
CL-USER> (par-checker '([ ] [ [  ]))
NIL
```