ECE36800 Data structures

Queues

Chapter 4 (pp. 153-161)

Chapter 9 (pp. 383-392)

Lu Su

School of Electrical and Computer Engineering
Purdue University

Slides Courtesy: Prof. Cheng-Kok Koh

Overview

- Queue definition
- Primitive operators
- Linked list implementation
- Array implementation
- Priority queues

- Reference pages: pp. 153-161, pp. 383-392

Queues

- A queue is an ordered collection of items where items are inserted (enqueued) at the end and are removed (dequeued) from the front
- Can also be called FIFO (First In First Out)
- A check-out line at grocery store, for example
  - The line (queue) is either empty or not empty
  - The cashier serves the customer at the front of the line
  - Customers are ordered from the front to the end of line
  - Customers can only join at the end of the line
- Other examples:
  - Printer queues
  - Buffer queue of switching packets in data communication

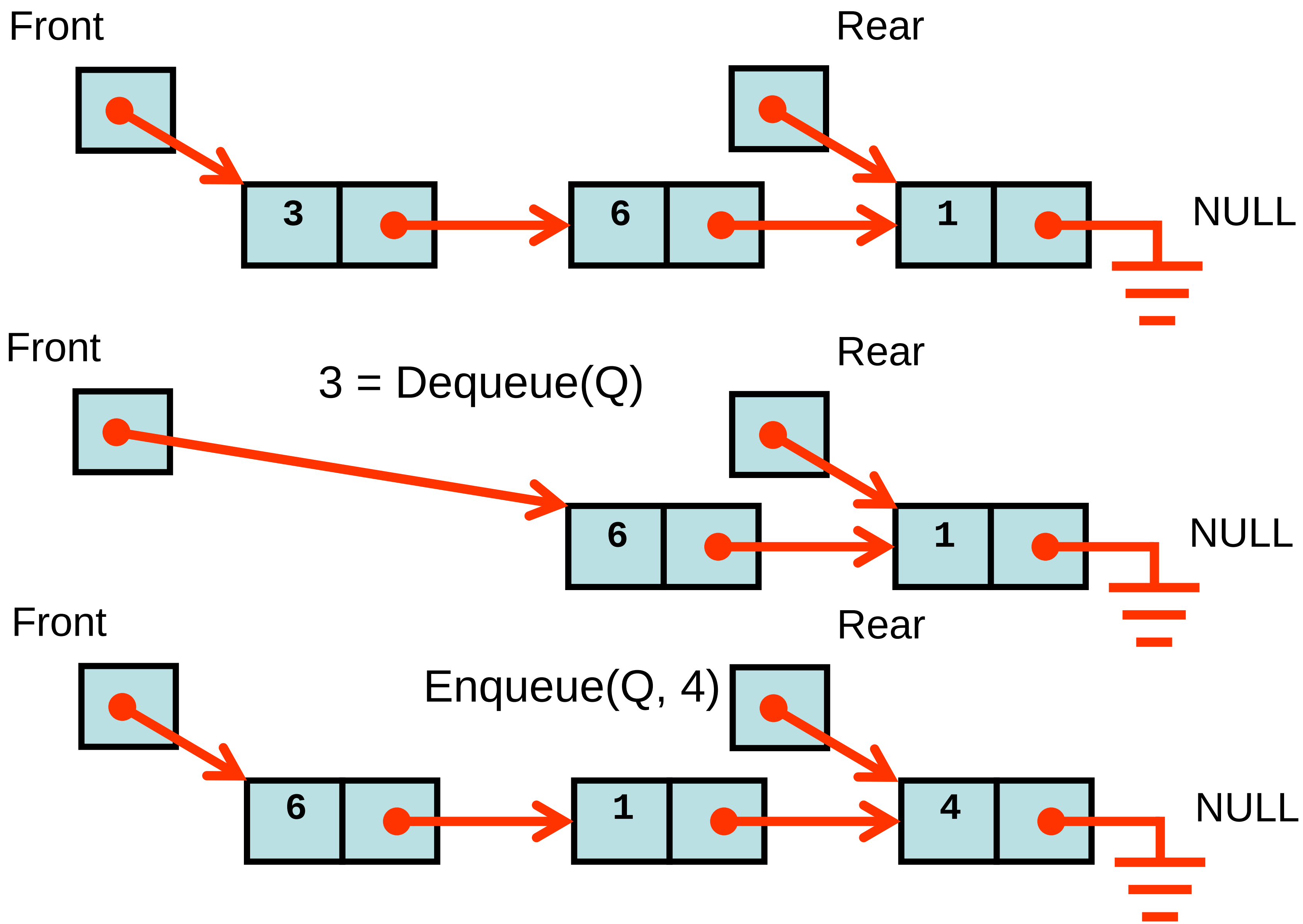
Primitive operators

- Enqueue(Q, i)
  - Add item i at the end/rear of queue Q
  - Must make sure it does not cause overflow
- Dequeue(S)
  - Get the front item of queue Q, remove it from Q and return it, e.g., i = Dequeue(Q)
  - Must make sure that queue was not empty
- Front(Q)
  - Return the front item of queue Q with no change to Q, e.g., i = Front(Q)
  - Must make sure that queue was not empty
- Rear(Q)
  - Return the last item of queue Q with no change to Q, e.g., i = Rear(Q)
  - Must make sure that queue was not empty
- Empty(Q) (or Is_empty(Q))
  - Return true if queue Q is empty; false otherwise

Linked list implementation

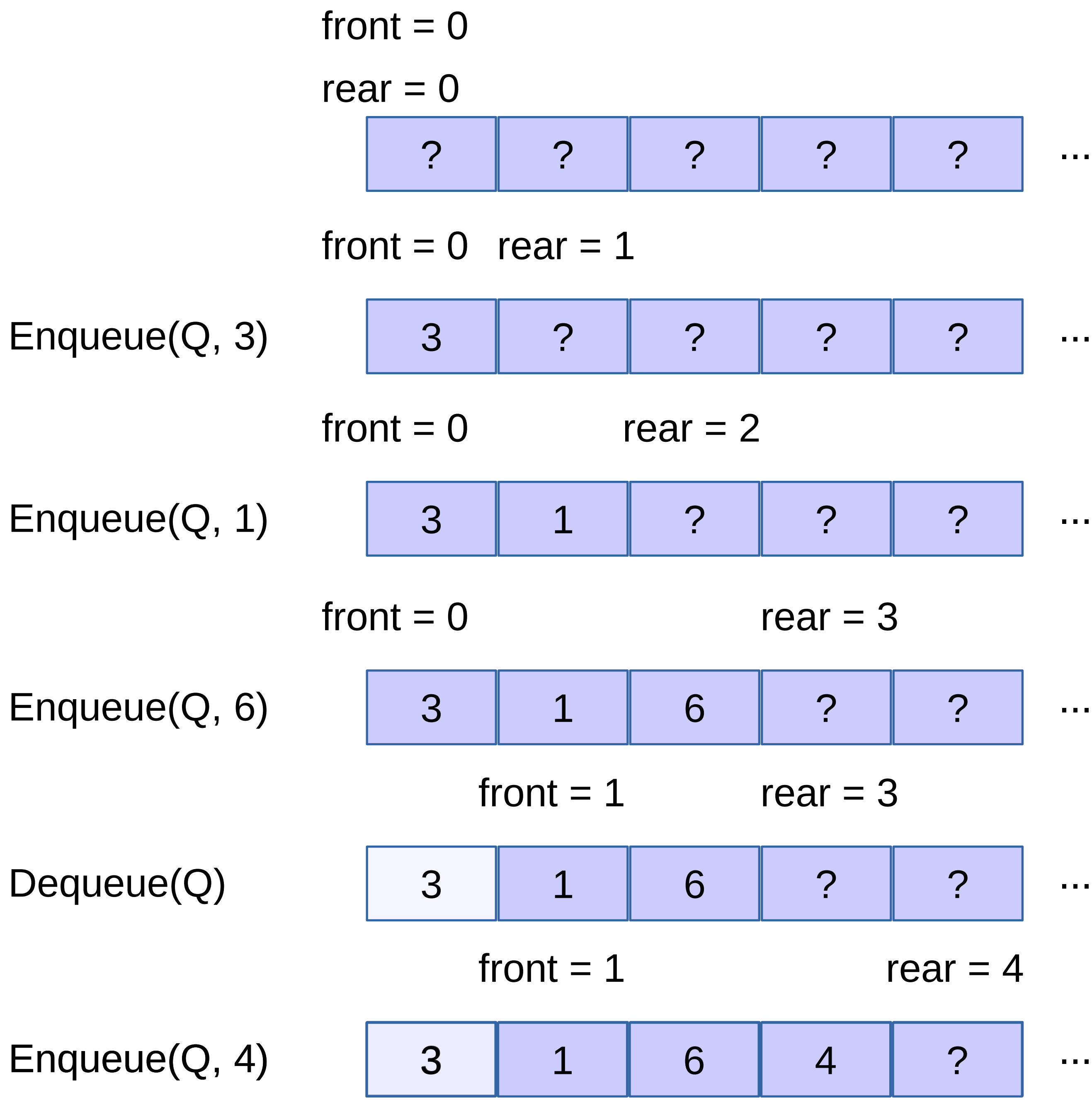
- Seek O(1) time complexity for all primitive operations
- Must maintain both head and tail of linked list for access of front and rear
- Empty list == Empty queue
- Head of list == Queue front
  - Meaningful only if list is not empty
- Tail of list == Queue rear
  - Meaningful only if list is not empty
- Insert at tail = Enqueue
  - To check for overflow, verify that the address returned by malloc is not NULL
- Remove at head = Dequeue
  - Meaningful only if list is not empty
- What would be the time complexity for each of the primitive operations for a queue if the head of the linked list is the queue rear and the tail is the queue front?

Linked list example



Array implementation

- Use a struct to store four fields:
  - Address of the array to store items
  - Total size of the array
  - Index of the queue front
  - Index of the queue rear
- First attempt (assume the array is of infinite size)
  - Assume that front and rear indices initialized to 0 for an empty queue
- We may initialize front and rear differently and would have to implement the primitive operations slightly differently
  - Empty queue == number of items in queue = rear index – front index is 0
  - Queue front == item in array at location indexed by front (if queue not empty)
  - Queue rear == item in array at location indexed by rear – 1 (if queue not empty)
  - Enqueue == Store item in array at location indexed by rear, increment rear
  - Dequeue == Store in a temporary variable the item in the array at location indexed by front (if queue not empty), increment front, return value stored in the temporary variable



3 is not over-written in the array even though it is no longer a valid entry in queue

Array example: array size of 5

front = 1 rear = 4

3	1	6	4	?

front = 2 rear = 4

Dequeue(Q)

3	1	6	4	?

1 is not over-written in the array even though it is no longer a valid entry in queue

front = 3 rear = 4

Dequeue(Q)

3	1	6	4	?

6 is not over-written in the array even though it is no longer a valid entry in queue

front = 3 rear = 5

Enqueue(Q, 2)

3	1	6	4	2

Enqueue(Q, 7)?

Enqueue(Q, 7)

- Resize array (recall dynamically growing stack array by a multiplicative factor)
  - Wasting space vacated by dequeue operations
- Copy all valid entries to the left and update indices, then enqueue
  - O(n), if there are n items in the queue
  - Worst case scenario, 1 empty slot in front, enqueue, dequeue, enqueue, dequeue, ...

front = 3 rear = 5

3	1	6	4	2

front = 0 rear = 2

Copy entries to left

4	2	6	4	2

front = 0 rear = 3

Enqueue(Q, 7)

4	2	7	4	2

Enqueue(Q, 7)

- Use a circular array (or circular buffer), i.e., the index rear = 5 is equivalent to rear = 5 mod 5 = 0
  - We apply modulo 5 because that is the size of the array
  - When we increment the index, instead of doing (rear + 1) or (front + 1), we perform (rear + 1) mod 5 or (front + 1) mod 5
  - From slide 9, we would have the following after Enqueue(Q, 2)

rear = 0 front = 3

Enqueue(Q, 2)

3	1	6	4	2

  - If we ask for the rear element, we perform (rear – 1) mod 5 = (0 – 1) mod 5 = 4 to get the index of the rear element

rear = 1 front = 3

Enqueue(Q, 7)

7	1	6	4	2

- The number of valid items in queue is (rear – front) mod 5
  - (rear – front) mod 5 = (1 – 3) mod 5 = –2 mod 5 = 3

Queue is full

rear = 1 front = 3

Enqueue(Q, 7)

3	1	6	4	2

rear = 2 front = 3

Enqueue(Q, 0)

3	0	6	4	2

front = 3
rear = 3

Enqueue(Q, 8)

3	0	8	4	2

- The number of valid entries in queue = (rear – front) mod 5 = 0
  - Solution 1: We include one more field in the struct for queue to indicate whether a queue is empty or a queue is fully occupied
  - Solution 2: We only fill up to (array size – 1) elements; when (rear + 1) mod 5 is the same as front, the array is deemed “full”

Queue is full: Solution 2

rear = 2 front = 3

Enqueue(Q, 0)

3	0	6	4	2

front = 3
rear = 3

Enqueue(Q, 8)

3	0	8	4	2

front = 3
rear = 3

Reallocate (assume we double the size)

3	0	8	4	2	?	?	?	?	?

Turn it into valid queue by copying 4 and 2 to the right end of the array and update front

rear = 3 front = 8

3	0	8	4	2	?	?	?	4	2

Queue is full: Alternate solution 2

rear = 2 front = 3

Enqueue(Q, 0)

3	0	6	4	2

front = 3
rear = 3

Enqueue(Q, 8)

3	0	8	4	2

front = 3
rear = 3

Reallocate (assume we double the size)

3	0	8	4	2	?	?	?	?	?

front = 3 rear = 8

Turn it into valid queue by copying 3, 0, and 8 to the right end of 4 and 2, and update rear

3	0	8	4	2	3	0	8	?	?

Queue is full

- Typically, pick the left or right partition to copy to the new location based on the size of the partition to minimize overhead
- Can also grow the array first before enqueuing the new element
  - Grow the array
  - If left partition has fewer elements, copy the left partition to the right of the right partition, update rear
  - Else copy the right partition to the right of array, update front
  - Enqueue new element
- Average time complexity of enqueue operation is O(1)

Double-ended queues (deques)

- A queue is an ordered collection of items where items may be inserted (enqueued) and removed (dequeued) from either end (left or right, front or rear, first or last)
- Primitive operators
  - Empty(DQ) or Is_empty(DQ)
  - Left(DQ), Right(DQ)
  - Dequeue_left(DQ), Dequeue_right(DQ)
  - Enqueue_left(DQ, i) or Enqueue_right(DQ, I)
- Doubly-linked list or array implementation would allow O(1) time complexity for all operators
- Can be used to implement stacks or queues

Priority queues (PQs)

- A priority queue is an ordered collection of items where items are removed (dequeued) in the order of their levels of priority
  - Ascending priority queue: A sequence of dequeue operations remove items in ascending order of levels of priority (lowest or minimum first)
  - Descending priority queue: A sequence of dequeue operations remove items in descending order of levels of priority (highest or maximum first)
- Can be used to implement stacks or queues, with the level of priority of an item being the time the item is pushed or enqueued, respectively

Linked list implementation

- Implementation with an unordered linked list
  - O(1) to enqueue
- Insert at the beginning of the list
  - O(n) to dequeue, where n is the number of items in the list
- Must always visit all items to find the item with the minimum or maximum priority for removal
- Implementation with an ordered linked list
  - O(n) to enqueue
- On the average, visit n/2 items for inserting the new item at the correct sorted position (see insertion sort)
  - O(1) to dequeue
- The item with the minimum or maximum priority is at the beginning of the list

Array implementation

- Unordered array
  - O(1) to enqueue
- Insert at the beginning of the list
  - O(n) to dequeue, where n is the number of items in the list
- Must always visit all items to find the item with the minimum or maximum priority for removal
- Ordered array (ascending array for descending priority queue and descending array for ascending priority queue)
  - O(n) to enqueue
- This is simply insertion of a new item into a sorted array
- On the average, visit n/2 items for inserting the new item at the correct sorted position
  - O(1) to dequeue
- The item with the minimum or maximum priority is at the (right) end of the array

A better array implementation

- An array can be interpreted as a tree, with indices of items determining their positions within the tree
  - Called a heap
  - A max-heap implements a descending priority queue and a min-heap implements an ascending priority queue
  - O(log n) for enqueue and dequeue, where n is the number of items in the heap
  - Will elaborate on heaps when we cover heap sort