Stacks and Queues

4/15/26

Green slides based on ones by Cynthia Lee

Administrivia

- HW 3 (testing and implementing list methods) due tonight

- Exam 1 on Friday (4/17)
  - In-class, open-notes, on-paper exam
  - Codingbat-like problems, testing, array-based list methods, linked list methods, running times

Recall: Asymptotic running time

Want to ignore

a) multiplicative constants

b) behavior at “small” n

Big-O notation:

f(n) = O(g(n)) if there exist constants n_0 and c such that $f(n) \leq cg(n)$ for all $n \geq n_0$

Looking at worst case time

What most accurately characterizes the running time of the array-based implementation of add?

A. O(1)
B. O(n)

What most accurately characterizes the running time of the array-based implementation of add?

A. O(1)
B. O(n)

What most accurately characterizes the running time of the array-based implementation of add?

A. O(1)

B. O(n)

But… if you double the array when you resize, resizes are rare enough that any sequence of operations takes constant time per operation on average. We call this constant *amortized time*.

Stack ADT

- Stores collection of items in the order opposite to the one in which they were added

- Supports:

    void push(Item i)

    Item pop()

    Item peek()

    boolean isEmpty()

    int size()

One application: program stack

- Each function call is represented by an *activation record* or *activation frame* stored in a stack in memory
  - This contains values of local variables and return address
- New function calls are pushed onto stack and returning calls pop off it

Another application: “Undo”

- Many applications have an “Undo” button to reverse the last operation
- Can use a stack to support repeated “Undo”s

Implementing a stack

- Array-based
  - store values with oldest at cell 0
  - keep track of top index (just like num)

- Linked memory
  - keep reference (top) to Node at top of the stack

What is the running time of the linked-memory implementation of the Stack's push method (add to front)?

A. O(1)
B. O(n)

What is the running time of the linked-memory implementation of the Stack's push method (add to front)?

A. O(1)
B. O(n)

What is the running time of the linked-memory implementation of the Stack's pop method (remove first)?

A. O(1)
B. O(n)

What is the running time of the linked-memory implementation of the Stack's pop method (remove first)?

A. O(1)
B. O(n)

What is the running time of the array-based implementation of the Stack's push method (add to end)?

A. O(1)
B. O(n)

What is the running time of the array-based implementation of the Stack's push method (add to end)?

A. O(1)
B. O(n) (but O(1) amortized time)

What is the running time of the array-based implementation of the Stack's pop method (remove from end)?

A. O(1)
B. O(n)

What is the running time of the array-based implementation of the Stack's pop method (remove from end)?

A. O(1)
B. O(n)

Stack running times

	push	pop
array-based	Linear (constant amortized)	Constant
linked list	Constant	Constant

Queue ADT

- Stores collection of items in the order they were added to the queue

- Supports:

  Queue()

  void enqueue(Item i)

  Item dequeue()

  boolean isEmpty()

  int size()

When an item is removed from a queue, it is...?

A. The one that has been in the queue for the longest time

B. The one that has been in the queue for the shortest time

C. The smallest one in the queue

D. The largest one in the queue

E. Not exactly one of the above

When an item is removed from a queue, it is...?

A. The one that has been in the queue for the longest time

B. The one that has been in the queue for the shortest time

C. The smallest one in the queue

D. The largest one in the queue

E. Not exactly one of the above

When using enqueue() to place the following items into a queue:

enqueue(32)
enqueue(65)
enqueue(0)
enqueue(23)
enqueue(-1)

the output when dequeueing from the queue is:

A. -1, 23, 0, 65, 32
B. -1, 0, 23, 32, 65
C. 65, 32, 23, 0, -1
D. 32, 65, 0, 23, -1
E. None of the above

When using enqueue() to place the following items into a queue:

enqueue(32)
enqueue(65)
enqueue(0)
enqueue(23)
enqueue(-1)

the output when dequeueing from the queue is:

A. -1, 23, 0, 65, 32
B. -1, 0, 23, 32, 65
C. 65, 32, 23, 0, -1
D. **32, 65, 0, 23, -1**
E. None of the above

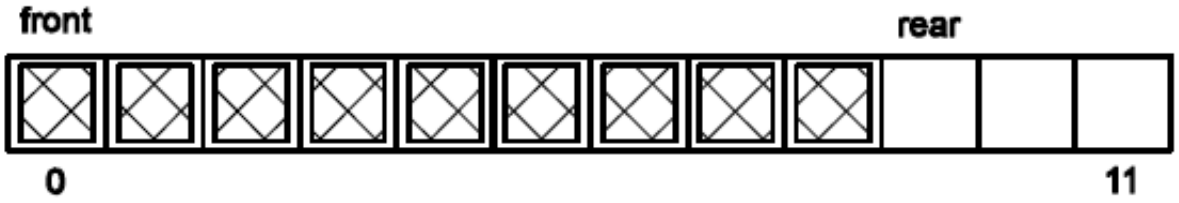
Applications of Queue

- "Calls will be answered in the order in which they are received"
  - Customer service
  - Print jobs
  - Packets of a network message
  - …

  Though sometimes things seem like they could be a queue, but aren't…

Queue implementation 1

- Array with front element of queue at index 0
- Attribute rear is index of first unoccupied cell
  - Empty queue denoted by rear being 0



(a) Queue.*front* is always at 0 – shift elements *left* on dequeue().

```
public E dequeue(){
  // potential issue if empty, for now, assume not empty
  E e = array[front];
  <YOUR CODE HERE>
  return e;
}
```

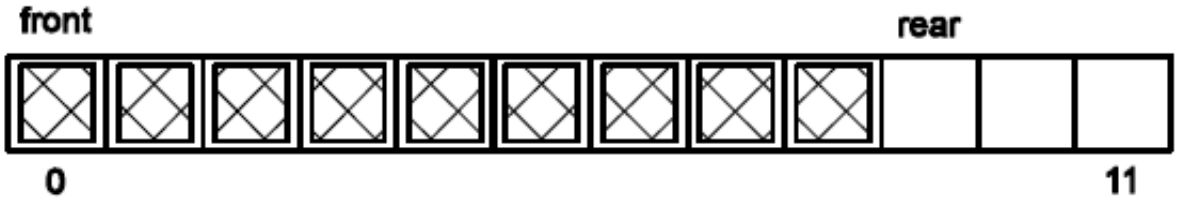
Select the correct code to insert from below:

A. front++;

B. rear = rear-1;

C. for(int i= 0; i<rear-1; i++)
   array[i] = array[i+1];
   rear = rear - 1;

D. None of these are correct



(a) Queue.*front* is always at 0 – shift elements *left* on dequeue().

```
public E dequeue(){
  // potential issue if empty, for now, assume not empty
  E e = array[front];
  <YOUR CODE HERE>
  return e;
}
```

Select the correct code to insert from below:

A. front++;

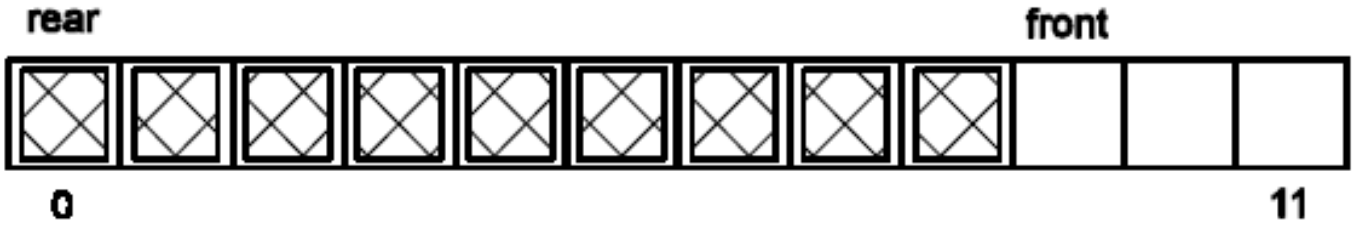
B. rear = rear-1;

C. for(int i= 0; i<rear-1; i++)
   array[i] = array[i+1];
   rear = rear - 1;

D. None of these are correct

Queue implementation 2

- Array with rear element of queue at index 0
- Attribute front is index of the first unoccupied cell after the queue
  - Empty queue denoted by front being 0



(b) Queue.*rear* is always at 0 – shift elements *right* on enqueue().

```
public void enqueue( E element){
  // potential issue if full, for now, assume room
  <YOUR CODE HERE>
  front++;
}
```

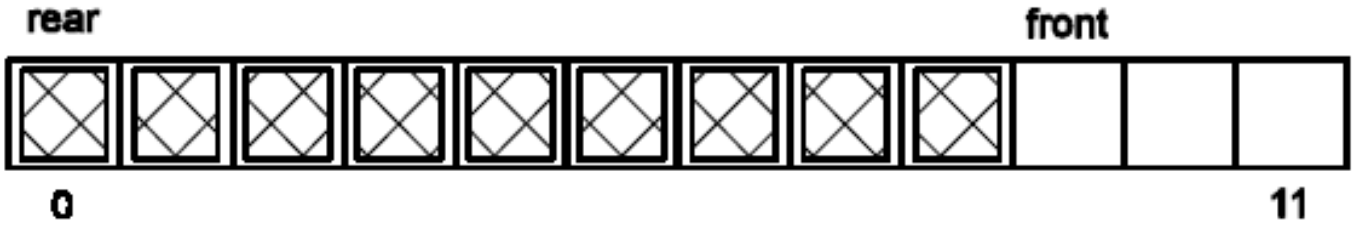
Select the correct code to insert from below:

A
```
array[0] = element;
```

B
```
array[front] = element;
```

C
```
for(int i=0; i<front; i++) {
  array[i+1] = array[i];
}
array[rear] = element;
```

D None of these are correct



(b) Queue.*rear* is always at 0 – shift elements *right* on enqueue().

```
public void enqueue( E element){
  // potential issue if full, for now, assume room
  <YOUR CODE HERE>
  front++;
}
```

Select the correct code to insert from below:

A
```
array[0] = element;
```

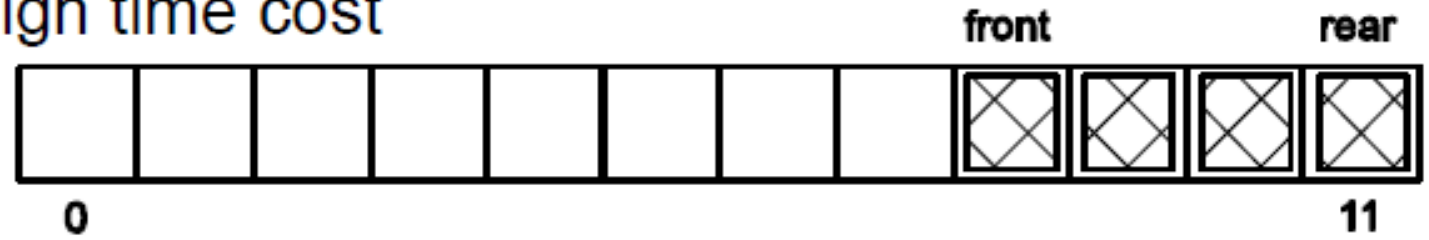
B
```
array[front] = element;
```

C
```
for(int i=0; i<front; i++) {
   array[i+1] = array[i];
}
array[rear] = element;
```

D None of these are correct

ArrayQueue: another option

- Neither of those solutions is very good as they both involve *moving all the existing* data elements, which has high time cost



- Idea: Instead of moving data elements to a fixed position for *front* when removing, let *front* advance through the array

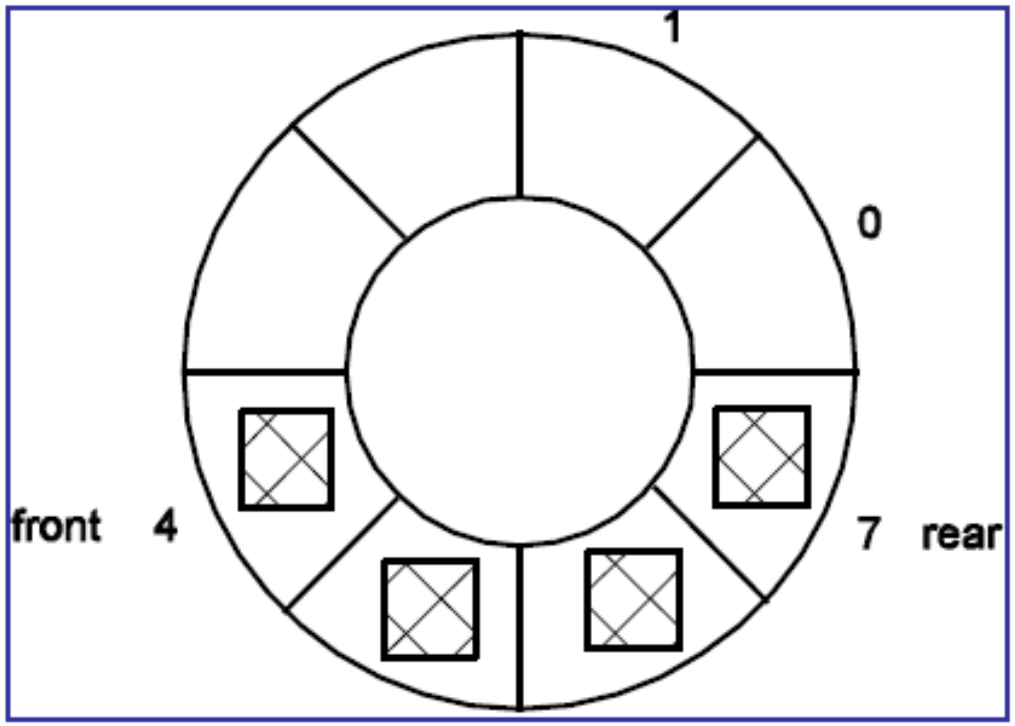
Hmmm….what do we do when we now add an element to that queue at the rear? What happens when we remove several elements, and *front* catches up with *rear*?...

ArrayQueue: Using a *circular* array underlying data structure

Solution: Be more creative!

View the array as *circular* and allow both *front* and *rear* to advance through (around) the array.

This will require *no* data movement for enqueues or dequeues!



front = index of first occupied cell in queue

rear = index of last occupied cell in queue

Trick for implementing circular arrays

- Modular arithmetic:

  Increment: i = (i + 1) % vals.length;

  Decrement: i = (i – 1 + vals.length) % vals.length;

Queue running times

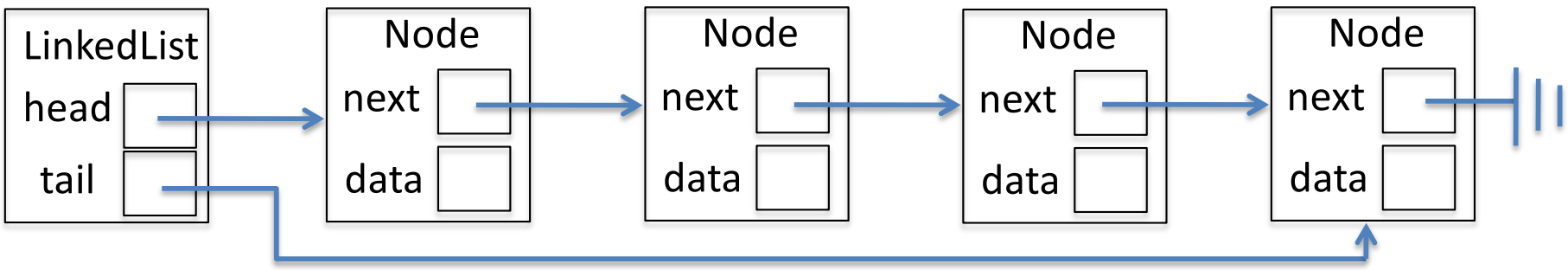
	enqueue	dequeue
using a circular array		
linked list (enqueue to last)		

Queue running times

	enqueue	dequeue
using a circular array	Linear (constant amortized)	Constant
linked list (enqueue to last)	Linear	Constant

For linked list, whichever operation changes the end of list takes linear time

Recall: Adding a tail pointer



"Tail pointer" (tail reference) refers to last Node in the list

Queue running times

	enqueue	dequeue
using a circular array	Linear (constant amortized)	Constant
linked list w/ tail pointer (enqueue to last)	Constant	Constant